Hydra : Resilient and Highly Available Remote Memory

Youngmoon Lee*[1], Hasan Al Maruf*[2], Mosharaf Chowdhury[2], Asaf Cidon[3], and Kang G. Shin[2]

[1]Hanyang University, [2]University of Michigan, [3]Columbia University

Abstract

We present Hydra, a low-latency, low-overhead, and highly available resilience mechanism for remote memory. Hydra can access erasure-coded remote memory within a single-digit μs read/write latency, significantly improving the performance-efficiency tradeoff over the state-of-the-art – it performs similar to in-memory replication with 1.6× lower memory overhead. We also propose CodingSets, a novel coding group placement algorithm for erasure-coded data, that provides load balancing while reducing the probability of data loss under correlated failures by an order of magnitude. With Hydra, even when only 50% memory is local, unmodified memory-intensive applications achieve performance close to that of the fully in-memory case in the presence of remote failures and outperforms the state-of-the-art remote-memory solutions by up to 4.35×.

1 Introduction

Modern datacenters are embracing a paradigm shift toward disaggregation, where each resource is decoupled and connected through a high-speed network fabric [4, 9, 13, 35–37, 58, 61, 62, 81]. In such disaggregated datacenters, each server node is specialized for specific purposes – some are specialized for computing, while others for memory, storage, and so on. Memory, being the prime resource for high-performance services, is becoming an attractive target for disaggregation [18, 19, 22, 32, 39, 47, 50, 58, 61].

Recent remote-memory frameworks allow an unmodified application to access remote memory in an implicit manner via well-known abstractions such as distributed virtual file system (VFS) and distributed virtual memory manager (VMM) [18, 47, 50, 58, 65, 81, 87]. With the advent of RDMA, remote-memory solutions are now close to meeting the single-digit μs latency required to support acceptable application-level performance [47, 58]. However, realizing remote memory for heterogeneous workloads running in a large-scale cluster faces considerable challenges [19, 24] stemming from two root causes:

1. *Expanded failure domains:* As applications rely on memory across multiple machines in a remote-memory cluster, they become susceptible to a wide variety of failure scenarios. Potential failures include independent and correlated failures of remote machines, evictions from and corruptions of remote memory, and network partitions.
2. *Tail at scale:* Applications also suffer from stragglers or late-arriving remote responses. Stragglers can arise from many sources including latency variabilities in a large network due to congestion and background traffic [41].

While one leads to catastrophic failures and the other manifests as service-level objective (SLO) violations, both are unacceptable in production [58, 68]. Existing solutions take three primary approaches to address them: (i) local disk backup [50, 81], (ii) remote in-memory replication [30, 42, 46, 64], and (iii) remote in-memory erasure coding [76, 80, 84, 86] and compression [58]. Unfortunately, they suffer from some combinations of the following problems.

High latency: Disk backup has no additional memory overhead, but the access latency is intolerably high under any correlated failures. Systems that take the third approach do not meet the single-digit μs latency requirement of remote memory even when paired with RDMA (Figure 1).

High cost: Replication has low latency, but it doubles memory consumption and network bandwidth requirements. Disk backup and replication represent the two extreme points in the performance-vs-efficiency tradeoff space (Figure 1).

Low availability: All three approaches lose availability to low latency memory when even a very small number of servers become unavailable. With the first approach, if a single server fails its data needs to be reconstituted from disk, which is a slow process. In the second and third approach, when even a small number of servers (e.g., three) fail simultaneously, some users will lose access to data. This is due to the fact that replication and erasure coding assign replicas and coding groups to *random* servers. Random data placement is susceptible to data loss when a small number of servers fail at the same time [27, 28] (Figure 2).

In this paper, we consider how to mitigate these problems and present Hydra, a low-latency, low-overhead, and highly available resilience mechanism for remote memory. While erasure codes are known for reducing storage overhead and for better load balancing, it is challenging for remote memory with μs-scale access requirements (preferably, 3-5μs) [47]. We demonstrate how to achieve resilient erasure-coded cluster

*These authors contributed equally to this work

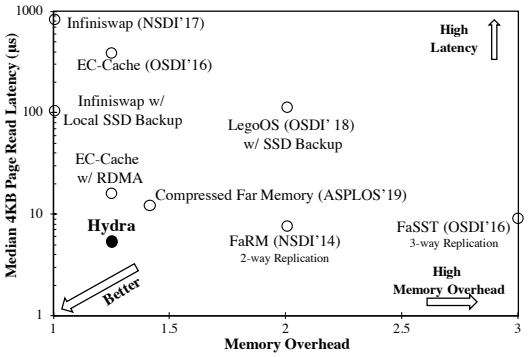


Figure 1: ***Performance-vs-efficiency tradeoff in the resilient cluster memory design space. Here, the Y-axis is in log scale.***

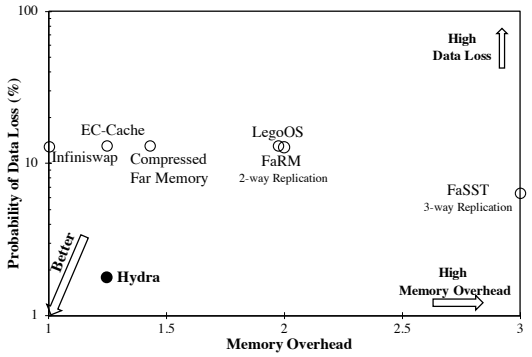


Figure 2: ***Availability-vs-efficiency tradeoff considering 1% simultaneous server failures in a 1000-machine cluster.***

memory with single-digit μs latency even under simultaneous failures at a reduced data amplification overhead.

We explore the challenges and tradeoffs for resilient remote memory without sacrificing application-level performance or incurring high overhead in the presence of correlated failures (§2). We also explore the trade-off between load balancing and high availability in the presence of simultaneous server failures. Our solution, Hydra, is a configurable resilience mechanism that applies online erasure coding to individual remote memory pages while maintaining high availability (§3). Hydra's carefully designed data path enables it to access remote memory pages within a single-digit μs median and tail latency (§4). Furthermore, we develop CodingSets, a novel coding group placement algorithm for erasure codes that provides load balancing while reducing the probability of data loss under correlated failures (§5).

We develop Hydra as a drop-in resilience mechanism that can be applied to existing remote memory frameworks [18, 22, 50, 65, 81]. We integrate Hydra with the two major remote memory approaches widely embraced today: disaggregated VMM (used by Infiniswap [50], and Leap [65]) and disaggregated VFS (used by Remote Regions [18]) (§6). Our evaluation using production workloads shows that Hydra achieves the best of both worlds (§7). Hydra closely matches the performance of replication-based resilience with 1.6× lower memory overhead with or without the presence of failures. At the same time, it improves latency and throughput of the benchmark applications by up to 64.78× and 20.61×, respectively, over SSD backup-based resilience with only 1.25× higher memory overhead. While providing resiliency, Hydra also improves the application-level performance by up to 4.35× over its counterparts. CodingSets reduces the probability of data loss under simultaneous server failures by about 10×. Hydra is available at https://github.com/SymbioticLab/hydra.

In this paper, we make the following contributions:

- Hydra is the first in-memory erasure coding scheme that achieves single-digit μs tail memory access latency.
- Novel analysis of load balancing and availability trade-off for distributed erasure codes.
- CodingSets is a new data placement scheme that balances availability and load balancing, while reducing probability of data loss by an order of magnitude during failures.

2 Background and Motivation

2.1 Remote Memory

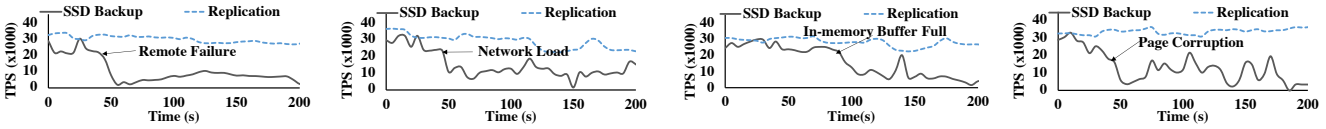
Remote memory exposes memory available in remote machines as a pool of memory shared by many machines. It is often implemented logically by leveraging stranded memory in remote machines via well-known abstractions, such as the file abstraction [18], remote memory paging [22,47,50,59,65], and virtual memory management for distributed OS [81]. In the past, specialized memory appliances for physical memory disaggregation were proposed as well [61, 63].

All existing remote-memory solutions use the 4KB page granularity. While some applications use huge pages for performance enhancement [57], the Linux kernel still performs paging at the basic 4KB level by splitting individual huge pages because huge pages can result in high amplification for dirty data tracking [23]. Existing remote-memory systems use disk backup [50, 81] and in-memory replication [46, 64] to provide availability during failures.

2.2 Failures in Remote Memory

The probability of failure or temporary unavailability is higher in a large remote-memory cluster, since memory is being accessed remotely. To illustrate possible performance penalties in the presence of such unpredictable events, we consider a resilience solution from the existing literature [50], where each page is asynchronously backed up to a local SSD. We run transaction processing benchmark TPC-C [16] on an in-memory database system, VoltDB [17]. We set VoltDB's available memory to 50% of its peak memory to force remote paging for up to 50% of its working set.

1. Remote Failures and Evictions Machine failures are the norm in large-scale clusters where thousands of machines crash over a year due to a variety of reasons, including software and hardware failures [31,33,38,88]. Concurrent failures within a rack or network segments are quite common and typ-



(a) *Remote failure* (b) *Background Network Load* (c) *Request burst* (d) *Page Corruption*

Figure 3: ***TPC-C throughput over time on VoltDB when 50% of the working set fits in memory. Arrows point to uncertainty injection time.***

ically occur dozens of times a year. Even cluster-wide power outage is not uncommon – occurs once or twice per year in a given data center. For example, during a recent cluster-wide power outage in Google Cloud, around 23% of the machines were unavailable for hours [6].

Without redundancy, applications relying on remote memory may fail when a remote machine fails or remote memory pages are evicted. As disk operations are significantly slower than the latency requirement of remote memory, disk-based fault-tolerance is far from being practical. In the presence of a remote failure, VoltDB experiences almost 90% throughput loss (Figure 3a); throughput recovery takes a long time after the failure happens.

2. Background Network Load Network load throughout a large cluster can experience significant fluctuations [41, 53], which can inflate RDMA latency and application-level stragglers, causing unpredictable performance issues [40, 89]. In the presence of an induced bandwidth-intensive background load, VoltDB throughput drops by about 50% (Figure 3b).

3. Request Bursts Applications can have bursty memory access patterns. Existing solutions maintain an in-memory buffer to absorb temporary bursts [18, 50, 74]. However, as the buffer ties remote access latency to disk latency when it is full, the buffer can become the bottleneck when a workload experiences a prolonged burst. While a page read from remote memory is still fast, backup page writes to the local disk become the bottleneck after the 100^{th} second in Figure 3c. As a result, throughput drops by about 60%.

4. Memory Corruption During remote memory access, if any one of the remote servers experiences a corruption, or if the memory gets corrupted over the network a memory corruption event will occur. In such case, disk access causes failure-like performance loss (Figure 3d).

Performance vs. Efficiency Tradeoff for Resilience In all of these scenarios, the obvious alternative – in-memory $2\times$ or $3\times$ replication [46, 64] – is effective in mitigating a small-scale failure, such as the loss of a single server (Figure 3a). When an in-memory copy becomes unavailable, we can switch to an alternative. Unfortunately, replication incurs high memory overhead in proportion to the number of replicas. This defeats the purpose of remote memory. Hedging requests to avoid stragglers [41] in a replicated system doubles its bandwidth requirement as well.

This leads to an impasse: one has to either settle for high latency in the presence of a failure or incur high memory overhead. Figure 1 depicts this performance-vs-efficiency tradeoff under failures and memory usage overhead to provide resilience. Beyond the two extremes in the tradeoff space, there are two primary alternatives to achieve high resilience with low overhead. The first is replicating pages to remote memory after compressing them (e.g., using zswap) [58], which improves the tradeoff in both dimensions. However, its latency can be more than 10μs when data is in remote memory. Especially, during resource scarcity, the presence of a prolonged burst in accessing remote compressed pages can even lead to orders of magnitude higher latency due to the demand spike in both CPU and local DRAM consumption for decompression. Besides, this approach faces similar issues as replication such as latency inflation due to stragglers.

The alternative is erasure coding, which has recently made its way from disk-based storage to in-memory cluster caching to reduce storage overhead and improve load balancing [20,25, 76, 83, 84, 86]. Typically, an object is divided into k *data splits* and encoded to create r equal-sized *parity splits* ($k > r$), which are then distributed across $(k+r)$ failure domains. Existing erasure-coded memory solutions deal with large objects (e.g., larger than 1 MB [76]), where hundreds-of-μs latency of the TCP/IP stack can be ignored. Simply replacing TCP with RDMA is not enough either. For example, the EC-Cache with RDMA (Figure 1) provides a lower storage overhead than compression but with a latency around 20μs.

Last but not least, all of these approaches experience high unavailability in the presence of correlated failures [28].

2.3 Challenges in Erasure-Coded Memory

High Latency Individually erasure coding 4 KB pages that are already small lead to even smaller data chunks ($\frac{4}{k}$ KB), which contributes to the higher latency of erasure-coded remote memory over RDMA due to following primary reasons:

1. ***Non-negligible coding overhead:*** When using erasure codes with on-disk data or over slower networks that have hundreds-of-μs latency, its 0.7μs encoding and 1.5μs decoding overheads can be ignored. However, they become non-negligible when dealing with DRAM and RDMA.
2. ***Stragglers and errors:*** As erasure codes require k splits before the original data can be constructed, any straggler can slow down a remote read. To detect and correct an error, erasure codes require additional splits; an extra read adds another round-trip to double the overall read latency.
3. ***Interruption overhead:*** Splitting data also increases the total number of RDMA operations for each request. Any

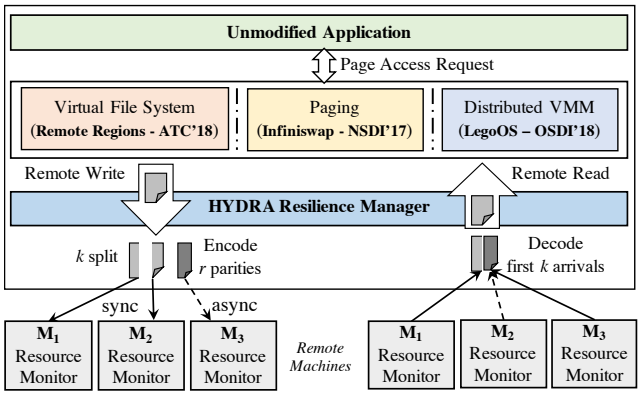


Figure 4: ***Resilience Manager provides with resilient, erasure-coded remote memory abstraction. Resource Monitor manages the remote memory pool. Both can be present in a machine.***

context switch in between can further add to the latency.

4. ***Data copy overhead:*** In a latency-sensitive system, additional data movement can limit the lowest possible latency. During erasure coding, additional data copy into different buffers for data and parity splits can quickly add up.

Availability Under Simultaneous Failures Existing erasure coding schemes can handle a small-scale failure without interruptions. However, when a relatively modest number of servers fail or become unavailable at the same time (e.g., due to a network partition or a correlated failure event), they are highly susceptible to losing availability to some of the data.

This is due to the fact that existing erasure coding schemes generate coding groups on random sets of servers [76]. In a coding scheme with k data and r parity splits, an individual coding group, will fail to decode the data if $r+1$ servers fail simultaneously. Now in a large cluster with $r+1$ failures, the probability that those $r+1$ servers will fail for a *specific* coding group is low. However, when coding groups are generated randomly (i.e., each one of them compromises a random set of $k+r$ servers), and there are a large number of coding groups per server, then the probability that those $r+1$ servers will affect *any* coding group in the cluster is much higher. Therefore, state-of-the-art erasure coding schemes, such as EC-Cache, will experience a very high probability of unavailability even when a very small number of servers fail simultaneously.

3 Hydra Architecture

Hydra is an erasure-coded resilience mechanism for existing remote-memory techniques to provide better performance-efficiency tradeoff under remote failures while ensuring high availability under simultaneous failures. It has two main components (Figure 4): (i) **Resilience Manager** coordinates erasure-coded resilience operations during remote read/write; (ii) **Resource Monitor** handles the memory management in a remote machine. Both can be present in every machine and work together without central coordination.

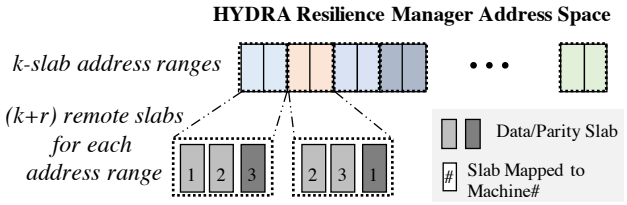


Figure 5: ***Hydra's address space is divided into fixed-size address ranges, each of which spans*** $(k+r)$ ***memory slabs in remote machines; i.e.,*** k ***for data and*** r ***for parity (*** k***=2 and*** r***=1 in this figure).***

3.1 Resilience Manager

Hydra Resilience Manager provides remote memory abstraction to a client machine. When an unmodified application accesses remote memory through different state-of-the-art remote-memory solutions (e.g., via VFS or VMM), the Resilience Manager transparently handles all aspects of RDMA communication and erasure coding. Each client has its own Resilience Manager that handles slab placement through CodingSets, maintains remote slab-address mapping, performs erasure-coded RDMA read/write. Resilience Manager communicates to Resource Monitor(s) running on remote memory host machines, performs remote data placement, and ensures resilience. As a client's Resilience Manager is responsible for the resiliency of its remote data, the Resilience Managers do not need to coordinate with each other.

Following the typical (k,r) erasure coding construction, the Resilience Manager divides its remote address space into fixed-size *address ranges*. Each address range resides in $(k+r)$ remote *slabs*: k slabs for page data and r slabs for parity (Figure 5). Each of the $(k+r)$ slabs of an address range are distributed across $(k+r)$ independent failure domains using CodingSets (§5). Page accesses are directed to the designated $(k+r)$ machines according to the address–slab mapping. Although remote I/O happens at the page level, the Resilience Manager coordinates with remote Resource Monitors to manage coarse-grained memory slabs to reduce metadata overhead and connection management complexity.

3.2 Resource Monitor

Resource Monitor manages a machine's local memory and exposes them to the remote Resilience Manager in terms of fixed-size (*SlabSize*) memory slabs. Different slabs can belong to different machines' Resilience Manager. During each control period (*ControlPeriod*), the Resource Monitor tracks the available memory in its local machine and proactively allocates (reclaims) slabs to (from) remote mapping when memory usage is low (high). It also performs slab regeneration during remote failures or corruptions.

Fragmentation in Remote Memory During the registration of Resource Monitor(s), Resilience Manager registers the RDMA memory regions and allocates slabs on the remote machines based on its memory demand. Memory regions are usually large (by default, 1GB) and the whole address space is

homogeneously splitted. Moreover, RDMA drivers guarantee the memory regions are generated in a contiguous physical address space to ensure faster remote-memory access. Hydra introduces no additional fragmentation in remote machines.

3.3 Failure Model

Assumptions In a large remote-memory cluster, (a) remote servers may crash or networks may become partitioned; (b) remote servers may experience memory corruption; (c) the network may become congested due to background traffic; and (d) workloads may have bursty access patterns. These events can lead to catastrophic application-failures, high tail latencies, or unpredictable performance. Hydra addresses all of these uncertainties in its failure domain. Although Hydra withstands a remote-network partition, as there is no local-disk backup, it cannot handle local-network failure. In such cases, the application is anyways inaccessible.

Single vs. Simultaneous Failure A single node failure means the unavailability of slabs in a remote machine. In such an event, all the data or parity allocated on the slab(s) become unavailable. As we spread the data and parity splits for a page across multiple remote machines (§5), during a single node failure, we assume that only a single data or parity split for that page is being affected.

Simultaneous host failures typically occur due to large-scale failures, such as power or network outage that cause multiple machines to become unreachable. In such a case, we assume multiple data and/or parity splits for a page become unavailable. Note that in both cases, the data is unavailable, but not compromised. Resilience Manager can detect the unreachability and communicate to other available Resource Monitor(s) on to regenerate specific slab(s).

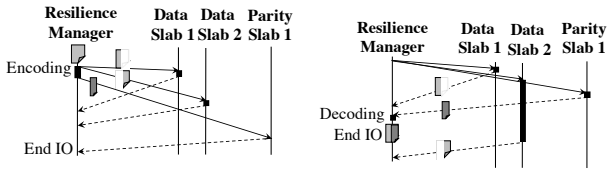
4 Resilient Data Path

Hydra can operate on different resilient modes based on a client's need – **(a)** *Failure Recovery*: provides resiliency in the presence of any remote failure or eviction; **(b)** *Corruption Detection*: only detects the presence of corruption in remote memory; **(c)** *Corruption Correction*: detects and corrects remote memory corruption; and **(d)** *EC-only mode*: provides erasure-coded faster remote-memory data path without any resiliency guarantee. Note that both of the corruption modes by default inherit the *Failure Recovery* mode.

Before initiating the Resilience Manager, one needs to configure Hydra to a specific mode according to the resilience requirements and memory overhead concerns (Table 1). Multiple resilience modes cannot act simultaneously, and the modes do not switch dynamically during runtime. In this section, we present Hydra's data path design to address the resilience challenges mentioned in §2.3.

4.1 Hydra Remote Memory Data Path

To minimize erasure coding's latency overheads, Resilience Manager's data path incorporate following design principles.



(a) *Remote Write* (b) *Remote Read*

Figure 6: ***To handle failures, Hydra (a) first writes data splits, then encodes/writes parities to hide encoding latency; (b) reads from $k+\Delta$ slabs to avoid stragglers, finishes with first k arrivals.***

4.1.1 Asynchronously Encoded Write

To hide the erasure coding latency, existing systems usually perform batch coding where multiple pages are encoded together. The encoder waits until a certain number of pages are available. This idle waiting time can be insignificant compared to disk or slow network (e.g., TCP) access. However, to maintain the tail latency of a remote I/O within the single-digit μs range, this "batch waiting" time needs to be avoided.

During a remote write, Resilience Manager applies erasure coding within each individual page by dividing it into k splits (for a 4 KB page, each split size is $\frac{4}{k}$KB), encodes these splits using Reed-Solomon (RS) codes [77] to generate r parity splits. Then, it writes these $(k+r)$ splits to different $(k+r)$ slabs that have already been mapped to unique remote machines. Each Resilience Manager can have their own choice of k and r. This individual page-based coding decreases latency by avoiding the "batch waiting" time. Moreover, the Resilience Manager does not have to read unnecessary pages within the same batch during remote reads, which reduces bandwidth overhead. Distributing remote I/O across many remote machines increases I/O parallelism too.

Resilience Manager sends the data splits first, then it encodes and sends the parity splits asynchronously. Decoupling the two hides encoding latency and subsequent write latency for the parities without affecting the resilience guarantee. In the absence of failure, any k successful writes of the $(k+r)$ allow the page to be recovered. However, to ensure resilience guarantee for r failures, all $(k+r)$ must be written. In the *failure recovery* mode, a write is considered complete after all $(k+r)$ have been written. In the *corruption correction (detection)* mode, to correct (detect) Δ corruptions, a write waits for $k+2\Delta+1$ $(k+\Delta)$ to be written. If the acknowledgement fails to reach the Resilience Manager due to a failure in the remote machine, the write for that split is considered failed. Resilience Manager tries to write that specific split(s) after a timeout period to another remote machine. Figure 6a depicts the timeline of a page write in the *failure recovery mode*.

4.1.2 Late-Binding Resilient Read

During read, any k out of the $k+r$ splits suffice to reconstruct a page. However, in *failure recovery* mode, to be resilient in the presence of Δ failures, during a remote read, Hydra Resilience Manager reads from $k+\Delta$ randomly chosen splits

Resilience Mode	# of Errors	Minimum # of Splits	Memory Overhead
Failure Recovery	r	k	$1+\frac{r}{k}$
Corruption Detection	Δ	$k+\Delta$	$1+\frac{\Delta}{k}$
Corruption Correction	Δ	$k+2\Delta+1$	$1+\frac{2\Delta+1}{k}$
EC-only	–	k	$1+\frac{r}{k}$

Table 1: ***Minimum number of splits needs to be written to/read from remote machines for resilience during a remote I/O.***

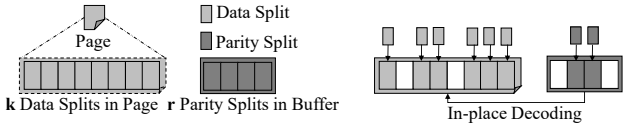
in parallel. A page can be decoded as soon as any k splits arrive out of $k+\Delta$. The additional Δ reads mitigate the impact of stragglers on tail latency as well. Figure 6b provides an example of a read operation in the *failure recovery* mode with $k=2$ and $\Delta=1$, where one of the data slabs (Data Slab 2) is a straggler. $\Delta=1$ is often enough in practice.

If simply "detect and discard corrupted memory" is enough for any application, one can configure Hydra with *corruption detection* mode and avoid the extra memory overhead of *corruption correction* mode. In *corruption detection* mode, before decoding a page, the Resilience Manager waits for $k+\Delta$ splits to arrive to *detect* Δ corruptions. After the detection of a certain amount of corruptions, Resilience Manager marks the machine(s) with corrupted splits as probable erroneous machines, initiates a background slab recovery operation, and avoids them during future remote I/O.

To correct the error, in *corruption correction* mode, when an error is detected, it requests additional $\Delta+1$ reads from the rest of the $k+r$ machines. Otherwise, the read completes just after the arrival of the $k+\Delta$ splits. If the error rate for a remote machine exceeds a user-defined threshold (*ErrorCorrectionLimit*), subsequent read requests involved with that machine initiates with $k+2\Delta+1$ split requests as there is a high probability to reach an erroneous machine. This will reduce the wait time for additional $\Delta+1$ reads. This continues until the error rate for the involved machine gets lower than the *ErrorCorrectionLimit*. If this continues for long and/or the error rate goes beyond another threshold (*SlabRegenerationLimit*), Resilience Manager initiates a slab regeneration request for that machine.

One can configure Hydra with *EC-only* mode to access erasure-coded remote memory and benefit from the fast data path without any resiliency guarantee. In this mode, a remote I/O completes just after writing/reading any k splits. Table 1 summarizes the minimum number of splits the Resilience Manager requires to write/read during a remote I/O operation to provide resiliency in different modes.

Overhead of Replication To remain operational after r failures, in-memory replication requires at least $r+1$ copies of an entire 4 KB page, and hence the memory overhead is $(r+1)\times$. However, a remote I/O operation can complete just after the confirmation from one of the $r+1$ machines. To detect and fix Δ corruptions, replication needs $\Delta+1$ and $2\Delta+1$ copies of the *entire* page, respectively. Thus, to provide the correctness guarantee over Δ corruptions, replication needs to wait until



(a) *Remote Write* (b) *Remote Read*

Figure 7: ***Hydra performs in-place coding with an extra buffer of r splits to reduce the data-copy latency.***

it writes to or reads from at least $2\Delta+1$ of the replicas along with a memory overhead of $(2\Delta+1)\times$.

4.1.3 Run-to-Completion

As Resilience Manager divides a 4 KB page into k smaller pieces, RDMA messages become smaller. In fact, their network latency decrease to the point that run-to-completion becomes more beneficial than a context switch. Hence, to avoid interruption-related overheads, the remote I/O request thread waits until the RDMA operations are done.

4.1.4 In-Place Coding

To reduce the number of data copies, Hydra Resilience Manager uses in-place coding with an extra buffer of r splits. During a write, the data splits are always kept in-page while the encoded r parities are put into the buffer (Figure 7a). Likewise, during a read, the data splits arrive at the page address, and the parity splits find their way into the buffer (Figure 7b).

In the failure recovery mode, a read can complete as soon as any k valid splits arrive. Corrupted/straggler data split(s) can arrive late and overwrite valid page data. To address this, as soon as Hydra detects the arrival of k valid splits, it deregisters relevant RDMA memory regions. It then performs decoding and directly places the decoded data in the page destination. Because the memory region has already been deregistered, any late data split cannot access the page. During all remote I/O, requests are forwarded directly to RDMA dispatch queues without additional copying.

4.2 Handling Uncertainties

Remote Failure Hydra uses reliable connections (RC) for all RDMA communication. Hence, we consider unreachability due to machine failures/reboots or network partition as the primary cause of failure. When a remote machine becomes unreachable, the Resilience Manager is notified by the RDMA connection manager. Upon disconnection, it processes all the in-flight requests in order first. For ongoing I/O operations, it resends the I/O request to other available machines. Since RDMA guarantees strict ordering, in the read-after-write case, read requests will arrive at the same RDMA dispatch queue after write requests; hence, read requests will not be served with stale data. Finally, Hydra marks the failed slabs and future requests are directed to the available ones. If the Resource Monitor in the failed machine revives and communicates later, Hydra reconsiders the machine for further remote I/O.

Adaptive Slab Allocation/Eviction Resource Monitor al-

locates memory slabs for Resilience Managers as well as proactively frees/evicts them to avoid local performance impacts (Figure 8). It periodically monitors local memory usage and maintains a headroom to provide enough memory for local applications. When the amount of free memory shrinks below the headroom (Figure 8a), the Resource Monitor first proactively frees/evicts slabs to ensure local applications are unaffected. To find the eviction candidates, we avoid random selection as it has a higher likelihood of evicting a busy slab. Rather, we uses the decentralized batch eviction algorithm [50] to select the least active slabs. To evict E slabs, we contact $(E+E')$ slabs (where $E' \leq E$) and find the least-frequently-accessed slabs among them. This doesn't require to maintain a global knowledge or search across all the slabs.

When the amount of free memory grows above the headroom (Figure 8b), the Resource Monitor first attempts to make the local Resilience Manager to reclaim its pages from remote memory and unmap corresponding remote slabs. Furthermore, it proactively allocates new, unmapped slabs that can be readily mapped and used by remote Resilience Managers.

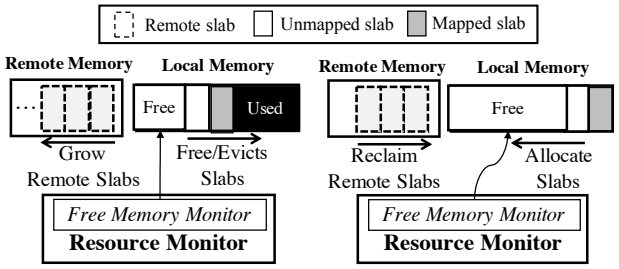
Background Slab Regeneration The Resource Monitor also regenerates unavailable slabs – marked by the Resilience Manager – in the background. During regeneration, writes to the slab are disabled to prevent overwriting new pages with stale ones; reads can still be served without interruption.

Hydra Resilience Manager uses the placement algorithm to find a new regeneration slab in a remote Resource Monitor with a lower memory usage. It then hands over the task of slab regeneration to that Resource Monitor. The selected Resource Monitor decodes the unavailable slab by directly reading the k randomly-selected remaining valid slab for that address region. Once regeneration completes, it contacts the Resilience Manager to mark the slab as available. Requests thereafter go to the regenerated slab.

5 CodingSets for High Availability

Hydra uses CodingSets, a novel coding group placement scheme to perform load-balancing while reducing the probability of data loss. Prior works show orders-of-magnitude more frequent data loss due to events causing multiple nodes to fail simultaneously than data loss due to independent node failures [27, 31]. Several scenarios can cause multiple servers to fail or become unavailable simultaneously, such as network partitions, partial power outages, and software bugs. For example, a power outage can cause 0.5%-1% machines to fail or go offline concurrently [28]. In case of Hydra, data loss will happen if a concurrent failure kills more than $r+1$ of $(k+r)$ machines for a particular *coding group*.

We are inspired by copysets, a scheme for preventing data loss under correlated failures in replication [27, 28], which constraints the number of replication groups, in order to *reduce the frequency of data loss events*. Using the same terminology as prior work, we define each unique set of $(k+r)$ servers within a coding group as a *copyset*. The number of



(a) *High memory pressure* (b) *Low memory pressure*

Figure 8: ***Resource Monitor proactively allocates memory for remote machines and frees local memory pressure.***

copysets in a single coding group will be: $\binom{k+r}{r+1}$. For example, in an (8+2) configuration, where nodes are numbered $1, 2, \ldots, 10$, the 3 nodes that will cause failure if they fail at the same time (i.e., copysets) will be every 3 combinations of 10 nodes: $(1,2,3), (1,2,4), \ldots, (8,9,10)$, and the total number of copysets will be $\binom{10}{3} = 120$.

For a data loss event impacting exactly $r+1$ random nodes simultaneously, the probability of losing data of a single specific coding group: $\mathbb{P}[Group] = \frac{\text{Num. of Copysets in Coding Group}}{\text{Total Copysets}} = \frac{\binom{k+r}{r+1}}{\binom{N}{r+1}}$, where N is the total number of servers.

In a cluster with more than $(k+r)$ servers, we need to use more than one coding group. However, if each server is a member of a single coding group, hot spots can occur if one or more members of that group are overloaded. Therefore, for load-balancing purposes, a simple solution is to allow each server to be a member of multiple coding groups, in case some members of a particular coding group are over-loaded at the time of online coding.

Assuming we have G *disjoint* coding groups, and the correlated failure rate is $f\%$, the total probability of data loss is: $1-(1-\mathbb{P}[Group]\cdot G)^{\binom{N\cdot f}{r+1}}$. We define disjoint coding groups where the groups do not share any copysets; or in other words, they do not overlap by more than r nodes.

Strawman: Multiple Coding Groups per Server In order to equalize load, we consider a scheme where each slab forms a coding group with the least-loaded nodes in the cluster at coding time. We assume the nodes that are least loaded at a given time are distributed randomly, and the number of slabs per server is S. When $S\cdot(r+k) \ll N$, the coding groups are highly likely to be disjoint [28], and the number of groups is equal to: $G = \frac{N\cdot S}{k+r}$.

We call this placement strategy the *EC-Cache scheme*, as it produces a random coding group placement used by the prior state-of-the-art in-memory erasure coding system, EC-Cache [76]. In this scheme, with even a modest number of slabs per server, a high number of combinations of $r+1$ machines will be a copyset. In other words, even a small number of simultaneous node failures in the cluster will result in data loss. When the number of slabs per server is high, almost

every combination of only $r+1$ failures across the cluster will cause data loss. Therefore, to reduce the probability of data loss, we need to minimize the number of copysets, while achieving sufficient load balancing.

CodingSets: Reducing Copysets for Erasure Coding To this end, we propose CodingSets, a novel load-balancing scheme, which reduces the number of copysets for distributed erasure coding. Instead of having each node participate in several coding groups like in EC-Cache, in our scheme, each server belongs to a single, *extended* coding group. At time of coding, $(k+r)$ slabs will still be considered together, but the nodes participating in the coding group are chosen from a set of $(k+r+l)$ nodes, where l is the load-balancing factor. The nodes chosen within the extended group are the least loaded ones. While extending the coding group increases the number of copysets (instead of $\binom{k+r}{r+1}$ copysets, now each extended coding group creates $\binom{k+r+l}{r+1}$ copysets, while the number of groups is $G = \frac{N}{k+r+l}$), it still has a significantly lower probability of data loss than having each node belong to multiple coding groups. Hydra uses CodingSets as its load balancing and slab placement policy. We evaluate it in Section 7.2.

Tradeoff Note that while CodingSets reduces the probability of data loss, it does not reduce the expected amount of data lost over time. In other words, it reduces the number of data loss events, but each one of these events will have a proportionally higher magnitude of data loss (i.e., more slabs will be affected) [28]. Given that our goal with Hydra is high availability, we believe this is a favorable trade off. For example, providers often provide an availability SLA, that is measured by the service available time (e.g., the service is available 99.9999% of the time). CodingSets would optimize for such an SLA, by minimizing the frequency of unavailability events.

6 Implementation

Resilience Manager is implemented as a loadable kernel module for Linux kernel 4.11 or later. Kernel-level implementation facilitates its deployment as an underlying block device for different remote-memory systems [18, 50, 81]. We integrated Hydra with two remote-memory systems: Infiniswap, a disaggregated VMM and Remote Regions, a disaggregated VFS. All I/O operations (e.g., slab mapping, memory registration, RDMA posting/polling, erasure coding) are independent across threads and processed without synchronization. All RDMA operations use RC and one-sided RDMA verbs (RDMA WRITE/READ). Each Resilience Manager maintains one connection for each active remote machine. For erasure coding, we use x86 AVX instructions and the ISA library [8] that achieves over 4 GB/s encoding throughput per core for (8+2) configuration in our evaluation platform.

Resource Monitor is implemented as a user-space program. It uses RDMA SEND/RECV for all control messages.

7 Evaluation

We evaluate Hydra on a 50-machine 56 Gbps InfiniBand CloudLab cluster against Infiniswap [50], Leap [65] (disaggregated VMM) and Remote Regions [18] (disaggregated VFS). Our evaluation addresses the following questions:

- Does it improve the resilience of cluster memory? (§7.1)
- Does it improve the availability? (§7.2)
- What is its overhead and sensitivity to parameters? (§7.3)
- How much TCO savings can we expect? (§7.4)
- What is its benefit over a persistent memory setup? (§7.5)

Methodology Unless otherwise specified, we use k=8, r=2, and Δ=1, targeting 1.25× memory and bandwidth overhead. We select r=2 because late binding is still possible even when one of the remote slab fails. The additional read Δ=1 incurs 1.125× bandwidth overhead during reads. We use 1GB *SlabSize*, The additional number of choices for eviction $E' = 2$. Free memory headroom is set to 25%, and the control period is set to 1 second. Each machine has 64 GB of DRAM and 2× Intel Xeon E5-2650v2 with 32 virtual cores.

We compare Hydra against the following alternatives:

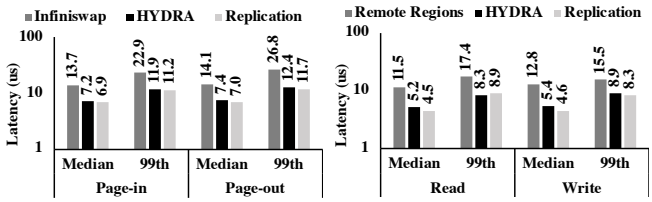
- **SSD Backup**: Each page is backed up in a local SSD for the minimum 1× remote memory overhead. We consider both disaggregated VMM and VFS systems.
- **Replication**: We directly write each page over RDMA to two remote machines' memory for a 2× overhead.
- **EC-Cache w/ RDMA**: Implementation of the erasure coding scheme in EC-Cache [76], but implemented on RDMA.

Workload Characterization Our evaluation consists of both micro-benchmarks and cluster-scale evaluations with real-world applications and workload combinations.

- We use TPC-C [16] on VoltDB [17]. We perform 5 different types of transactions to simulate an order-entry environment. We set 256 warehouses and 8 sites and run 2 million transactions. Here, the peak memory usage is 11.5 GB.
- We use Facebook's ETC, SYS workloads [21] on Memcached [12]. First, we use 10 million SETs to populate the Memcached server. Then we perform another 10 million operations (for ETC: 5% SETs, 95% GETs, for SYS: 25% SETs, 75% GETs). The key size is 16 bytes and 90% of the values are evenly distributed between 16–512 bytes. Peak memory usages are 9 GB for ETC and 15 GB for SYS.
- We use PageRank on PowerGraph [48] and Apache Spark/GraphX [49] to measure the influence of Twitter users on followers on a graph with 11 million vertices [56]. Peak memory usages are 9.5 GB and 14 GB, respectively.

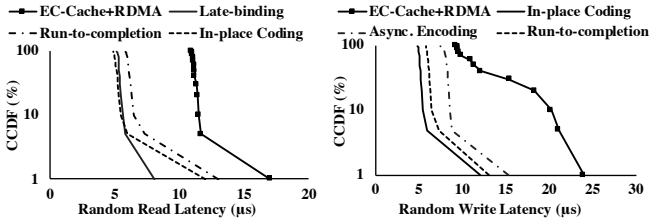
7.1 Resilience Evaluation

We evaluate Hydra both in the presence and absence of failures with microbenchmarks and real-world applications.



(a) *Disaggregated VMM Latency* (b) *Disaggregated VFS Latency*

Figure 9: ***Hydra provides better latency characteristics during both disaggregated VMM and VFS operations.***



(a) *Remote Read* (b) *Remote Write*

Figure 10: ***Hydra latency breakdown through CCDF.***

7.1.1 Latency Characteristics

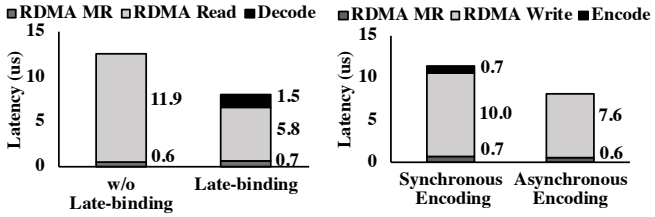
First, we measure Hydra's latency characteristics with micro-benchmarks in the absence of failures. Then we analyze the impact of its design components.

Disaggregated VMM Latency We use a simple application with its working set size set to 2GB. It is provided 1GB memory to ensure that 50% of its memory accesses cause paging. While using disaggregated memory for remote page-in, Hydra improves page-in latency over Infiniswap with SSD backup by 1.79× at median and 1.93× at the 99th percentile. Page-out latency is improved by 1.9× and 2.2× over Infiniswap at median and 99th percentile, respectively. Replication provides at most 1.1× improved latency over Hydra, while incurring 2× memory and bandwidth overhead (Figure 9a).

Disaggregated VFS Latency We use `fio` [5] to generate one million random read/write requests of 4 KB block I/O. During reads, Hydra provides improved latency over Remote Regions by 2.13× at median and 2.04× at the 99th percentile. During writes, Hydra also improves the latency over Remote Regions by 2.22× at median and 1.74× at the 99th percentile. Replication has a minor latency gain over Hydra, improving latency by at most 1.18× (Figure 9b).

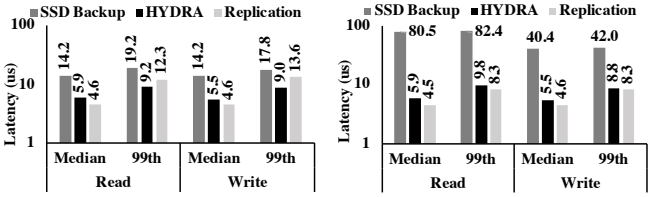
Benefit of Data Path Components Erasure coding over RDMA (i.e., EC-Cache with RDMA) performs worse than disk backup due to its coding overhead. Figure 10 shows the benefit of Hydra's data path components to reduce the latency.

1. Run-to-completion avoids interruptions during remote I/O, reducing the median read and write latency by 51%.
2. In-place coding saves additional time for data copying, which substantially adds up in remote-memory systems, reducing 28% of the read and write latency.



(a) *Read breakdown* (b) *Write breakdown*

Figure 11: ***Hydra latency breakdown at the 99th percentile.***



(a) *Background network flow* (b) *Remote Failures*

Figure 12: ***Latency in the presence of uncertainty events.***

3. Late binding specifically improves the tail latency during remote read by 61% by avoiding stragglers. The additional read request increases the median latency only by 6%.
4. Asynchronous encoding hides erasure coding overhead during writes, reducing the median write latency by 38%.

Tail Latency Breakdown The latency of Hydra consists of the time for (i) RDMA Memory Registration (MR), (ii) actual RDMA read/write, and (iii) erasure coding. Even though decoding a page takes about 1.5μs, late binding effectively improves the tail latency by 1.55× (Figure 11a). During writes, asynchronous encoding hides encoding latency and latency impacts of straggling splits, improving tail latency by 1.34× w.r.t. synchronous encoding (Figure 11b). At the presence of corruption ($r = 3$), accessing extra splits increases the tail latency by 1.51× and 1.09× for reads and writes, respectively.

7.1.2 Latency Under Failures

Background Flows We generate RDMA flows on the remote machine constantly sending 1 GB messages. Unlike SSD backup and replication, Hydra ensures consistent latency due to late binding (Figure 12a). Hydra's latency improvement over SSD backup is 1.97–2.56×. It even outperforms replication at the tail read (write) latency by 1.33× (1.50×).

Remote Failures Both read and write latency are disk-bound when it's necessary to access the backup SSD (Figure 12b). Hydra reduces latency over SSD backup by 8.37–13.6× and 4.79–7.30× during remote read and write, respectively. Furthermore, it matches the performance of replication.

7.1.3 Application-Level Performance

We now focus on Hydra's benefits for real-world memory-intensive applications and compare it with that of SSD backup and replication. We consider container-based application de-

		TPS/OPS (thousands)		Latency (ms)			
				50th		99th	
		HYD	REP	HYD	REP	HYD	REP
VoltDB	100%	39.4	39.4	52.8	52.8	134.0	134.0
	75%	36.1	35.3	56.3	56.1	142.0	143.0
	50%	32.3	34.0	57.8	59.0	161.0	168.0
ETC	100%	123.0	123.0	123.0	123.0	257.0	257.0
	75%	119.0	125.0	120.0	121.0	255.0	257.0
	50%	119.0	119.0	118.0	122.0	254.0	264.0
SYS	100%	108.0	108.0	125.0	125.0	267.0	267.0
	75%	100.0	104.0	120.0	125.0	262.0	305.0
	50%	101.0	102.0	117.0	123.0	257.5	430.0

Table 2: ***Hydra (HYD) provides similar performance to replication (REP) for VoltDB and Memcached workloads (ETC and SYS). Higher is better for throughput; Lower is better for latency.***

	Apache Spark/GraphX Completion Time (s)			PowerGraph Completion Time (s)		
	100%	75%	50%	100%	75%	50%
Hydra	77.91	105.41	191.93	73.10	66.90	68.00
Replication	77.91	91.89	195.54	73.10	73.30	73.70

Table 3: ***Hydra also provides similar completion time to replication for graph analytic applications.***

ployment [82] and run each application in an `lxc` container with a memory limit to fit 100%, 75%, 50% of the peak memory usage for each application. For 100%, applications run completely in memory. For 75% and 50%, applications hit their memory limits and performs remote I/O via Hydra.

We present Hydra's application-level performance against replication (Table 2 and Table 3) to show that it can achieve similar performance with a lower memory overhead even in the absence of any failures. For brevity, we omit the results for SSD backup, which performs much worse than both Hydra and replication – albeit with no memory overhead.

For VoltDB, when half of its data is in remote memory, Hydra achieves 0.82× throughput and almost transparent latency characteristics compared to the fully in-memory case.

For Memcached, at 50% case, Hydra achieves 0.97× throughput with read-dominant ETC workloads and 0.93× throughput with write-intensive SYS workloads compared to the 100% scenario. Here, latency overhead is almost zero.

For graph analytics, Hydra could achieve almost transparent application performance for PowerGraph; thanks to its optimized heap management. However, it suffers from increased job completion time for GraphX due to massive thrashing of in-memory and remote memory data – the 14 GB working set oscillates between paging-in and paging-out. This causes bursts of RDMA reads and writes. Even then, Hydra outperforms Infiniswap with SSD backup by 8.1×. Replication does not have significant gains over Hydra.

Performance with Leap Hydra's drop-in resilience mechanism is orthogonal to the functionalities of remote-memory frameworks. To observe Hydra's benefit even with faster in-kernel lightweight remote-memory data path, we integrate it to Leap [65] and run VoltDB and PowerGraph with 50% remote-memory configurations.

Leap waits for an interrupt during a 4KB remote I/O, whereas Hydra splits a 4KB page into smaller chunks and performs asynchronous remote I/O. Note that RDMA read for 4KB-vs-512B is 4μs-vs-1.5μs. With self-coding and run-to-completion, Hydra provides competitive performance guarantees as Leap for both VoltDB (0.99× throughput) and PowerGraph (1.02× completion time) in the absence of failures.

7.1.4 Application Performance Under Failures

Now we analyze Hydra's performance in the presence of failures and compare against the alternatives. In terms of impact on applications, we first go back to the scenarios discussed in Section 2.2 regarding to VoltDB running with 50% memory constraint. Except for the corruption scenario where we set r=3, we use Hydra's default parameters. At a high level, we observe that Hydra performs similar to replication with 1.6× lower memory overhead (Figure 13).

Next, we start each benchmark application in 50% settings and introduce one remote failure while it is running. We select a Resource Monitor with highest slab activities and kill it. We measure the application's performance while the Resilience Manager initiates the regeneration of affected slabs.

Hydra's application-level performance is transparent to the presence of remote failure. Figure 14 shows Hydra provides almost similar completion times to that of replication at a lower memory overhead in the presence of remote failure. In comparison to SSD backup, workloads experience 1.3–5.75× lower completion times using Hydra. Hydra provides similar performance at the presence of memory corruption. Completion time gets improved by 1.2–4.9× w.r.t. SSD backup.

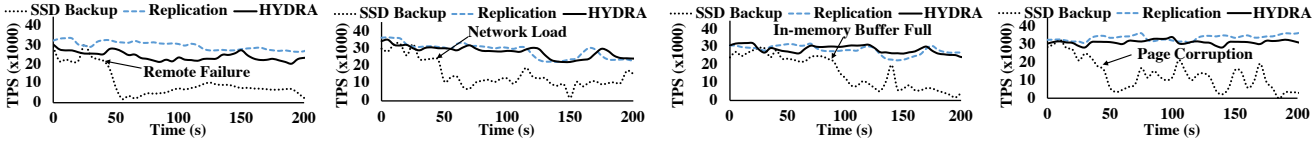
7.2 Availability Evaluation

In this section, we evaluate Hydra's availability and load balancing characteristics in large clusters.

7.2.1 Analysis of CodingSets

We compare the availability and load balancing of Hydra with EC-Cache and power-of-two-choices [67]. In CodingSets, each server is attached to a disjoint coding group. During encoded write, the $(k+r)$ least loaded nodes are chosen from a subset of the $(k+r+l)$ coding group at the time of replication. EC-Cache simply assigns slabs to coding groups comprising of random nodes. Power-of-two-choices finds two candidate nodes at random for each slab, and picks the less loaded one.

Probability of Data Loss Under Simultaneous Failures To evaluate the probability of data loss of Hydra under different scenarios in a large cluster setting, we compute the probability of data loss under the three schemes. Note that, in terms of data loss probability, we assume EC-Cache and power of two choices select random servers, and are therefore equivalent. Figure 15 compares the probabilities of loss for different parameters on a 1000-machine cluster. Our baseline comparison is against the best case scenario for EC-Cache and power-of-two-choices, where the number of slabs per



(a) *Remote failure* (b) *Remote Network Load* (c) *Request Burst* (d) *Page Corruption*

Figure 13: ***Hydra throughput with the same setup in Figure 3.***

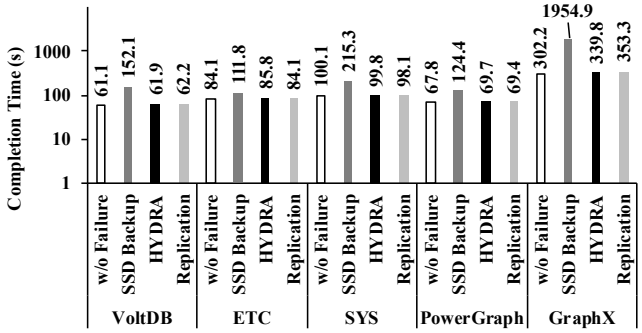
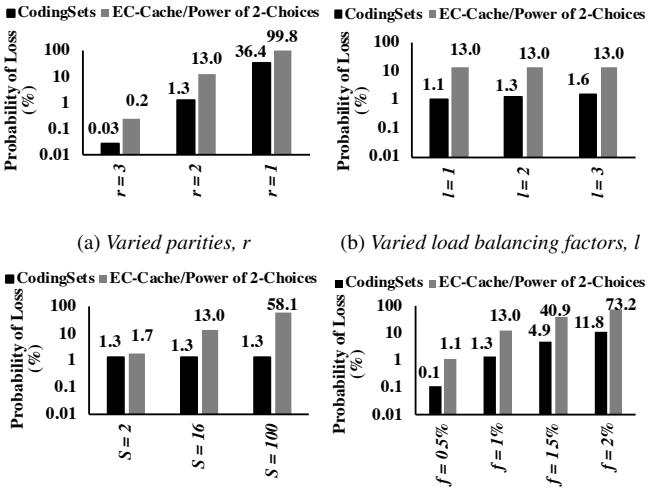


Figure 14: ***Hydra provides transparent completions in the presence of failure. Note that the Y-axis is in log scale.***



(a) *Varied parities, r* (b) *Varied load balancing factors, l*

(c) *Varied slabs per machine, S* (d) *Varied failure rate, f*

Figure 15: ***Probability of data loss at different scenarios (base parameters k=8, r=2, l=2, S=16, f=1%) on a 1000-machine cluster.***

server is low (1 GB slabs, with 16 GB of memory per server).

Even for a small number of slabs per server, Hydra reduces the probability of data loss by *an order of magnitude*. With a large number of slabs per server (e.g., 100) the probability of failure for EC-Cache becomes very high during correlated failure. Figure 15 shows that there is an inherent trade-off between the load-balancing factor (l) and the probability of data loss under correlated failures.

Load Balancing of CodingSets Figure 16 compares the load balancing of the three policies. EC-Cache's random selection of $(k+r)$ nodes causes a higher load imbalance, since some nodes will randomly be overloaded more than others. As a result, CodingSets improves load balancing over EC-Cache scheme by 1.1× even when $l = 0$, since CodingSets'

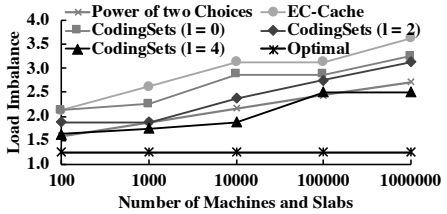


Figure 16: ***CodingSets enhances Hydra with better load balancing across the cluster (base parameters k=8, r=2).***

Latency (ms)		50th			99th		
		SSD	HYD	REP	SSD	HYD	REP
VoltDB	100%	55	60	48	179	173	177
	75%	60	57	48	217	185	225
	50%	78	61	48	305	243	225
ETC	100%	138	119	118	260	245	247
	75%	148	113	120	9912	240	263
	50%	167	117	111	10175	244	259
SYS	100%	145	127	125	249	269	267
	75%	154	119	113	17557	271	321
	50%	124	111	117	22828	452	356

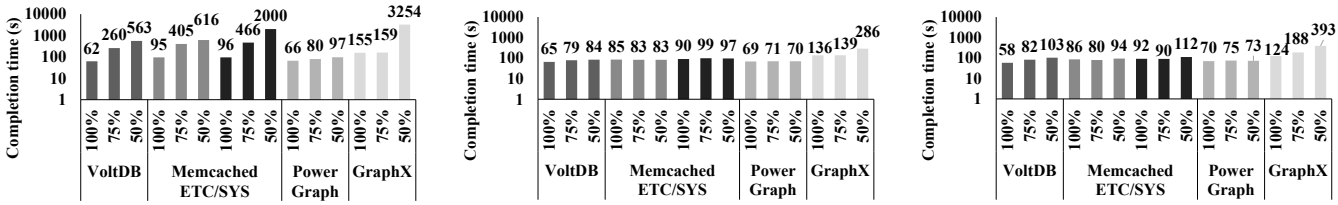
Table 4: ***VoltDB and Memcached (ETC, SYS) latencies for SSD backup, Hydra (HYD) and replication (REP) in cluster setup.***

coding groups are non-overlapping. For $l = 4$, CodingSets provides with 1.5× better load balancing over EC-Cache at 1M machines. The power of two choices improves load balancing by 0%-20% compared CodingSets with $l = 2$, because it has more degrees of freedom in choosing nodes, but suffers from an order of magnitude higher failure rate (Figure 15).

7.2.2 Cluster Deployment

We run 250 containerized applications across 50 machines. For each application and workload, we create a container and randomly distribute it across the cluster. Here, total memory footprint is 2.76 TB; our cluster has 3.20 TB of total memory. Half of the containers use 100% configuration; about 30% use the 75% configuration; and the rest use the 50% configuration. There are at most two simultaneous failures.

Application Performance We compare application performance in terms of completion time (Figure 17) and latency (Table 4) that demonstrate Hydra's performance benefits in the presence of cluster dynamics. Hydra's improvements increase with decreasing local memory ratio. Its throughput improvements w.r.t. SSD backup were up to 4.87× for 75% and up to 20.61× for 50%. Its latency improvements were up to 64.78× for 75% and up to 51.47× for 50%. Hydra's performance benefits are similar to replication (Figure 17c), but with lower memory overhead.



(a) *SSD Backup* (b) *Hydra* (c) *Replication*

Figure 17: ***Median completion times (i.e., throughput) of 250 containers on a 50-machine cluster.***

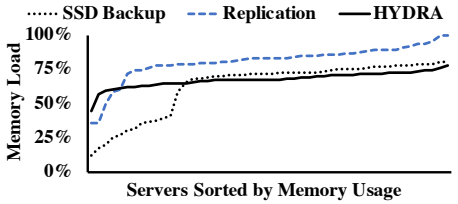
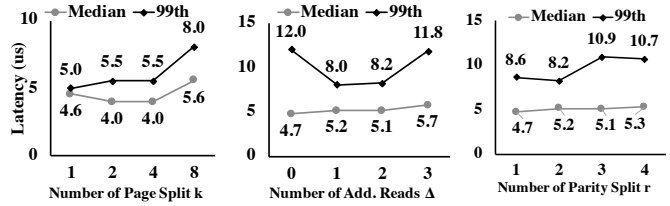


Figure 18: ***Average memory usage across 50 servers.***



(a) *r=4, Δ=1* (b) *k=8, r=4* (c) *k=8, Δ=1*

Figure 19: ***Impact of page splits (k), additional reads (Δ) on read latency, and parity splits (r) on write latency.***

Impact on Memory Imbalance and Stranding Figure 18 shows that Hydra reduces memory usage imbalance w.r.t. coarser-grained memory management systems: in comparison to SSD backup-based (replication-based) systems, memory usage variation decreased from 18.5% (12.9%) to 5.9% and the maximum-to-minimum utilization ratio decreased from 6.92× (2.77×) to 1.74×. Hydra better exploits unused memory in under-utilized machines, increasing the minimum memory utilization of any individual machine by 46%. Hydra incurs about 5% additional total memory usage compared to disk backup, whereas replication incurs 20% overhead.

7.3 Sensitivity Evaluation

Impact of (k, r, Δ) Choices Figure 19a shows read latency characteristics for varying k. Increasing from k=1 to k=2 reduces median latency by parallelizing data transfers. Further increasing k improves space efficiency (measured as $\frac{r}{k+r}$) and load balancing, but latency deteriorates as well.

Figure 19b shows read latency for varying values of Δ. Although just one additional read (from Δ=0 to Δ=1) helps tail latency, more additional reads have diminishing returns; instead, it hurts latency due to proportionally increasing communication overheads. Figure 19c shows write latency variations for different r values. Increasing r does not affect the median write latency. However, the tail latency increases from $r = 3$ due to the increase in overall communication overheads.

Monthly Pricing	Google	Amazon	Microsoft
Standard machine	$1,553	$2,304	$1,572
1% memory	$5.18	$9.21	$5.92
Hydra	6.3%	8.4%	7.3%
Replication	3.3%	4.8%	3.9%
PM Backup	3.5%	7.6%	4.9%

Table 5: ***Revenue model and TCO savings over three years for each machine with 30% unused memory on average.***

Resource Overhead We measure average CPU utilization of Hydra components during remote I/O. Resilience Manager uses event-driven I/O and consumes only 0.001% CPU cycles in each core. Erasure coding causes 0.09% extra CPU usage per core. As Hydra uses one-sided RDMA, remote Resource Monitors do not have CPU overhead in the data path.

In cluster deployment, Hydra increases CPU utilization by 2.2% on average and generates 291 Mbps RDMA traffic per machine, which is only 0.5% of its 56 Gbps bandwidth. Replication has negligible CPU usage but generates more than 1 Gbps traffic per machine.

Background Slab Regeneration To observe the overall latency to regenerate a slab, we manually evict one of the remote slabs. When it is evicted, Resilience Manager places a new slab and provides the evicted slab information to the corresponding Resource Monitor, which takes 54 ms. Then the Resource Monitor randomly selects k out of remaining remote slabs and read the page data, which takes 170 ms for a 1 GB slab. Finally, it decodes the page data to the local memory slab within 50 ms. Therefore, the total regeneration time for a 1 GB size slab is 274 ms, as opposed to taking several minutes to restart a server after failure.

To observe the impact of slab regeneration on disaggregated VMM, we run the micro-benchmark mentioned in §7.1. At the half-way of the application's runtime, we evict one of the remote slabs. Background slab regeneration has a minimal impact on the remote read – remote read latency increases by 1.09×. However, as remote writes to the victim slab halts until it gets regenerated, write latency increases by 1.31×.

7.4 TCO Savings

We limit our TCO analysis only to memory provisioning. The TCO savings of Hydra is the revenue from leveraged unused memory after deducting the TCO of RDMA hardware. We consider capital expenditure (CAPEX) of acquiring RDMA

System	Year	Deployability	Fault Tolerance	Load Balancing	Latency Tolerance
Memory Blade [61]	'09	HW Change	Reprovision	None	None
RamCloud [73]	'10	App. Change	Remote Disks	Power of Choices	None
FaRM [42]	'14	App. Change	Replication	Central Coordinator	None
EC-Cache [76]	'16	App. Change	Erasure Coding	Multiple Coding Groups	Late Binding
Infiniswap [50]	'17	Unmodified	Local Disk	Power of Choices	None
Remote Regions [18]	'18	App. Change	None	Central Manager	None
LegoOS [81]	'18	OS Change	Remote Disk	None	None
Compressed Far Memory [58]	'19	OS Change	None	None	None
Leap [65]	'20	OS Change	None	None	None
Kona [22]	'21	HW Change	Replication	None	None
Hydra		Unmodified	Erasure Coding	CodingSets	Late Binding

Table 6: ***Selected proposals on remote memory in recent years.***

hardware and operational expenditure (OPEX) including their power usage over 3 years. An RDMA adapter costs $600 [10], RDMA switch costs $318 [11] per machine, and the operating cost is $52 over 3 years [50] – overall, the 3-year TCO is $970 for each machine. We consider the standard machine configuration and pricing from Google Cloud Compute [7], Amazon EC2 [2], and Microsoft Azure [2] to build revenue models and calculate the TCO savings for 30% of leveraged memory for each machine (Table 5). For example, in Google, the savings of disaggregation over 3 years using Hydra is (($5.18*30*36)/1.25-$970)/($1553*36)*100% = 6.3%.

7.5 Disaggregation with Persistent Memory Backup

To observe the impact of persistent memory (PM), we run all the micro-benchmarks and real-world applications mentioned earlier over Infiniswap with local PM backup. Unfortunately, at the time of writing, we cannot get hold of a real Intel Optane DC. We emulate PM using DRAM with the latency characteristics mentioned in prior work [34].

Replacing SSD with local PM can significantly improve Infiniswap's performance in a disaggregated cluster. However, for the micro-benchmark mentioned in §7.1, Hydra still provides 1.06× and 1.09× better 99th percentile latency over Infiniswap with PM backup during page-in and page-out, respectively. Even for real-world applications mentioned in §7.1.3, Hydra almost matches the performance of local PM backup – application-level performance varies within 0.94–1.09× of that with PM backup. Note that replacing SSD with PM throughout the cluster does not improve the availability guarantee in the presence of cluster-wide uncertainties. Moreover, while resiliency through unused remote DRAM is free, PM backup costs $11.13/GB [14]. In case of Google, the additional cost of $2671.2 per machine for PM reduces the savings of disaggregation over 3 years from 6.3% to (($5.18*30*36)-$970-$2671.2)/($1553*36)*100% = 3.5% (Table 5).

8 Related Work

Remote-Memory Systems Many software systems tried leveraging remote machines' memory for paging [1,22,26,43,45,50,58,59,64,65,71,79], global virtual memory abstraction [15, 44, 55], and to create distributed data stores [3, 29, 30, 42, 54, 60, 73, 78]. Hardware-based remote access to memory using PCIe interconnects [61] and extended NUMA fabric [72] are also proposed. Table 6 compares a selected few.

Cluster Memory Solutions With the advent of RDMA, there has been a renewed interest in cluster memory solutions. The primary way of leveraging cluster memory is through key-value interfaces [42,52,66,73], distributed shared memory [70,75], or distributed lock [85]. However, these solutions are either limited by their interface or replication overheads. Hydra, on the contrary, is a transparent, memory-efficient, and load-balanced mechanism for resilient remote memory.

Erasure Coding in Storage Erasure coding has been widely employed in RAID systems to achieve space-efficient fault tolerance [80,90]. Recent large-scale clusters leverage erasure coding for storing *cold* data in a space-efficient manner to achieve fault-tolerance [51,69,83]. EC-Cache [76] is an erasure-coded in-memory cache for 1MB or larger objects, but it is highly susceptible to data loss under correlated failures, and its scalability is limited due to communication overhead. In contrast, Hydra achieves resilient erasure-coded remote memory with single-digit μs page access latency.

9 Conclusion

Hydra leverages online erasure coding to achieve single-digit μs latency under failures, while judiciously placing erasure-coded data using CodingSets to improve availability and load balancing. It matches the resilience of replication with 1.6× lower memory overhead and significantly improves latency and throughput of real-world memory-intensive applications over SSD backup-based resilience. Furthermore, CodingSets allows Hydra to reduce the probability of data loss under simultaneous failures by about 10×. Overall, Hydra makes resilient remote memory practical.

Acknowledgments

We thank the anonymous reviewers, our shepherd, Danyang Zhuo, and SymbioticLab members for their insightful comments and feedback that helped improve the paper. This work was supported in part by National Science Foundation grants (CNS-1845853, CNS-2104243) and a gift from VMware.

References

[1] Accelio based network block device. https://github.com/accelio/NBDX.

[2] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing. Accessed: 2019-08-05.

[3] ApsaraDB for POLARDB: A next-generationrelational database - alibaba cloud. https://www.alibabacloud.com/products/apsaradb-for-polardb .

[4] Facebook announces next-generation Open Rack frame. https://engineering.fb.com/2019/03/15/data-center-engineering/open-rack/.

[5] Fio - Flexible I/O Tester. https://github.com/axboe/fio.

[6] Google Cloud Networking Incident 20005. https://status.cloud.google.com/incident/cloud-networking/20005.

[7] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing. Accessed: 2019-08-05.

[8] Intel Intelligent Storage Acceleration Library (Intel ISA-L). https://software.intel.com/en-us/storage/ISA-L.

[9] Intel Rack Scale Design Architecture Overview. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[10] Mellanox InfiniBand Adapter Cards. https://www.mellanoxstore.com/categories/adapters/infiniband-and-vpi-adapter-cards.html.

[11] Mellanox Switches. https://www.mellanoxstore.com/categories/switches/infiniband-and-vpi-switch-systems.html.

[12] Memcached - A distributed memory object caching system. http://memcached.org.

[13] Open Compute Project : Open Rack Charter. https://github.com/facebookarchive/opencompute/blob/master/open_rack/charter/Open_Rack_Charter.pdf.

[14] Pricing of Intel's Optane DC Persistent Memory. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks.

[15] The Versatile SMP (vSMP) Architecture. http://www.scalemp.com/technology/versatile-smp-vsmp-architecture/.

[16] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[17] VoltDB. https://github.com/VoltDB/voltdb.

[18] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *USENIX ATC*, 2018.

[19] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, 2017.

[20] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[21] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[22] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, 2021.

[23] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS, 2019.

[24] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets*, 2017.

[25] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *FAST*, 2014.

[26] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[27] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *USENIX ATC*, 2015.

[28] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *USENIX ATC*, 2013.

[29] Alexandre Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases . In *SIGMOD*, 2017.

[30] Anuj Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs . In *OSDI*, 2016.

[31] Daniel Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems . In *OSDI*, 2010.

[32] Feng Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA . In *SIGMOD*, 2016.

[33] Jeffrey Dean. Evolution and future directions of large-scale storage and computation systems at google . In *SoCC*, 2010.

[34] Joseph Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane DC persistent memory module . *arXiv preprint arXiv:1903.05714*, 2019.

[35] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision . In *DATE*, 2016.

[36] Kimberly Keeton. The machine: An architecture for memory-centric computing . In *ROSS*, 2015.

[37] Krste Asanović. FireBox: A hardware building block for 2020 warehouse-scale computers . In *FAST*. USENIX Association, 2014.

[38] Robert J. Chansler. Data availability and durability with the hadoop distributed file system . *login Usenix Mag.*, 37, 2012.

[39] Wolf Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks . In *VLDB*, 2015.

[40] Yiwen Zhang, J. Gu, Y. Lee, M. Chowdhury, and K. G. Shin. Performance Isolation Anomalies in RDMA . In *KBNets*, 2017.

[41] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[42] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[43] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[44] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[45] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, Mar 1991.

[46] M. D. Flouris and E. P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Journal of Cluster Computing*, 2(4):281–293, 1999.

[47] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[48] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[49] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[50] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[51] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[52] A. K. M. Kaminsky and D. G. Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[53] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[54] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *SOSP*, 2017.

[55] Y. Kuperman, J. Nider, A. Gordon, and D. Tsafrir. Paravirtual Remote I/O. In *ASPLOS*, 2016.

[56] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[57] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with Ingens. In *OSDI*, 2016.

[58] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, 2019.

[59] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[60] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[61] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[62] K. Lim, Y. Turner, J. Chang, J. Santos, and P. Ranganathan. Disaggregated memory benefits for server consolidation. 2011.

[63] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[64] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *USENIX ATC*, 1996.

[65] H. A. Maruf and M. Chowdhury. Effectively Prefetching Remote Memory with Leap. In *USENIX ATC*, 2020.

[66] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[67] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[68] J. C. Mogul and J. Wilkes. Nines are not enough: Meaningful metrics for clouds. In *HotOS*, 2019.

[69] S. Muralidhar, W. Lloyd, S. California, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. Facebook's Warm BLOB Storage System. In *OSDI*, 2014.

[70] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[71] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[72] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *ASPLOS*, 2014.

[73] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[74] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS OSR*, 43(4), 2010.

[75] R. Power and J. Li. Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[76] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI*, 2016.

[77] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[78] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, 2020.

[79] A. Samih, R. Wang, C. Maciocco, T.-Y. C. Tai, R. Duan, J. Duan, and Y. Solihin. Evaluating dynamics and bottlenecks of memory collaboration in cluster systems. In *CCGrid*, 2012.

[80] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *VLDB*, 2013.

[81] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[82] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with Borg. In *EuroSys*, 2015.

[83] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[84] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory kv storage. In *SYSTOR*, 2017.

[85] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with rdma: decentralization without starvation. In *SIGMOD*, 2018.

[86] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *FAST*, 2016.

[87] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo. Understanding the effect of data center resource disaggregation on production dbmss. In *VLDB*, 2020.

[88] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[89] Y. Zhang, Y. Tan, B. Stephens, and M. Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *USENIX NSDI*, 2022.

[90] Z. Zhang, Z. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Technical Report May, Microsoft Research Technical Report MSR-TR-2010-52, May 2010, 2010.