Carbink: Fault-Tolerant Far Memory

Yang Zhou[†*] Hassan M.G. Wassel[‡] Sihang Liu[§*] Jiaqi Gao[†] James Mickens[†] Minlan Yu[†‡]
Chris Kennelly[‡] Paul Turner[‡] David E. Culler[‡] Henry M. Levy[‖‡] Amin Vahdat[‡]

[†]*Harvard University* [‡]*Google* [§]*University of Virginia* [‖]*University of Washington*

Abstract

Far memory systems allow an application to transparently access local memory as well as memory belonging to remote machines. Fault tolerance is a critical property of any practical approach for far memory, since machine failures (both planned and unplanned) are endemic in datacenters. However, designing a fault tolerance scheme that is efficient with respect to both computation and storage is difficult. In this paper, we introduce Carbink, a far memory system that uses erasure-coding, remote memory compaction, one-sided RMAs, and offloadable parity calculations to achieve fast, storage-efficient fault tolerance. Compared to Hydra, a state-of-the-art fault-tolerant system for far memory, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher memory usage.

1 Introduction

In a datacenter, matching a particular application to just enough memory and CPUs is hard. A commodity server tightly couples memory and compute, hosting a fixed number of CPUs and RAM modules that are unlikely to exactly match the computational requirements of any particular application. Even if a datacenter contains a heterogeneous mix of server configurations, the load on each server (and thus the amount of available resources for a new application) changes dynamically as old applications exit and new applications arrive. Thus, even state-of-the-art cluster schedulers [51, 52] struggle to efficiently bin-pack a datacenter's aggregate collection of CPUs and RAM. For example, Google [52] and Alibaba [34] report that the average server has only ~60% memory utilization, with substantial variance across machines.

Memory is a particularly vexing resource for two reasons. First, for several important types of applications [19, 20, 33, 54], the data set is too big to fit into the RAM of a single machine, even if the entire machine is assigned to a single application instance. Second, for these kinds of applications, alleviating memory pressure by swapping data between RAM and storage [14] would lead to significant application slowdowns, because even SSD accesses are orders of magnitude slower than RAM accesses. For example, Google runs a graph analysis engine [28] whose data set is dozens of GBs in size. This workload runs 46% faster when it shuffles data purely through RAM instead of between RAM and SSDs.

Disaggregated datacenter memory [2, 5, 15, 16, 22, 44, 46] is a promising solution. In this approach, a CPU can be paired with an arbitrary set of possibly-remote RAM modules, with a fast network interconnect keeping access latencies to far memory small. From a developer's perspective, far memory can be exposed to applications in several ways. For example, an OS can treat far RAM as a swap device, transparently exchanging pages between local RAM and far RAM [5, 22, 46]. Alternatively, an application-level runtime like AIFM [44] can expose remotable pointer abstractions to developers, such that pointer dereferences (or the runtime's detection of high memory pressure) trigger swaps into and out of far memory.

Much of the prior work on disaggregated memory [2, 44, 55] has a common limitation: a lack of fault tolerance. Unfortunately, in a datacenter containing hundreds of thousands of machines, faults are pervasive. Many of these faults are planned, like the distribution of kernel upgrades that require server reboots, or the intentional termination of a low-priority task when a higher-priority task arrives. However, many server faults are unpredictable, like those caused by hardware failures or kernel panics. Thus, any *practical* system for far memory has to provide a scalable, fast mechanism to recover from unexpected server failures. Otherwise, the failure rate of an application using far memory will be much higher than the failure rate of an application that only uses local memory; the reason is that the use of far memory increases the set of machines whose failure can impact an application [8].

Some prior far-memory systems do provide fault tolerance via replication [5, 22, 46]. However, replication-based approaches suffer from high storage overheads. Hydra [29] uses erasure coding, which has smaller storage penalties than replication. However, Hydra's coding scheme stripes a single memory page across multiple remote nodes. This means that a compute node requires multiple network fetches to reconstruct a page; furthermore, computation over that page cannot be outsourced to remote memory nodes, since each node contains only a subset of the page's bytes.

[*]Contributed to this work during internships at Google.

In this paper, we present Carbink,[1] a new framework for far memory that provides efficient, high-performance fault recovery. Like (non-fault-tolerant) AIFM, Carbink exposes far memory to developers via application-level remoteable pointers. When Carbink's runtime must evict data from local RAM, Carbink writes erasure-coded versions of that data to remote memory nodes. The advantage of erasure coding is that it provides equivalent redundancy to pure replication, while avoiding the double or triple storage overheads that replication incurs. However, straightforward erasure coding is a poor fit for the memory data created by applications written in standard programming languages like C++ and Go; those applications allocate variable-sized memory objects, but erasure coding requires equal-sized blocks. To solve this problem, Carbink eschews the object-granularity swapping strategy of AIFM, and instead swaps at the granularity of *spans*. A single span consists of multiple memory pages that contain objects with similar sizes. Carbink's runtime asynchronously and transparently moves local objects within the spans in local memory, grouping cold objects together and hot objects together. When necessary, Carbink batch-evicts cold spans, calculating parity bits for those spans at eviction time, and writing the associated fragments to remote memory nodes. Carbink utilizes one-sided remote memory accesses (RMAs) to efficiently perform swapping activity, minimizing network utilization. Unlike Hydra, Carbink's erasure coding scheme allows a compute node to fetch a far memory region using a single network request.

In Carbink, each span lives in exactly one place: the local RAM of a compute node, or the far RAM of a memory node. Thus, swapping a span from far RAM to local RAM creates dead space (and thus fragmentation) in far RAM. Carbink runs pauseless defragmentation threads in the background, asynchronously reclaiming space to use for later swap-outs.

We have implemented Carbink atop our datacenter infrastructure. Compared to Hydra, Carbink has up to 29% lower tail latency and 48% higher application performance, with at most 35% more remote memory usage. Unlike Hydra, Carbink also allows computation to be offloaded to remote memory nodes.

In summary, this paper has four contributions:

- a span-based approach for solving the size mismatch between the granularity of erasure coding and the size of the objects allocated by compute nodes;
- new algorithms for defragmenting the RAM belonging to remote memory nodes that store erasure-encoded spans;
- an application runtime that hides spans, object migration within spans, and erasure coding from application-level developers; and
- a thorough evaluation of the performance trade-offs made by different approaches for adding fault tolerance to far memory systems.

[1] Carbink is a Pokémon that has a high defense score.

2 Background

Recent work on far memory has used one of two approaches. The first approach modifies the OS that runs applications, exploiting the fact that preexisting OS abstractions already decouple application-visible in-memory data from the backing storage hierarchy. For example, INFINISWAP [22], Fastswap [5], and LegoOS [46] leverage virtual memory support to swap application memory to far RAM instead of a local SSD or hard disk. Applications use standard language-level pointers to interact with memory objects; behind the scenes, the OS swaps pages between local RAM and far RAM, e.g., in response to page faults for non-locally-resident pages. In contrast, the remote region approach [2] exposes far memory via file system abstractions. Applications name remote memory regions using standard filenames, and interact with regions using standard file operations like `open()` and `read()`.

Exposing far memory via OS abstractions is attractive because it requires minimal changes to application-level code. However, invasive kernel changes are needed; such changes require substantial implementation effort, and are difficult to maintain as other parts of the kernel evolve.

The second far-memory approach requires more help from application-level code. For example, AIFM [44] uses a modified C++ runtime to hide the details of managing far memory. The runtime provides special pointer types whose dereferencing may trigger the swapping of a remote C++-level object into local RAM. AIFM's runtime tracks object hotness using GC-style read/write barriers, and uses background threads to swap out cold local objects when local memory pressure is high. To synchronize the local memory accesses generated by application threads and runtime threads, AIFM embeds a variety of metadata bits (e.g., `present`, `isBeingEvicted`) in each smart pointer, leveraging an RCU-like scheme [36] to protect concurrent accesses to a pointer's referenced object.

Listing 1 provides an example of how applications use AIFM's smart pointers. Like AIFM, Carbink exposes far memory via smart pointers, but unlike AIFM, Carbink provides fault tolerance.

3 Carbink Design

Figure 1 depicts the high-level architecture of Carbink. **Compute nodes** execute single-process (but potentially multi-threaded) applications that want to use far memory. **Memory nodes** provide far memory that compute nodes use to store application data that cannot fit in local RAM. A logically-centralized **memory manager** tracks the liveness of compute nodes and memory nodes. The manager also coordinates the assignment of far memory **regions** to compute nodes. When a memory node wants to make a local memory region available to compute nodes, the memory node *registers* the region with the memory manager. Later, when a compute node requires far memory, the compute node sends an *allocation* request to the memory manager, who then assigns a registered, unallo-

```
RemUniquePtr<Node> rem_ptr = AIFM::MakeUnique<Node>();
{
  DerefScope scope;
  Node* normal_ptr = rem_ptr.Deref(scope);
  computeOverNodeObject(normal_ptr);
} // Scope is destroyed; Node object can be evicted.
```

Listing 1: Example of how AIFM applications interact with far memory. In the code above, the application first allocates a `Node` object that is managed by a particular `RemUniquePtr`. Such a remote unique pointer represents a pointer to an object that (1) can be swapped between local and far memory, and (2) can only be pointed to by a single application-level pointer. The code then creates a new scope via an open brace, declares a `DerefScope` variable, and invokes the `RemUniquePtr`'s `Deref()` method, passing the `DerefScope` variable as an argument. `Deref()` essentially grabs an RCU lock on the remotable memory object, and returns a normal C++ pointer to the application. After the application has finished using the normal pointer, the scope terminates and the destructor of the `DerefScope` runs, releasing the RCU lock and allowing the object to be evicted from local memory.

cated region. Upon receiving a *deallocation* message from a compute node, the memory manager marks the associated region as available for use by other compute nodes. A memory node can ask the memory manager to *deregister* a previously registered (but currently unallocated) region, withdrawing the region from the global pool of far memory.

Carbink does not require participating machines to use custom hardware. For example, any machine in a datacenter can be a memory node if that machine runs the Carbink memory host daemon. Similarly, any machine can be a compute node if that node's applications use the Carbink runtime.

From the perspective of an application developer, the Carbink runtime allows a program to dynamically allocate and deallocate memory objects of arbitrary size. As described in Section 3.2, programs access those objects through AIFM-like remotable pointers [44]. When applications dereference pointers that refer to non-local (i.e., swapped-out) objects, Carbink pulls the desired objects from far memory. Under the hood, Carbink's runtime manages objects using **spans** (§3.3) and **spansets** (§3.4). A span is a contiguous run of memory pages; a single region allocated by a compute node contains one or more spans. Similar to slab allocators like Facebook's jemalloc [17] and Google's TCMalloc [21, 24], Carbink rounds up each object allocation to the bin size of the relevant span, and aligns each span to the page size used by compute nodes and memory nodes. Carbink swaps far memory into local memory at the granularity of a span; however, when local memory pressure is high, Carbink swaps local memory out to far memory at the granularity of a spanset (i.e., a collection of spans of the same size). In preparation for

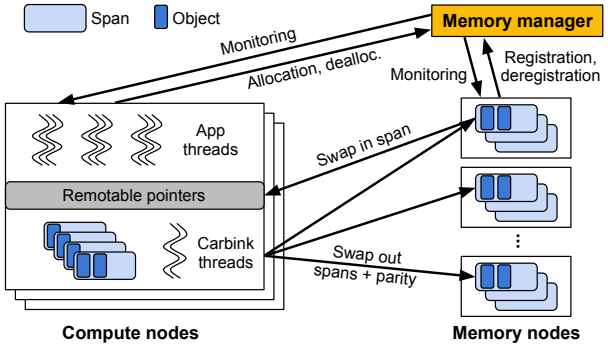


Figure 1: Carbink's high-level architecture.

swap-outs, background threads on compute nodes group cold objects into cold spans, and bundle a group of cold spans into a spanset; at eviction time, the threads generate erasure-coding parity data for the spanset, and then evict the spanset and the parity data to remote nodes. As we discuss in Sections 3.4 and 3.5, this approach simplifies memory management and fault tolerance.

Carbink disallows cross-application memory sharing. This approach is a natural fit for our target applications, and has the advantage of simplifying failure recovery and avoiding the need for expensive coherence traffic [46].

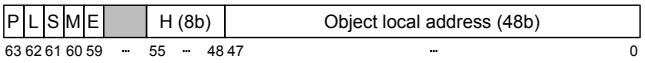
3.1 Failure Model

Carbink implements the logically-centralized memory manager as a replicated state machine [1, 45]. Thus, Carbink assumes that the memory manager will not fail. Carbink assumes that memory nodes and compute nodes may experience fail-stop faults. Carbink does not handle Byzantine failures or partial network failures.

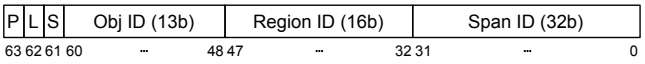
The memory manager tracks the liveness of compute nodes and memory nodes via heartbeats. When a compute node fails, the memory manager instructs the memory nodes to deallocate the relevant spans; if applications desire, they can use an application-level fault tolerance scheme like checkpointing to ensure that application-level data is recoverable. When a memory node fails, the memory manager deregisters the node's regions from the global pool of far memory. However, erasure-coding recovery of the node's regions is initiated by a compute node when the compute node unsuccessfully tries to read or write a span belonging to the failed memory node. If an application thread on a compute node tries to read a span that is currently being recovered, the read will use Carbink's degraded read protocol (§3.5), reconstructing the span using data from other spans and parity blocks.

3.2 Remotable Pointers

Like AIFM, Carbink exposes far memory through C++-level smart pointers. However, as shown in Figure 2, Carbink uses a different pointer encoding to represent span information.



(a) Local object.



(b) Far object.

Field	Meaning
Present	Is the object in local RAM or far RAM?
Lock	Is the object (spin)locked by a thread?
Shared	Is the pointer a unique pointer or a shared pointer?
Moving	Is the object being moved by a background thread?
Evicting	Is the object being evicted by a background thread?
Hotness	Is the object frequently accessed?

(c) Field semantics.

Figure 2: Carbink's `RemUniquePtr` representation. In contrast to AIFM [44], Carbink does not embed information about a data structure ID or an object size. Instead, Carbink embeds span metadata (namely, a Region ID and a Span ID) to associate a pointed-to object with its backing span.

A Carbink `RemUniquePtr` has the same size as a traditional `std::unique_ptr` (i.e., 8 bytes). The **P**resent bit indicates whether the pointed-to object resides in local RAM. The **S**hared bit indicates whether a pointer implements unique-pointer semantics or shared-pointer semantics; the former only allows a single reference to the pointed-to object. The **L**ock, **M**oving, and **E**victing bits are used to synchronize object accesses between application threads and Carbink's background threads (§3.6). The **H**otness byte is consulted by the background threads when deciding whether an object is cold (and thus a priority for eviction).

If an object is local, the local virtual address of the object is directly embedded in the pointer. If an object has been evicted, the pointer describes how to locate the object. In particular, the Obj ID indicates the location of an object within a particular span; the Span ID identifies that span; and the Region ID denotes the far memory region that contains the span.

Carbink supports two smart pointer types: `RemUniquePtr`, which only allows one reference to the underlying object, and `RemSharedPtr`, which allows multiple references. When moving or evicting an object, Carbink's background threads need a way to locate and update the smart pointer(s) which reference the object. To do so, Carbink uses AIFM's approach of embedding a "reverse pointer" in each object; the reverse pointer points to the object's single `RemUniquePtr`, or to the first `RemSharedPtr` that references the object. An individual `RemSharedPtr` is 16 bytes large, with the last 8 bytes storing a pointer that references the next `RemSharedPtr` in the list. Thus, Carbink's runtime can find all of an object's `RemSharedPtrs` by discovering the first one via the object's reverse pointer, and then iterating across the linked list.

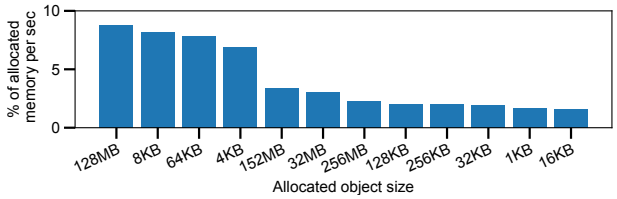


Figure 3: Allocation sizes in our production workloads.

3.3 Span-Based Memory Management

Local memory management: A span is a contiguous set of pages that contain objects of the same size class. Carbink supports 86 different size classes, and aligns each span on an 8KB boundary; Carbink borrows these configuration parameters from TCMalloc [21, 24], which observed these parameters to reduce internal fragmentation. When an application allocates a new object, Carbink tries to round the object size up to the nearest size class and assign a free object slot from an appropriate span. If the object is bigger than the largest size class, Carbink rounds the object size up to the nearest 8KB-aligned size, and allocates a dedicated span to hold it.

To allocate spans locally, Carbink uses a *local page heap*. The page heap is an array of free lists, with each list tracking 8KB-aligned free spans of a particular size (e.g., 2MB, 4MB, etc.). If Carbink cannot find a free span big enough to satisfy an allocation request, Carbink allocates a new span, using `mmap()` to request 2MB huge pages from the OS.

Allocating and deallocating via the page heap is mutex-protected because application threads may issue concurrent allocations or deallocations. To reduce contention on the page heap, each thread reserves a private (i.e., thread-local) cache of free spans for each size class. Carbink also maintains a global cache of free lists, with each list having its own spinlock. When a thread wants to allocate a span whose size can be handled by one of Carbink's predefined size classes, the thread first tries to allocate from the thread-local cache, then the global cache, and finally the page heap. For larger allocation requests, threads allocate spans directly from the page heap.

Carbink associates each span with several pieces of metadata, including an integer that describes the span's size class, and a bitvector that indicates which object slots are free. To map a locally-resident object to its associated span metadata, Carbink uses a two-level radix tree called the *local page map*. The lookup procedure is similar to a page table walk: the first 20 bits of an object's virtual address index into the first-level radix tree table, and the next 15 bits index into a second-level table. The same mapping approach allows Carbink to map the virtual address of a locally-resident span to its metadata.

Far memory management: On a compute node, local spans contain a subset of an application's memory state. The rest of that state is stored in far spans that live in far memory regions. Recall from Figure 2b that a Carbink pointer to a non-local object embeds the object's Region ID and Span ID.

To allocate or deallocate a region, a compute node sends a request to the memory manager. A single Carbink region is 1GB or larger, since Carbink targets applications whose total memory requirements are hundreds or thousands of GBs. Upon successfully allocating a region, the compute node updates a *region table* which maps the Region ID of the allocated region to the associated far memory node.

A compute node manages far spans and far regions using additional data structures that are analogous to the ones that manage local spans. A *far page heap* handles the allocation and deallocation of far spans belonging to allocated regions. A *far page map* associates a far Span ID with metadata that (1) names the enclosing region (as a Region ID) and (2) describes the offset of the far span within that region.

Each application thread has a private far cache; Carbink also maintains a global far cache that is visible to all application threads. To swap out a local span of size s, a compute node must first use the far page heap (or a far cache if possible) to allocate a free far span of size s. Similarly, after a compute node swaps in a far span, the node deallocates the far span, returning the far span to its source (either the far page heap or a far cache).

Span filtering and swapping: The Carbink runtime executes *filtering threads* that iterate through the objects in locally-resident spans and move those objects to different local spans. Carbink's object shuffling has two goals.

- First, Carbink wants to create *hot spans* (containing only hot objects) and *cold spans* (containing only cold ones); when local memory pressure is high, Carbink's *eviction threads* prefer to swap out spansets containing cold spans. Carbink tracks object hotness using GC-style read/write barriers [4, 23]. Thus, by the time that a filtering thread examines an object, the Hotness byte in the object's pointer (see Figure 2) has already been set. Upon examining the Hotness byte, a filtering thread updates the byte using the CLOCK algorithm [12].
- Second, object shuffling allows Carbink to garbage-collect dead objects by moving live objects to new spans and then deallocating the old spans. During eviction, Carbink utilizes efficient one-sided RMA writes to swap spansets out to far memory nodes; this approach allows Carbink to avoid software-level overheads (e.g., associated with thread scheduling) on the far node.

From the application's perspective, object movement and spanset eviction are transparent. This transparency is possible because each object embeds a reverse pointer (§3.2) that allows filtering threads and evicting threads to determine which smart pointers require updating.

Carbink swaps far memory into local memory at the granularity of a span. As with swap-outs, Carbink uses one-sided RMAs for swap-ins. Swapping at the granularity of a span simplifies far memory management, since compute nodes only have to remember how spans map to memory nodes (as opposed to how the much larger number of *objects* map to memory nodes). However, swapping in at span granularity instead of object granularity has a potential disadvantage: if a compute node swaps in a span containing multiple objects, but only uses a small number of those objects, then the compute node will have wasted network bandwidth (to fetch the unneeded objects) and CPU time (to update the remotable pointers for those unneeded objects). We collectively refer to these penalties as *swap-in amplification*.

To reduce the likelihood of swap-in amplification, Carbink's filtering and eviction threads prioritize the scanning and eviction of spansets containing large objects. The associated spans contain fewer objects per span; thus, swapping in these spans will reduce the expected number of unneeded objects. Figure 3 shows that, for our production workloads, large objects occupy the majority of memory. Moreover, most hot objects are small; for example, in our company's geo-distributed database [13], roughly 95% of accesses involve objects smaller than 1.8KB. As a result, an eviction scheme which prioritizes large-object spansets is well-suited for our target applications.

In Carbink, a local span has a three-state lifecycle. A span is first *created* due to a swap-in or local allocation. The span transitions to the *filtering* state upon being examined by filtering threads. Once filtering completes, those spans transition to the *evicting* state when evicting threads begin to swap out spansets. The transition from created to filtering to evicting is fixed, and determines which Carbink runtime threads race with application threads at any given moment (§3.6).

3.4 Fault Tolerance via Erasure Coding

Erasure coding provides data redundancy with lower storage overhead than traditional replication. However, the design space for erasure coding schemes is more complex. Carbink seeks to minimize both average and long-tail access penalties for far objects; per our fault model (§3.1), Carbink also wants to efficiently recover from the failure of memory nodes. Achieving these goals forced us to make careful decisions involving coding granularity, parity recalculation, and cross-node transport protocols.

Coding granularity: To motivate Carbink's decision to erasure-code at the spanset granularity, first consider an approach that erasure-codes individual spans. In this approach, to swap out a span, a compute node breaks the span into data fragments, generates the associated parity fragments, and then writes the entire set of fragments (data+parity) to remote nodes. During the swap-in of a span, a compute node must fetch multiple fragments to reconstruct the target span.

This scheme, which we call EC-Split, is used by Hydra [29]. With EC-Split, handling the failure of memory nodes during swap-out or swap-in is straightforward: the compute node who is orchestrating the swap-out or swap-in will detect the memory node failure, select a replacement memory node, trigger span reconstruction, and then restart the swap-in or

Schemes	EC data fragment size	Network transport	Parity computation	Defragmentation
EC-Split (Hydra [29])	Span chunk	RMA in & out	Local	N/A
EC-2PC	Full span	RMA in, RPC out (+updating parity via 2PC)	Remote	N/A
EC-Batch Local (**Carbink**)	Full span	RMA in & out	Local	Remote compaction
EC-Batch Remote (**Carbink**)	Full span	RMA in & out (+parallel 2PC for compaction)	Local (swap-out)+ Remote (compaction)	Remote compaction

Table 1: The erasure-coding approaches that we study.

swap-out. The disadvantage of EC-Split is that, to reconstruct a single span, a compute node must contact multiple memory nodes to pull in all of the needed fragments. This requirement to contact multiple memory nodes makes the swap-in operation vulnerable to stragglers (and thus high tail latency[2]). This requirement also frequently prevents a compute node from offloading computation to memory nodes; unless a particular object is small, the object will span multiple fragments, meaning that no single memory node will have a complete local copy of the object.

An alternate approach is to erasure-code across a group of equal-sized spans. We call such a group a *spanset*. In this approach, each span in the spanset is treated as a fragment, with parity data computed across all of the spans in the set. To reconstruct a span, a compute node merely has to contact the single memory node which stores the span. Carbink uses this approach to minimize tail latencies.

Parity updating: Erasure-coding at the spanset granularity but swapping in at the span granularity does introduce complications involving parity updates. The reason is that swapping in a span s leaves an invalid, span-sized hole in the backing spanset; the hole must be marked as invalid because, when s is later swapped out, s will be swapped out as part of a new spanset. The hole created by swapping in s causes fragmentation in the backing spanset. Determining how to garbage-collect the hole and update the relevant parity information is non-trivial. Ideally, a scheme for garbage collection and parity updating would not incur overhead on the critical path of swap-ins or swap-outs. An ideal scheme would also allow parity recalculations to occur at either compute nodes or memory nodes, to enable opportunistic exploitation of free CPU resources on both types of nodes.

Cross-node transport protocols: In systems like RAMCloud [39], machines use RPCs to communicate. RPCs involve software-level overheads on both sides of a communication. Carbink avoids these overheads by using one-sided RMA, avoiding unnecessary thread wakeups on the receiver. However, in and of itself, RMA does not automatically solve the consistency issues that arise when offloading parity calculations to remote nodes (§3.4.2).

Throughout the paper, we compare Carbink's erasure-coding approach to various alternatives.

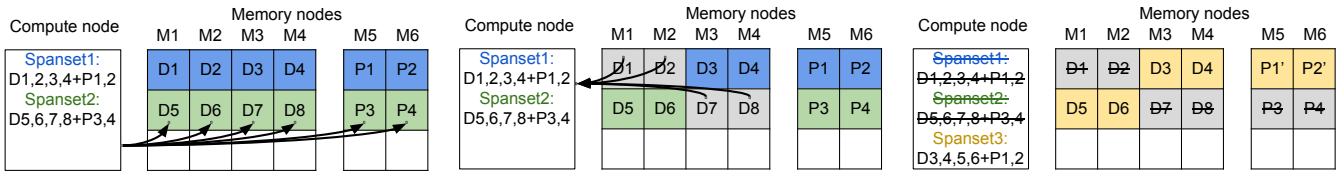
[2]Hydra [29] and EC-Cache [42] try to minimize straggler-induced latencies by contacting $k+\Delta$ memory nodes instead of the minimum k, using the first k responses to reconstruct an object. This approach increases network traffic and compute-node CPU overheads.

- **EC-Split** is Hydra's approach, which erasure-codes at the span granularity, swaps data using RMA, and synchronously recalculates parity at compute nodes when swap-outs occur. Fragmentation within an erasure-coding group never occurs, as a span is swapped in and out as a full unit.
- **EC-2PC** erasure-codes using spansets, and uses RMA to swap in at the span granularity. During a swap-out (which happens at the granularity of a span), EC-2PC writes the updated span to the backing memory node; the memory node then calculates the updates to the parity fragments, and sends the updates to the relevant memory nodes which store the parity fragments. To provide crash consistency for the update to the span and the parity fragments, EC-2PC implements a two-phase commit protocol using RPCs. There is no fragmentation within an erasure-coding group because swap-ins and swap-outs both occur at the span granularity.
- **EC-Batch Local** and **EC-Batch Remote** are the approaches used by Carbink. Both schemes erasure-code at spanset granularity, using RMA for swap-in as well as swap-out. Swap-ins occur at the granularity of a span, but swap-outs occur at the granularity of spansets (§3.4.1); thus, both EC-Batch approaches deallocate a span's backing area in far memory upon swapping that span into a compute node's local RAM. The result is that swap-ins create dead space on a remote memory node. Both EC-Batch schemes reclaim dead space and recalculate parity data using asynchronous garbage collection. EC-Batch Local always recalculates parity on compute nodes, whereas EC-Batch Remote can recalculate parity on compute nodes or memory nodes. When EC-Batch Remote offloads parity computations to remote nodes, it employs a parallel commit scheme that avoids the latencies of traditional two-phase commit (§3.4.2).

Table 1 summarizes the various schemes. We now discuss EC-Batch Local and Remote in more detail.

3.4.1 EC-Batch: Swapping

Swapping out: In both varieties of EC-Batch, a spanset contains multiple spans of the same size. At swap-out time, a compute node writes a *batch* (i.e., a spanset and its parity fragments) to a memory node. Figure 4a shows an example. In that example, the compute node has two spansets: spanset1 (consisting of data spans $< D1, D2, D3, D4 >$ and parity fragments $< P1, P2 >$), and spanset2 (containing data spans $< D5, D6, D7, D8 >$ and parity fragments $< P3, P4 >$). Carbink uses Reed-Solomon codes [43] to create parity data, and prioritizes the eviction of spansets that contain cold spans



(a) Swapping out spans and parity in a batch. **(b)** Swapping in individual spans. **(c)** Compacting spansets to reclaim space.

Figure 4: EC-Batch swapping-out, swapping-in, and far compaction.

(§3.3). Neither variant of EC-Batch updates spansets in place, so eviction may require a compute node to request additional far memory regions from the memory manager.

Swapping in: When an application tries to access an object that is currently far, the Carbink runtime inspects the application pointer and extracts the Span ID (see Figure 2b). The runtime consults the far page map (§3.3) to discover which remote node holds the span. Finally, the runtime initiates the appropriate RMA operation to swap in the span.

However, swapping in at the span granularity creates *remote fragmentation*. In Figure 4b, the compute node in the running example has pulled four spans ($D1$, $D2$, $D7$, and $D8$) into local memory. Any particular span lives exclusively in local memory or far memory; thus, the swap-ins of the four spans creates dead space on the associated remote memory nodes. If Carbink wants to fill (say) $D1$'s dead space with a new span $D9$, Carbink must update parity fragments $P1$ and $P2$. For a Reed-Solomon code, those parity fragments will depend on both $D1$ and $D9$.

There are two strawman approaches to update $P1$ and $P2$:

- The compute node can read $D1$ into local memory, generate the parity information, and then issue writes to $P1$ and $P2$.
- Alternatively, the compute node can send $D9$ to memory node $M1$, and request that $M1$ compute the new parity data and update $P1$ and $P2$.

The second approach requires a protocol like 2PC to guarantee the consistency of data fragments and parity fragments; without such a protocol, if $M1$ fails after updating $P1$, but before updating $P2$, the parity information will be out-of-sync with the data fragments.

The first approach, in which the compute node orchestrates the parity update, avoids the inconsistency challenges of the second approach. If a memory node dies in the midst of a parity update, the compute node will detect the failure, pick a new memory node to back the parity fragment, and retry the parity update. If the compute node dies in the midst of the parity update, then the memory manager will simply deallocate all regions belonging to the compute node (§3.1).

Unfortunately, both approaches require a lot of network bandwidth to fill holes in far memory. To reclaim one vacant span, the first approach requires three span-sized transfers—the compute node must read $D1$ and then write $P1$ and $P2$. The second approach requires two span-sized transfers to update $P1$ and $P2$. To reduce these network overheads, Carbink performs *remote compaction*, as described in the next section.

3.4.2 EC-Batch: Remote Compaction

Carbink employs *remote compaction* to defragment far memory using fewer network resources than the two strawmen above. On a compute node, Carbink executes several *compaction threads*. These threads look for "matched" spanset pairs; in each pair, the span positions containing dead space in one set are occupied in the other set, and vice versa. For example, the two spansets in Figure 4b are a matched pair. Once the compaction threads find a matched pair, they create a new spanset whose data consists of the live spans in the matched pair (e.g., $< D3, D4, D5, D6 >$ in Figure 4b). The compaction threads recompute and update the parity fragments $P1'$ and $P2'$ using techniques that we discuss in the next paragraph. Finally, the compaction threads deallocate the dead spaces in the matched pair (e.g., $< D1, D2, D7, D9, P3, P4 >$ in Figure 4b), resulting in a situation like the one shown in Figure 4c. Carbink's compaction can occur in the background, unlike the synchronous parity updates of EC-2PC which place consensus activity on the critical path of swap-outs.

So, how should compaction threads update parity information? Carbink uses Reed-Solomon codes over the Galois field $GF(2^8)$. The new parity data to compute in Figure 4c is therefore represented by the following equations on $GF(2^8)$:

$$P1' - P1 = A_{1,1}(D5 - D1) + A_{2,1}(D6 - D2)$$
$$P2' - P2 = A_{1,2}(D5 - D1) + A_{2,2}(D6 - D2)$$

where $A_{i,j}$ $(i \in \{0,1,2,3\}, j \in \{0,1\})$ are fixed coefficient vectors in the Reed-Solomon code. Carbink provides two approaches for updating the parity information.

- In EC-Batch Local, the compute node that triggered the swap-out orchestrates the updating of parity data. In the running example, the compute node asks $M1$ to calculate the span delta $D5 - D1$, and asks $M2$ to calculate the span delta $D6 - D2$. After retrieving those updates, the compute node determines the parity deltas (i.e., $P1' - P1$ and $P2' - P2$) and pushes those deltas to the parity nodes $M5$ and $M6$.
- In EC-Batch Remote, the compute node offloads parity recalculation and updating to memory nodes. In the running example, the compute node asks $M1$ to calculate the span delta $D5 - D1$, and $M2$ to calculate the span delta $D6 - D2$. The compute node also asks $M1$ and $M2$ to calculate partial parity updates (e.g., $A_{1,1}(D5 - D1)$ and $A_{1,2}(D5 - D1)$ on $M1$). $M1$ and $M2$ are then responsible for sending the partial parity updates to the parity nodes. For example, $M1$ sends $A_{1,1}(D5 - D1)$ to $M5$, and $A_{1,2}(D5 - D1)$ to $M6$.

In EC-Batch Local, recovery from memory node failure is orchestrated by the compute node in a straightforward way, as in EC-Split (§3.4). In EC-Batch Remote, a compute node performs remote compaction by offloading parity updates to memory nodes. The compute node ensures fault tolerance for an individual compaction via 2PC. However, the compute node aggressively issues compaction requests in parallel. Two compactions (i.e., two instance of the 2PC protocol) are safe to concurrently execute if the compactions involve different spansets; the prepare and commit phases of the two compactions can partially or fully overlap.

On a compute node, Carbink's runtime can monitor the CPU load and network utilization of remote memory nodes. The runtime can default to remote compaction via EC-Batch Local, but opportunistically switch to EC-Batch Remote if spare resources emerge on memory nodes. During a switch to a different compaction mode, Carbink allows all in-flight compactions to complete before issuing new compactions that use the new compaction mode.

The strawmen defragmentation schemes in Section 3.4.1 require two or three span-sized network transfers to recover one dead span. In the context of Figure 4, EC-Batch Local recovers four dead spans using four span-sized network transfers. EC-Batch Remote requires four span-sized network transfers (plus some small messages generated by the consistency protocol) to recover four dead spans.

3.5 Failure Recovery

Carbink handles two kinds of memory node failures: planned and unplanned. Planned failures are scheduled by the cluster manager [51, 52] to allow for software updates, disk reformatting, and so on. Unplanned failures happen unexpectedly, and are caused by phenomena like kernel panics, defective hardware, and power disruptions.

Planned failures: When the cluster manager decides to schedule a planned failure, the manager sends a warning notification to the affected memory nodes. When a memory node receives such a warning, the memory node informs the memory manager. In turn, the memory manager notifies any compute nodes that have allocated regions belonging to the soon-to-be-offline memory node. Those compute nodes stop swapping-out to the memory node, but may continue to swap-in from the node as long as the node is still alive. Meanwhile, the memory manager orchestrates the migration of regions from the soon-to-be-offline memory node to other memory nodes. When a particular region's migration has completed, the memory manager informs the relevant compute node, who then updates the local mapping from Region ID to backing memory node. At some point during this process, the memory manager may also request non-failing memory nodes to contribute additional regions to the global pool of far memory.

Unplanned Failures: On a compute node, the Carbink runtime is responsible for detecting the unplanned failure of a memory node. The runtime does so via connection timeouts or more sophisticated leasing protocols [15, 16]. Upon detecting an unplanned failure, the runtime spawns background threads to reconstruct the affected spans using erasure coding. The runtime is also responsible for allowing application threads to read spans whose recovery is in-flight.

Span reconstruction: To reconstruct the spans belonging to a failed memory node M_{fail}, a compute node first requests a new region from the memory manager. Suppose that the new region is provided by memory node M_{new}. The compute node iterates through each lost spanset associated with M_{fail}; for each spanset, the compute node tells M_{new} which external spans and parity fragments to read in order to erasure-code-restore M_{fail}'s data. As the relevant spans are restored, a compute node can still swap in and remotely compact those spans. However, the swap-in and remote compaction activity will have to synchronize with recovery activity (§3.6).

In EC-Batch Local, when a compute node detects a memory node failure, the compute node cancels all in-flight compactions involving that node. A compute node using EC-Batch Remote does the same; however, for each canceled compaction, the compute node must also instruct the surviving memory nodes in the 2PC group to cancel the transaction.

The data and parity for a swapped-out spanset reside on multiple memory nodes. As a compute node recovers from the failure of one of the nodes in that group, another node in the group may fail. As long as the number of failed nodes does not exceed the number of parity nodes, Carbink can recover the spanset. The reason is that all of the information needed to recover is stored on a compute node, e.g., in the far page heap (§3.3). Due to space limitations, we omit a detailed explanation of how Carbink deals with concurrent failures.

Degraded reads: During the reconstruction of an affected span, application threads may try to swap in the span. The runtime handles such a fetch using a *degraded read* protocol. For example, consider Figure 4a. Suppose that $M1$ fails unexpectedly, and while the Carbink runtime is recovering $M1$'s spans ($D1$ and $D5$), an application thread tries to read an object residing in $D1$. The runtime will swap in data spans $D2$, $D3$, and $D4$, as well as parity fragment $P1$, and then reconstruct $D1$ via erasure coding. Degraded reads ensure that the failure of a memory node merely slows down an application instead of blocking it. In Section 5.3, we show that application performance only drops for 0.6 seconds, and only suffers a throughput degradation of 36% during that time.

Network bandwidth consumption: During failure recovery, Carbink consumes the same amount of network bandwidth as Hydra. For example, suppose that both Hydra and Carbink use RS4.2 encoding and have 4 spans, with a span stored on each of 4 memory nodes. In Hydra, a single node failure will lose four 1/4th spans. Reconstructing each 1/4th span will require the reading of four 1/4th span/parity regions from the surviving nodes, resulting in an aggregate network bandwidth requirement of 1 full span. So, reconstructing four 1/4th spans

will require an aggregate network bandwidth of 4 full spans. In Carbink, the failure of a single memory node results in the loss of 1 full span. To recover that span, Carbink (like Hydra) must read 4 span/parity regions.

3.6 Thread Synchronization

On a compute node, the main kinds of Carbink threads are applications threads (which read objects, write objects, and swap in spans), filtering threads (which move objects within local spans), and eviction threads (which reclaim space by swapping local spansets to far memory). At any given time, a span may be in one of two concurrency regimes (§3.3): the span is either accessible to application threads and filtering threads, or to application threads and eviction threads. In both regimes, Carbink has to synchronize how the relevant threads update Carbink's smart pointers (§3.2).

At a high level, Carbink uses an RCU locking scheme that is somewhat reminiscent of AIFM's approach [44]. Due to space restrictions, we merely sketch the design. Carbink optimizes for the common case in which a span is only being accessed by an application thread. In this common case, an application thread grabs an RCU read lock on the pointer via the pointer's `Deref()` method, as shown in Listing 1. The thread sees that either (1) the **P**resent bit is not set, in which case the Carbink runtime issues an RMA read to swap in the appropriate span; (2) alternatively, the thread sees that the **P**resent bit is set, but the **M** and **E** bits are unset. In the second case, `Deref()` can just return a normal pointer back to the application. The application can be confident that concurrent filtering or evicting threads will not move or evict the object, because those threads cannot touch the object until application-level threads have released their RCU read locks via the `DerefScope` destructor (Listing 1).

The more complicated scenarios arise when the **P**resent bit is set and either the **M** or **E** bit are set as well. In this case, the (say) **M** bit has been set because the filtering thread set the bit and then called `SyncRCU()` (i.e., the RCU write waiting lock). The concurrent application thread and filtering thread essentially race to acquire the pointer's spinlock; if the application thread (i.e., `Deref()`) wins, it makes a copy of the object, clears **M**, releases the spinlock, and returns the address of the object copy to the application. Otherwise, if the filtering thread wins, it moves the object, clears **M**, and releases the spinlock. The losing thread has to retry the desired action. An analogous situation occurs if the **E** bit is set.

Carbink's eviction and remote compaction threads directly poll the network stack to learn about RMA completions and RPC completions. An application thread which has issued an RMA swap-in operation will yield, but a dedicated RMA poller thread detects when application RMAs have completed and awakens the relevant application threads. Polling avoids the overheads of context switching to new threads and notifying old threads that network events have occurred.

During recovery (§3.5), Carbink spawns additional threads to orchestrate the reconstruction of spans. Those threads acquire per-spanset mutexes which are also acquired by threads performing swap-ins, swap-outs, and remote compactions.

4 Implementation

Our Carbink prototype contains 14.3K lines of C++. It runs atop unmodified OSes, using standard POSIX abstractions for kernel-visible threads and synchronization. The runtime leverages the PonyExpress user-space network stack [35]. On a compute node, all threads in a particular application (both application-defined threads and Carbink-defined threads) execute in the same process. On a memory node, a Carbink daemon exposes far memory via RMAs or RPCs. We use Intel ISA-L v2.30.0 [25] for Reed-Solomon erasure coding.

Our current prototype has a simplified memory manager that is unreplicated, does not handle planned failures, and statically assigns memory nodes to compute nodes. Implementing the full version of the memory manager will be conceptually straightforward, since we can use off-the-shelf libraries for replicated state machines [1, 45] and cluster management [51, 52]. We also note that the experiments in §5 are insensitive to the performance of the memory manager, regardless of whether the manager is replicated or not. The reason is that memory allocations and deallocations (which must be routed through the memory manager) are rare and are not on the critical path of steady-state compute node operations like swap-in and swap-out.

To better understand the performance overheads of Carbink's erasure-coding approach, we built an AIFM-like [44] far memory system. That system uses remotable pointers like Carbink, but swaps in and out at the granularity of objects, and provides no fault tolerance. Like Carbink, it leverages the PonyExpress [35] user-space network stack. Our AIFM clone is 5.8K lines of C++.

5 Evaluation

In this section, we answer the following questions:

1. What is the latency, throughput, and remote memory usage of EC-Batch compared with the other fault tolerance schemes (§5.1 and §5.2)?
2. How does an unplanned memory node failure impact the performance of Carbink applications (§5.3)?
3. How does the performance of Carbink's span-based memory organization compare to the performance of an AIFM-like object-level approach (§5.4)?

Testbed setup: We deployed eight machines in the same rack, including one compute node and seven memory nodes; one of the memory nodes was used for failover. Each machine was equipped with dual-socket 2.2 GHz Intel Broadwell processors and a 50 Gbps NIC.

Fault tolerance schemes: Using the Carbink runtime, we compared our proposed EC-Batch schemes to four ap-

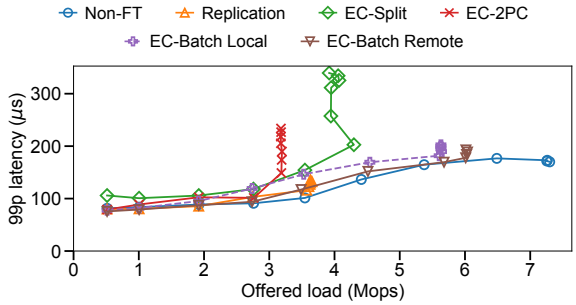


Figure 5: Microbenchmark load-latency curves.

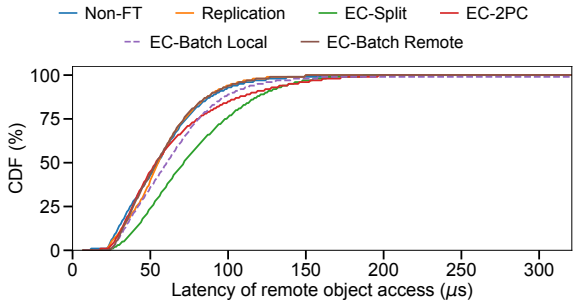


Figure 6: Latency distribution of remote object accesses in the microbenchmark under an offered load of 2 Mops.

proaches: Non-FT (a non-fault-tolerant scheme that used RMA to swap spans), Replication (which replicated spans on multiple nodes), EC-Split (the approach used by Hydra [29]), and EC-2PC (Table 1). We configured all fault tolerance schemes to tolerate up to two memory node failures. So, the Replication scheme replicated each swapped-out span on three memory nodes, whereas the EC schemes used six memory nodes—four held data, and two held RS4.2 parity bits [43]. EC-Batch spawned two compaction threads by default.

As mentioned in Section 4, we also built an AIFM-like far memory system. This system did not provide fault tolerance, but it provided a useful comparison with our Non-FT Carbink version.

Carbink borrows the span sizes that are used by TCMalloc (§3.3). These parameters have been empirically observed to reduce internal fragmentation. In our evaluation, EC-Batch (both Local and Remote) grouped four equal-size spans into a spanset, swapping out at the granularity of a spanset. Increasing spanset sizes would allow Carbink to issue larger batched RMAs, improving network efficiency. However, spansets whose evictions are in progress must be locked in local memory while RMAs complete; thus, larger spanset sizes would delay the reclamation of larger portions of local memory.

5.1 Microbenchmarks

To get a preliminary idea of Carbink's performance, we created a synthetic benchmark that wrote 15 million 1 KB objects (totalling 15 GB) to a remotable array. The compute node's local memory had space to store 7.5 GB of objects (i.e., half of the total set). By default, the compute node spawned 128

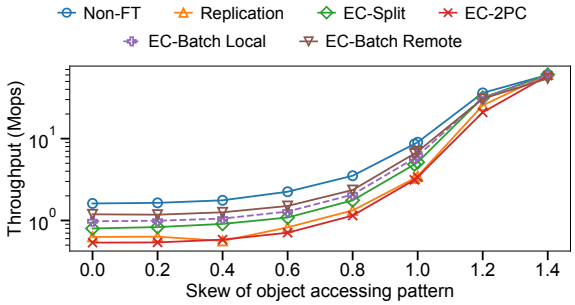


Figure 7: Impact of skew on throughput.

threads on 32 logical cores to access objects; the access pattern had a Zipfian-distributed [41] skew of 0.99. Such skews are common in real workloads for key/value stores [7].

Object access throughput and tail latency: Figure 5 shows the 99th-percentile latency with various object access loads. All of the fault-tolerant schemes eventually hit a "hockey stick" in tail latency growth when the schemes could no longer catch up with the offered load. EC-Batch Remote had the highest sustained throughput (6.0 Mops), which was 40% higher than the throughput of the state-of-the-art EC-Split (4.3 Mops). EC-Batch Local achieved 5.6 Mops, which was 30% higher than EC-Split. EC-Split had worse performance because it had to issue four RMA requests to swap in one span; thus, EC-Split quickly became bottlenecked by network IO. In contrast, EC-Batch only issued one RMA request per swap-in.

EC-Batch Remote had 18%-29% lower tail latency than EC-Split under the same load (before reaching the "hockey-stick"). The reason was that EC-Split's larger number of RMAs per swap-in left EC-Split more vulnerable to stragglers [29]. Also recall that EC-Batch can support computation offloading [3, 27, 44, 57], which is hard with EC-Split (§3.4).

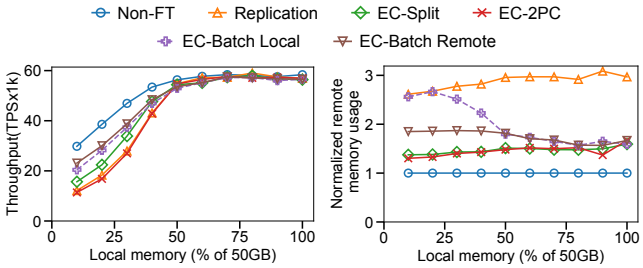
EC-2PC had the worst throughput because it relied on costly RPCs and 2PC protocols to swap out spans. Thus, EC-2PC could not reclaim local memory as fast as other schemes. The Replication scheme was bottlenecked by network bandwidth, since every swap-out incurred a 3× network write penalty; in contrast, EC-based schemes used RS4.2 erasure coding to reduce the write penalty to 1.5×.

Latency distribution of remote object accesses: Figure 6 shows the latency of accessing remote objects under 2 Mops of offered load. With this low offered load, Replication and EC-Batch Remote achieved similar access latencies as Non-FT because none of the schemes were bottlenecked by network bandwidth. EC-Batch Local had slightly higher remote access latencies. However, EC-Split had significantly higher access latencies (e.g., at the median and tail) than EC-Batch Local and Remote; the reason was that EC-Split issued four times as many network IOs and thus was more sensitive to stragglers. EC-2PC's tail latency was slightly higher than that of EC-Batch Local and Remote due to the overhead of costly RPCs and 2PC traffic.

Impact of skewness: Figure 7 shows how the skewness of object accesses impacted throughput. EC-Batch Remote and

	# Compaction threads	Norm. remote mem usage	Avg. # remote logical cores	Avg. BW (Gbps)
EC-Batch Local	1	2.54	0.23	1.27
	2	2.35	0.53	1.64
	3	2.28	0.56	1.76
EC-Batch Remote	1	1.89	1.97	2.98
	2	1.83	2.10	3.15
	3	1.74	2.27	3.40
W/o compaction	0	3.03	–	–

Table 2: Remote resource usage in the microbenchmark. The remote memory usage is normalized with respect to the usage of Non-FT. The number of remote logical cores and the network bandwidth are averaged across all six memory nodes.



(a) Transaction throughput. **(b)** Remote memory usage.

Figure 8: Transactional KV-store evaluation.

Local performed best due to their more efficient swapping approaches. However, the throughput of all schemes increased with higher skewness. The reason is that high skewness led to a smaller working set and thus a higher likelihood that hot objects were locally resident. In these scenarios, schemes with faster swapping were not rewarded as much.

Remote resource usage with compaction: Table 2 shows the impact of compaction on the average memory, CPU, and bandwidth usage per memory node. Without compaction, EC-Batch used 3.03× remote memory (normalized with respect to Non-FT memory consumption). With two local compaction threads, EC-Batch Remote's memory overhead reduced to 1.83×. The memory reduction was at the expense of 2.1 cores and 3.15 Gbps bandwidth on each memory node. With more compaction threads, Carbink could further reduce memory usage at the cost of higher CPU and bandwidth utilization. That being said, we note that the synthetic microbenchmark application represented an extreme case of remote CPU and network usage, since the workload accessed objects without actually computing on them.

EC-Batch Remote vs. Local: EC-Batch Remote had higher throughput and lower tail latency than EC-Batch Local (Figure 5). This was because EC-Batch Local's compaction required (1) local CPUs for parity computation and (2) network bandwidth for transferring span deltas and parity updates, leaving fewer local resources for application threads and RMA reads. Because of EC-Batch Remote's faster compaction, EC-Batch Remote also used 28%-34% less remote memory than EC-Batch Local (Table 2). However, EC-Batch Remote consumed more remote CPUs (2.10 vs. 0.53 cores) and more network bandwidth (3.15 vs. 1.64 Gbps) than Local. In practice, the Carbink runtime could transparently switch between EC-Batch Remote and Local based on an application developer's policy about resource/performance trade-offs.

5.2 Macrobenchmarks

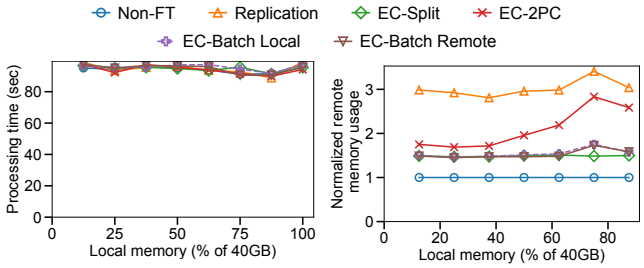
We evaluated Carbink using two memory-intensive applications that would benefit from remote memory: an in-memory transactional key-value store, and a graph processing algorithm. The two applications exhibited different patterns of object accesses, and had different working set behaviors.

Transactional KV-store: This application implemented a transactional in-memory B-tree, exposing it via a key/value interface similar to that of MongoDB [37]. Each remotable object was a 4 KB value stored in a B-tree leaf. The application spawned 128 threads, and each thread processed 20 K transactions. The compute node provisioned 32 logical cores, with the application overlapping execution of the threads for higher throughput [26, 38, 44, 56]. Each transaction contained three reads and three writes, similar to the TPC-A benchmark [53]. Each update created a new version of a particular key's value; asynchronously, the application trimmed old versions. The maximum working set size during the experiment was roughly 50 GB.

Throughput: Figure 8a shows the KV-store throughput when varying the size of local memory (normalized as a fraction of the maximum working set size). In scenarios with less than 50% local memory, EC-Batch Remote achieved higher transactions per second (TPS) than all other fault tolerance schemes. For example, TPS for EC-Batch Remote was 1.5%-48% higher than that of EC-Split; this was because EC-Batch only needed one RMA request to swap in a span. EC-Batch Remote was at most 29% slower than Non-FT, mainly due to the additional parity update required for fault tolerance. EC-Batch Local was at most 13% slower than EC-Batch Remote. EC-2PC performed the worst among EC schemes.

All schemes achieved similar throughput when the local memory size was above 50%. The reason was that the *average* working set size of the workload was only half the size of the *maximum* memory usage. The maximum memory usage only occurred when the B-Tree had fallen very behind in culling old versions of objects.

Remote memory usage: Figure 8b plots remote memory usage as a function of local memory sizes; remote memory usage is normalized with respect to that of Non-FT. Compared to EC-Split, EC-Batch Remote and Local used up to 35% and 93% more remote memory, respectively. EC-Batch schemes defragmented remote memory using compaction, but when local memory space was less than 50%, remote compaction could not immediately defragment the spanset holes created by frequent span swap-ins. As local memory grew larger, span fetching became less frequent, making it



(a) Processing time. **(b)** Remote memory usage.

Figure 9: Graph processing evaluation.

easier for remote compaction to reclaim space. In this less hectic environment, EC-Batch's remote memory usage was similar to that of the other erasure-coding schemes.[3]

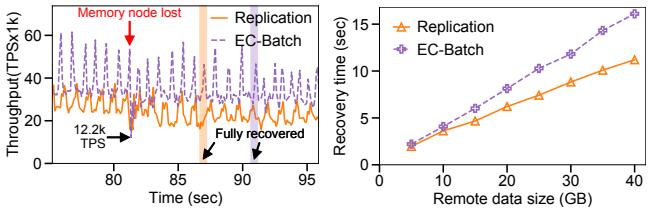
Graph processing: We implemented a connected-components algorithm [50] that found all sets of linked vertices in a graph. This kind of algorithm is critical to various Google services. We evaluated the algorithm using the Friendster graph [30] which contained 65 million vertexes and 1.8 billion edges. In the graph analysis code, each vertex's adjacency list was referenced via remotable pointers. The total size of the objects stored in Carbink was roughly 40 GB. The application used 80 application threads that ran atop 80 logical cores. In our experimental results, the reported processing times exclude graph loading, since graph loading is dominated by disk latencies.

Figure 9a shows that all schemes had similar processing times as Non-FT, regardless of the local memory size. The reason was that the graph application had a high compute-to-network ratio—the application fetched all neighbors associated with each vertex and then spent non-trivial time enumerating each neighbor and computing on them. As a result of this good spatial locality and high "think time," the graph application did not incur frequent data swapping, and thus avoided fault tolerance overhead that the KV-store could not.

Figure 9b shows that EC-Batch Local and Remote had similar remote memory usage as EC-Split: 15%-39% lower than EC-2PC and roughly 50% lower than Replication. All EC-based schemes had lower remote memory overheads than Replication because the erasure coding only incurred a 1.5× space overhead for the extra parity data.

EC-2PC used more memory than EC-Batch because the graph workload randomly fetched diverse-sized spans. The random fetch sizes reflected the fact that different vertices had different sizes for their adjacency lists. This lack of span size locality hindered dead space reclamation, since EC-2PC had to wait longer for all of the spans in an erasure-coding group to be swapped in. EC-Batch avoided this problem by bundling equal-sized spans into the same spanset and using remote compaction.

[3]The remote memory usage of triple-replication was slightly less than 3× the usage of Non-FT because Non-FT could swap out memory faster during periods of high local memory pressure.



(a) KV-store TPS over time. **(b)** Microbenchmark recovery.

Figure 10: Failure recovery evaluation.

5.3 Failure Recovery

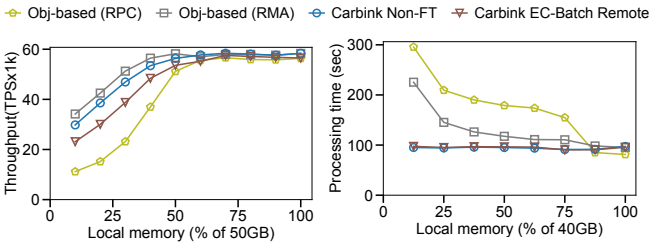
We measured the recovery time for an unplanned memory node failure in the KV-store, the graph processor, and the microbenchmark application. For the graph application, all schemes achieved similar processing time during unplanned failures; thus, in the text below, we focus on the KV-store and the microbenchmark.

Transactional KV-store: Figure 10a shows the KV-store throughput of Replication and EC-Batch Local, with a data point collected every 100 ms before and after an unplanned memory node failure. Upon detecting the failure, EC-Batch Local immediately reconstructed the lost data on a pre-configured failover memory node. We gave the KV-store 15 GB of local memory, equivalent to 30% of the 50 GB maximum working set size.

The throughput of both schemes fluctuated sinusoidally because the KV-store frequently tried to swap in remote objects, but the swap-ins sometimes had to synchronously block until eviction threads could reclaim enough local memory. After a memory node failed, EC-Batch needed 0.6 seconds to restore normal throughput, while replication needed 0.3 seconds. This is because, during failure recovery, an EC-Batch read that targeted an affected span used the degraded read protocol which uses more bandwidth than a normal read (§3.5); in contrast, a Replication read that targeted an affected span consumed the same amount of bandwidth as a read during non-failure-recovery. During recovery, the throughput of Replication and EC-Batch dropped an average of 35% and 36% respectively.

EC-Batch required 9.7 seconds to fully regenerate the lost data on the failover node, taking 1.7× longer than Replication. This difference arose because, in EC-Batch, the new memory node read 4× span/parity information involving the lost data and computed erasure codes to reconstruct the lost data. In contrast, Replication lost more data per memory node, but only read one copy of the lost data. Note that with EC-Batch, degraded reads mostly happened during the first second of failure recovery; the skewed workload meant that a small number of objects were the targets of most reads, and once a hot object was pulled into local memory (perhaps by a degraded read), the object would not generate additional degraded reads.

Microbenchmark: Figure 10b shows recovery times as a function of the remote data size. The recovery time of EC-Batch increased almost linearly with the remote data size,



(a) KV-store application. **(b)** Graph application.

Figure 11: Application performance: AIFM-like object-based systems and Carbink.

with 0.6 GB/s recovery speed. This speed was 12%-44% slower than Replication due to the larger amount of recovery information that EC-Batch had to transfer around the network, and the computational overhead of generating erasure codes.

Prior work [10, 29, 58] also found that, during recovery, erasure-coding schemes had longer recovery times and worse performance degradation than replication schemes. However, this drawback only happens for unplanned failures which, in our production environment, are rare compared to planned failures; in an erasure-coding scheme, handling a *planned* failure just requires simple copying of the information on a departing memory node, and does not incur additional work to find parity information or recompute erasure coding. Thus, in our deployment setting where unplanned failures are rare, erasure-coding schemes (which have lower memory utilization than replication schemes) are very attractive.

5.4 Comparison with AIFM-like Systems

We compared span-based swapping in Carbink with the object-based approach used in AIFM [44]. We implemented two AIFM-like systems using our threading and network stack (§4). The first system used RPCs to swap individual objects, with the remote memory nodes tracking the object-to-remote-location mapping (as done in AIFM). Our second object-granularity swapping system used more-efficient RMAs to swap objects, and had compute nodes track the mapping between objects and their remote locations; recall that RMA is one-sided, so compute nodes could not rely on memory nodes to synchronously update mappings during swaps. Like the original AIFM, neither system provided fault tolerance.

Transactional KV-store: Figure 11a shows that, if local memory was too small to hold the average working set, Non-FT Carbink had 45%-167% higher throughput than the AIFM-like system with RPC. The reason is that, when local memory pressure was high, more swapping occurred, and the better efficiency of RMAs over RPCs became important. However, Non-FT Carbink achieved 5.6%-15% lower throughput than the object-based system with RMA. This was due to swap-in amplification. For example, Non-FT Carbink might swap in an 8KB span but only use one 4KB object in the span; this never happens in a system that swaps at an object granularity.

Graph processing: Figure 11b shows the graph application's processing time. When the local memory size was below 87.5%, Carbink performed 18%-58% faster than the object-based system with RMA. This is because, in the graph workload, 4% of large objects occupied 50% of the overall data set. Carbink prioritized swapping out large cold objects (§3.3), keeping most small objects in local memory and reducing the miss rate for those objects. In contrast, the object-based systems did not consider object sizes when swapping, leading to an increased miss rate for small objects. Note that, with larger local memories, all schemes had similar performance; indeed, when all objects fit into local memory, the object-based system with RPC slightly outperformed the rest because it did not require a dedicated core to poll for RMA completions.

6 Discussion

EC-Batch for paging-based systems: Carbink uses EC-Batch to transparently expose far memory via remotable pointers. However, EC-Batch can also be used to expose far memory via OS paging mechanisms [5, 22, 46]. In a traditional paging-based approach for far memory, a compute node swaps in and out at the granularity of a page. However, a compute node can use EC-Batch to treat each page as a span, such that pages are swapped out at the "pageset" granularity, and pages are swapped in at the page granularity.

Custom one-sided operations: EC-Batch requires memory nodes to calculate span deltas and parity updates (§3.4.2). In our Carbink prototype, memory nodes use separate threads to execute these calculations. However, memory nodes could instead implement them as custom one-sided operations in the network stack, such that the network stack itself performs the calculations, avoiding the need to context-switch to external threads. This approach has been used in prior work [6, 9, 35, 47, 48] to avoid thread scheduling overheads.

Designing the memory manager: We used a centralized manager because such a manager (1) simplified our overall design, and (2) made it easier to drive memory utilization high (because a centralized manager will have a global, accurate view of memory allocation metadata). A similarly-centralized memory manager is used by the distributed transaction system FaRM [16]. If the centralized manager became unavailable, Carbink could fall back to a decentralized memory allocation scheme like the one used by Hydra [29] or INFINISWAP [22].

The state maintained by the memory manager is not large. With 1 GB regions, we expect up to 500 regions in a typical memory node (similar to FaRM [16]). With thousands of memory nodes, the memory manager just needs to store a few MBs of state for region assignments.

Fault tolerance for compute nodes: In Carbink, a compute node does not share memory with other compute nodes. Thus, a Carbink application can checkpoint its own state without fear of racing with other compute nodes that modify the state being checkpointed. Checkpoint data could be placed in a

	Fast s/o	Low mem	Fast s/i	Interface	Coding granularity
On-disk rpl.	✗	✓	✓	Various	–
In-memory rpl.	✓	✗	✓	Various	–
Hydra [29]	✓	✓	✗	Paging	Split 4KB pages
Cocytus [10]	✓	✓	✗	KV-store	Across 4KB pages
BCStore [31]	✓	✓	✗	KV-store	Across objs
Hybrid [32]	✗	✗	✓	KV-store	Split 4KB pages
Carbink	✓	✓	✓	Remotable pointers	Across spans

Table 3: Comparison of existing fault-tolerant approaches for far memory. "Fast s/o" indicates whether a system can swap out at network/memory speeds. "Low mem" means that a system has relatively low memory pressure. "Fast s/i" refers to whether a system can swap in at network/memory speeds.

non-Carbink store, obviating the need to track how checkpointed spans move across Carbink memory nodes during compaction and invalidation. Alternatively, Carbink itself could store checkpoints, e.g., in the fault-tolerant address space of a well-known Carbink application whose sole purpose is to store checkpoints.

7 Related Work

Fault tolerance for far memory: Many far memory systems do not provide fault tolerance [2, 44, 55]. Of the systems that do, most replicate swapped-out data to local disks or remote ones [5, 22, 46]. Unfortunately, this approach forces application performance to bottleneck on disk bandwidth or disk IOPs during bursty workloads or failure recovery [29]. This behavior is unattractive, since a primary goal of a far memory system is to have applications run at *memory* speeds as much as possible.

Like Carbink, Hydra [29] is a far memory system that provides fault tolerance by writing erasure-coded local memory data to far RAM. Hydra uses the EC-Split coding approach that we describe in Section 3.4. As we demonstrate in Section 5, Carbink's erasure-coding scheme provides better application performance in exchange for somewhat higher memory consumption. Carbink's coding scheme also enables the offloading of computations to far memory nodes. Such offloading can significantly improve the performance of various applications [3, 27, 44, 57].

Fault tolerance for in-memory transactions and KV-stores: In-memory transaction systems typically provide fault tolerance by replicating data across the memory of multiple nodes [15, 16, 26]. These approaches suffer from the classic disadvantages of replication: double or triple storage overhead, and the associated increase in network traffic.

Recent in-memory KV-stores use erasure coding to provide fault tolerance. For example, Cocytus [10] and BCStore [31] only rely on in-memory replication to store small instances of metadata; object data is erasure-coded using a default page size of 4KB. Cocytus erasure-codes using a scheme that resembles EC-2PC (§3.4). To reduce the network utilization of a Cocytus-style approach, a BCStore compute node buffers outgoing writes; this approach allows the node to batch the computation of parity fragments (and thus issue fewer updates to remote data and parity regions). Batching reduces network overhead at the cost of increasing write latency.

Both Cocytus and BCStore rely on two-sided RPCs to manipulate far memory. RPCs incur software-level overheads involving thread scheduling and context switching on remote nodes. To avoid these costs, Carbink eschews RPCs for one-side RMA operations. Carbink also issues fewer parity updates than Cocytus; whereas Cocytus uses expensive 2PC to update parity information during every write, Carbink defers parity updates until compaction occurs on remote nodes (§3.4.2). Carbink's compaction approach is also more efficient than that of BCStore. BCStore's compaction algorithm performs actual copying of data objects on memory nodes, whereas Carbink compaction just manipulates span pointers inside of spanset metadata.

A far memory system could use both replication and erasure coding [32]. For example, during a Hydra-style swap-out, a span would be erasure-coded and the fragments written to memory nodes; however, a full replica of the span would also be written out. Relative to Carbink, this hybrid approach would have lower reconstruction costs (assuming that the full replica did not live on the failed node). However, Carbink would have lower memory overheads because no full replica of a span would be stored. Carbink would also have faster swap-outs, because swap-outs in the hybrid scheme would require an EC-2PC-like mechanism to ensure consistency.

Table 3 summarizes the strengths and weaknesses of the various systems discussed above.

Memory compaction: In Carbink, the far memory regions used by a program become fragmented as spans are swapped in. Memory compaction is a well-studied topic in the literature about "moving" garbage collectors for managed languages (e.g., [11, 18, 49]). Moving garbage collection is also possible for C/C++ programs; Mesh [40] represents the state-of-the-art. With respect to this prior work, Carbink's unique challenge is that the compaction algorithm (§3.4.2) must compose well with an erasure coding scheme that governs how objects move between local memory and far memory.

8 Conclusion

Carbink is a far memory system that provides low-latency, low-overhead fault tolerance. Carbink erasure-codes data using a span-centric approach that does not expose swap-in operations to stragglers. Whenever possible, Carbink uses efficient one-sided RMAs to exchange data between compute nodes and memory nodes. Carbink also uses novel compaction techniques to asynchronously defragment far memory. Compared to Hydra, a state-of-the-art fault-tolerant system for far memory, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher memory usage.

Acknowledgments

We thank our shepherd Luís Rodrigues and the anonymous reviewers for their insightful comments. We also thank Kim Keeton and Jeff Mogul for their comments on early drafts of the paper, and Maria Mickens for her comments on a later draft. Yang Zhou and Minlan Yu were supported in part by NSF CNS-1955422 and CNS-1955487.

References

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 106–118, 2020.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, and et al. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of USENIX ATC*, pages 775–787, 2018.

[3] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of ACM HotOS*, pages 120–126, 2019.

[4] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-Rationing Garbage Collection for Hybrid Memories. *ACM SIGPLAN Notices*, 53(4):62–77, 2018.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of ACM EuroSys*, pages 1–16, 2020.

[6] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote Memory Calls. In *Proceedings of ACM HotNets*, pages 38–44, 2020.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):53–64, 2012.

[8] Cristina Băsescu and Bryan Ford. Immunizing Systems from Distant Failures by Limiting Lamport Exposure. In *Proceedings of ACM HotNets*, pages 199–205, 2021.

[9] Matthew Burke, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R.K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of USENIX SOSP*, pages 228–242, 2021.

[10] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.

[11] Jon Coppeard. Compacting Garbage Collection in SpiderMonkey. https://hacks.mozilla.org/2015/07/compacting-garbage-collection-in-spidermonkey/, 2015.

[12] Fernando J. Corbato. A Paging Experiment with the Multics System. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of USENIX NSDI*, pages 401–414, 2014.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[17] Jason Evans. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of BSDCan Conference*, 2006.

[18] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, 1969.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of USENIX OSDI*, pages 17–30, 2012.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of USENIX OSDI*, pages 599–613, 2014.

[21] Google. TCMalloc Open Source. https://github.com/google/tcmalloc.

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of USENIX NSDI*, pages 649–667, 2017.

[23] Xianglong Huang, Stephen M Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. *ACM SIGPLAN Notices*, 39(10):69–80, 2004.

[24] Andrew Hamilton Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond Malloc Efficiency to Fleet Efficiency: A Hugepage-Aware Memory Allocator. In *Proceedings of USENIX OSDI*, pages 257–273, 2021.

[25] Intel. Intel Intelligent Storage Acceleration Library. https://github.com/intel/isa-l.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[27] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Proceedings of Conference on Innovative Data Systems Research*, 2022.

[28] Jakub Łącki, Vahab Mirrokni, and Michał Włodarczyk. Connected Components at Scale via Local Contractions. *arXiv preprint arXiv:1807.10727*, 2018.

[29] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *arXiv preprint arXiv:1910.09727*, 2019.

[30] Jure Leskovec. Friendster Social Network Dataset. https://snap.stanford.edu/data/com-Friendster.html.

[31] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. BCStore: Bandwidth-Efficient In-Memory KV-store with Batch Coding. *Proceedings of IEEE International Conference on Massive Storage Systems and Technology*, 2017.

[32] Yuzhe Li, Jiang Zhou, Weiping Wang, and Yong Chen. RE-Store: Reliable and Efficient KV-Store with Erasure Coding and Replication. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 1–12, 2019.

[33] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. Graphlab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041*, 2014.

[34] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Proceedings of IEEE International Conference on Big Data*, pages 2884–2892, 2017.

[35] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[36] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[37] MongoDB Inc. MongoDB Open Source. https://github.com/mongodb/mongo.

[38] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[39] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, and et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[40] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of ACM PLDI*, pages 333–346, 2019.

[41] David M.W. Powers. Applications and Explanations of Zipf's Law. In *Proceedings of New Methods in Language Processing and Computational Natural Language Learning*, 1998.

[42] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of USENIX OSDI*, pages 401–417, 2016.

[43] Irving S. Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of USENIX OSDI*, pages 315–332, 2020.

[45] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of USENIX OSDI*, pages 69–87, 2018.

[47] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of ACM EuroSys*, pages 1–16, 2020.

[48] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M.K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of ACM SIGCOMM*, pages 93–105, 2021.

[49] SUN Microystems. Memory Management in the Java HotSpot Virtual Machine, 2006.

[50] Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, pages 12–21, 2018.

[51] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, and et al. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of USENIX OSDI*, pages 787–803, 2020.

[52] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of ACM EuroSys*, pages 1–14, 2020.

[53] Transaction Processing Performance Council (TPC). TPC-A. http://tpc.org/tpca/default5.asp.

[54] Volt Active Data. VoltDB. https://www.voltdb.com/.

[55] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *Proceedings of USENIX OSDI*, pages 261–280, 2020.

[56] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid Is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[57] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of USENIX NSDI*, pages 633–651, 2021.

[58] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does Erasure Coding Have a Role to Play in My Data Center? *Microsoft Research Technical Report*, 2010.