Aceso: Achieving Efficient Fault Tolerance in Memory-Disaggregated Key-Value Stores

Zhisheng Hu[♠], Pengfei Zuo[◇], Yizou Chen[♠], Chao Wang[♠], Junliang Hu[♠], Ming-Chang Yang[♠]
[♠]The Chinese University of Hong Kong, [◇]Huawei Cloud

Abstract

Disaggregated memory (DM) has garnered increasing attention due to high resource utilization. Fault tolerance is critical for key-value (KV) stores on DM since machine failures are common in datacenters. Existing KV stores on DM are generally based on replication to achieve fault tolerance, which however suffer from high memory space and performance overheads. In this paper, we investigate the efficiency of different fault-tolerant mechanisms on DM and reveal that checkpointing and erasure coding work best for the index and KV pairs respectively. Based on these observations, we present Aceso, a DM-based KV store that employs a hybrid fault-tolerant mechanism, combining checkpointing for the index and erasure coding for KV pairs. However, applying this hybrid mechanism to DM introduces multiple challenges, *i.e.*, performance interference and data loss of checkpointing, slow space reclamation and failure recovery of erasure coding. To address these challenges, Aceso leverages a differential checkpointing scheme to reduce performance interference incurred by the bandwidth consumption to transmit checkpoints, a versioning approach to recover lost index updates on failures, a delta-based space reclamation mechanism to reclaim obsolete KV pairs with negligible overhead, and a tiered recovery scheme to minimize user disruption. Our experiments show that Aceso simultaneously achieves up to 2.7× throughput improvement, up to 54% tail latency reduction, and 44% memory space savings compared with the state-of-the-art replication-based KV store on DM.

CCS Concepts: • **Information systems → Distributed storage**.

Keywords: Disaggregated Memory, RDMA, Key-Value Store

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11…$15.00
https://doi.org/10.1145/3694715.3695951

1 Introduction

Disaggregated memory (DM) has attracted extensive attention from both the academic and industrial communities [2, 24, 26, 53, 60, 61, 63]. This architecture decouples the compute and memory resources of monolithic servers into independent compute pool and memory pool interconnected with high-performance networks, *e.g.*, remote direct memory access (RDMA) [68] and compute express link (CXL) [45]. The resources in both pools are independently allocated and scaled on demand, thus having high resource utilization.

In-memory key-value (KV) stores are widely employed in the industry to serve various applications [20, 87]. In recent years, they have been ported into the DM architecture to achieve high resource utilization [3, 36, 62, 63, 69]. Fault tolerance is crucial for in-memory KV stores, as losing data can severely impact performance. For instance, it takes Facebook 2.5-3 hours to rebuild 120 GB of data from disks [22]. Existing KV stores on DM rely on replication-based approaches [5, 34] to achieve fault tolerance. However, these approaches exhibit severe performance and memory space overheads, impeding the application and development of DM-based KV stores. The performance overhead arises from synchronously maintaining multiple index replicas, requiring numerous costly remote access operations to maintain strong consistency among index replicas. The memory space overhead results from the replicas of KV pairs, necessitating at least n times the memory space to tolerate $n-1$ failures.

This paper aims to ensure fault tolerance of KV stores on DM with both high performance and space efficiency. The idea is to leverage a hybrid fault-tolerant mechanism including checkpointing and erasure coding. Checkpointing can be applied to the index to avoid synchronously maintaining multiple index replicas, reduce the consistency overhead and thus improve performance. Erasure coding is a more space-efficient way than replication to achieve data fault tolerance and can be applied to the KV pairs. Nevertheless, it is non-trivial to employ checkpointing and erasure coding in DM-based KV stores due to the following challenges.

1) Performance interference and data loss of checkpointing on DM. Checkpointing for the index necessitates the periodic transmission of checkpoints, which increases bandwidth usage and reduces the performance of KV requests. Besides, when a failure occurs, using checkpoints can only recover the index to a previous state, leading to the loss of recently committed KV pairs and thus compromising the data consistency of the KV store.

2) Slow space reclamation and failure recovery of erasure coding on DM. In replication-based KV stores on DM, an obsolete KV pair can be overwritten directly by a new one via one-sided writes. However, using erasure coding requires frequently invoking weak CPUs in the memory pool to update the parity for each overwritten old KV pair, which dramatically reduces the efficiency of space reclamation. Moreover, with erasure coding, the recovery of lost data necessitates decoding calculation, which increases the time taken for recovery.

To address the above challenges, we propose **Aceso**[1], a DM-based KV store that applies a hybrid fault-tolerant mechanism. First, to reduce the performance interference of checkpointing, Aceso leverages a differential checkpointing scheme to reduce the bandwidth consumption to transmit checkpoints. To recover lost index updates on failures, Aceso adopts a versioning approach. It retrieves the latest checkpoint and reprocesses the most recent KV pairs based on their version information, ensuring the index can be restored to its latest and consistent state. Second, to reduce the space reclamation overhead from erasure coding, Aceso utilizes the linearity property of the erasure code to implement a delta-based space reclamation mechanism to reclaim obsolete KV pairs with negligible overhead. Finally, to mitigate the extended recovery time caused by erasure coding, Aceso leverages a tiered recovery scheme to prioritize the restoration of critical data (*e.g.*, the index) during recovery, ensuring the fast recovery of the KV store functionality.

We implement Aceso and evaluate its effectiveness with micro, YCSB [9], and Twitter's workloads [84]. The results indicate that, compared with the state-of-the-art replication-based KV store [63] on DM, Aceso simultaneously achieves up to 2.7× throughput improvement, up to 54% tail latency reduction and 44% memory space savings. The code of Aceso is open-source[2].

In summary, this paper makes the following contributions:

- Based on a comprehensive analysis of the efficiency of different fault-tolerant mechanisms on DM, we present the insight that checkpointing and erasure coding are optimal for fault tolerance of the index and KV pairs, respectively, and present the corresponding challenges.
- We propose Aceso, a DM-based KV store that applies a hybrid fault-tolerant mechanism, which presents a differential checkpointing scheme with versioning for the index and an offline erasure coding scheme with lightweight space reclamation and fast failure recovery for KV pairs.
- We implement Aceso and conduct a thorough evaluation, demonstrating the effectiveness of our design.

[1]Aceso is the name of a Greek goddess of well-being and the healing process.
[2]https://github.com/dmemsys/Aceso.git

2 Background and Motivation

2.1 Disaggregated Memory and Failure Model

In traditional server clusters, the CPU and memory resources of each server are static, and users can only rent servers based on fixed CPU-memory ratios [24, 60, 94]. When in need of CPU or memory expansion, users are constrained by these fixed ratios, leading to inefficient resource utilization. Reports from Google [67] and Alibaba [46] indicate that server clusters often experience up to 60% of idle and underutilized memory resources. To address this issue, the disaggregated memory (DM) architecture is proposed [43, 44, 53, 61, 86], decoupling the CPU and memory components from traditional monolithic servers to form separate compute pool and memory pool. With this architecture, resources can be flexibly allocated on demand, achieving higher resource utilization.

In the existing DM architecture, nodes are primarily categorized into compute nodes (CNs) and memory nodes (MNs), interconnected with high-performance networks, *e.g.*, CXL and RDMA. In this paper, we consider CNs and MNs to be connected with RDMA, following recent works [36, 48, 63, 69]. RDMA provides APIs known as RDMA verbs [11] for developers, offering both one-sided and two-sided verbs. One-sided operations avoid the involvement of MN CPUs, making them a popular choice [15, 21, 51, 90, 94]. Within one-sided verbs, atomic operations like compare-and-swap (CAS) and fetch-and-add (FAA) have been extensively employed to resolve concurrent conflicts in KV stores on DM [63, 93].

Failure Model. Like existing works [25, 47, 63, 88], this paper considers the fail-stop failure model in which failures of CNs and MNs would result in memory data loss while the network is assumed to be reliable and Byzantine failures are out of scope. Besides, we consider a reliable master that maintains a lease-based membership service to detect node failures, similar to FUSEE [63]. Under this failure model, Aceso can tolerate an arbitrary number of CN crashes and at most two MN crashes (within each *coding group*, §3.1).

2.2 Fault Tolerance Mechanisms

In the data storage field, there are multiple kinds of fault tolerance mechanisms as follows.

Replication. Replication is a common fault tolerance mechanism that enhances data availability and durability by creating data replicas across multiple physical locations [1, 5, 47, 83, 91]. This method offers redundancy as the data is backed up in several places. In case of a node failure, the data can be recovered from other nodes. However, the downside is a higher requirement for storage space due to the replicas.

Erasure Coding. Erasure coding is a more efficient fault tolerance mechanism [28, 40, 59, 85, 88]. Instead of creating full copies of data like replication, erasure coding generates smaller parity data. The parity data allows for the reconstruction of the original data if a portion is lost or corrupted.

This process involves some computational cost but significantly reduces the storage space overhead. Erasure codes can be categorized into two main types based on their computational approaches: GF-based and XOR-based. GF-based erasure codes, such as the Reed-Solomon (RS) code [57], involve Galois Field (GF) multiplication and XOR operations during the encoding and decoding phases, resulting in substantial computational complexity. XOR-based erasure codes, such as X-Code [82], only require XOR operations, making them considerably faster.

Checkpointing. Checkpointing is another fault tolerance mechanism that periodically saves the system state [6, 12, 52, 72, 81]. In the event of a failure, the system can recover from the most recent checkpoint rather than starting over. The advantage of this method is a reduction in data recovery time compared to rebuilding from scratch, but the creation of checkpoints could impact system performance and require additional storage space.

2.3 Existing Fault-tolerant KV Stores on DM

Traditional in-memory KV stores have been ported to the DM architecture to achieve high resource utilization [36, 39, 48, 76, 94]. Similar to other distributed systems, DM-based KV stores are required to exhibit not only high performance but also fault tolerance.

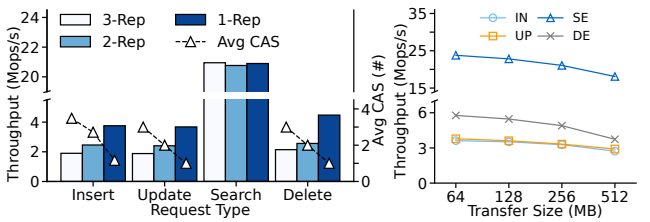
Clover [69] and FUSEE [63] are two state-of-the-art (SOTA) DM-based KV stores with fault tolerance. Clover adopts a partially disaggregated architecture, where metadata management is handled by traditional monolithic servers (*i.e.*, metadata servers), while the KV pairs are stored across multiple MNs. FUSEE adopts a fully disaggregated architecture, where both the metadata (*i.e.*, memory management metadata and index) and KV pairs are stored in MNs, while the clients directly manipulate these data in a decentralized manner via one-sided verbs. FUSEE achieves better performance and scalability than Clover in this fully disaggregated way.

However, both Clover and FUSEE rely on replication to ensure fault tolerance. Regarding the index, any modification to the index is synchronized across all replicas to ensure strong consistency. This synchronization increases the latency for each write request, significantly impairing system performance. Similarly, both Clover and FUSEE store multiple replicas of KV pairs in MNs, leading to substantial space overhead. We analyze the efficiency of replication-based fault tolerance solutions on DM in the next subsection.

2.4 Different Fault-tolerant Mechanisms on DM

In the following, we analyze the space overhead and performance of different fault-tolerant mechanisms on DM-based KV stores.

Replication on DM. We take FUSEE [63] as an example to analyze the overhead of the replication-based approach on DM. The first overhead is the significant **performance overhead**. In FUSEE, with n index replicas maintained, if multiple



(a) Different numbers of replicas. (b) Different checkpoint sizes.

Figure 1. Performance of DM-based KV stores under different numbers of replicas and checkpoint sizes.

clients concurrently update the same key, each client performs CAS operations on $n-1$ backup indexes first. Based on the CAS results, one winner is selected to CAS the remaining backup indexes, ensuring consistent updates. Finally, the winner CASes the primary index to commit its request. In this way, each write request requires at least n CAS operations to modify the index, significantly impairing system performance. Figure 1(a) illustrates the throughputs and the average numbers of CAS operations of four requests in FUSEE (i.e., `INSERT`, `UPDATE`, `SEARCH`, and `DELETE`) on microbenchmarks (§4.2). With the number of index replicas increasing from 1 to 3, the `SEARCH` performance is not affected due to no need for CAS. However, the throughputs of `INSERT`, `UPDATE`, and `DELETE` are significantly degraded by 49.6%, 48.7%, and 52.1% respectively, due to using a large number of CAS operations for maintaining the consistency of index replicas. The second overhead is the evident **space overhead**. Replication imposes increased memory consumption, necessitating at least n times memory space to withstand $n-1$ failures. This contradicts the original intent of the DM, which aims to enhance resource utilization.

Erasure Coding on DM. As introduced in §2.2, erasure coding is an efficient way of ensuring data fault tolerance with much higher storage space efficiency. This inspires us to apply erasure coding in DM-based KV stores to reduce memory space overhead. Previous studies [37, 92] have applied erasure coding in DM-based memory swapping systems. However, the integration of erasure coding into DM-based KV stores remains unexplored.

The out-of-place modification scheme, widely adopted in DM-based KV stores [36, 48, 63, 69] for its efficiency, provides a chance for offline computation of coarse-grained erasure coding for KV pairs. In other words, KV pairs can be initially written and subsequently batch-encoded by MN CPUs in the background [19, 41, 59, 85], thereby avoiding any performance impact on KV requests. On the other hand, the index undergoes frequent in-place updates by clients. If erasure coding were applied to the index, frequent updates to its parity would be required, which would easily saturate the weak compute power of MNs, resulting in significant performance degradation.

Table 1. Space overhead and performance of different fault-tolerant mechanisms on DM.

Metrics	Index		KV Pairs	
	Space	Perf	Space	Perf
Replication	Low	Mid	High	High
Erasure Coding	Low	Low	***Low***	***High***
Checkpointing	***Low***	***High***	High	Low

Checkpointing on DM. RDMA possesses a high bandwidth, but the RNICs have an IOPS bound [48, 64]. Index modifications, being small-sized updates, occupy a large portion of the IOPS bound, consequently leading to underutilization of the RNICs' bandwidth, which is evidenced by the pronounced asymmetry in read-write performance as shown in Figure 1(a). Checkpointing provides an opportunity to harness the redundant bandwidth of the RNICs. By applying checkpointing to the index, clients are no longer required to maintain multiple index replicas synchronously, thus reducing numerous small-sized updates and greatly enhancing performance. However, checkpointing is not suitable for KV pairs, as their large size would cause serious space overhead similar to replication. Moreover, the periodic transmission of KV pairs would consume a large amount of bandwidth, leading to a noticeable performance decline in KV requests.

Summary. Table 1 summarizes the space overhead (*Space*) and performance (*Perf*) of the three fault tolerance mechanisms when applied to the index and KV pairs on DM, respectively. For the index, we find that checkpointing is the most suitable for the index due to its low space overhead and high performance. As the index typically occupies a minor portion of memory space in DM-based KV stores, such as an 8B index slot pointing to a KV pair with hundreds of bytes[3] [36, 48, 63, 69], checkpointing does not cause significant space overhead. Additionally, checkpointing the index effectively reduces the number of small-sized updates compared to replication, thus mitigating the IOPS bound and improving performance. For the KV pairs, we find that erasure coding is the most suitable. Because, given the large size of KV pairs, erasure coding significantly reduces space overhead. Besides, erasure coding on KV pairs can be executed in an offline manner, thereby avoiding any performance degradation. Therefore, we adopt a hybrid fault-tolerant mechanism to ensure the fault tolerance of the KV store on DM, i.e., using checkpointing for the index and erasure coding for KV pairs.

2.5 Challenges

The previous discussion has motivated us to hybridize checkpointing and erasure coding. However, the efficient application of a hybrid fault-tolerant mechanism in the DM-based KV store confronts the following challenges.

[3]Extremely small KV pairs are not the focus of this paper.

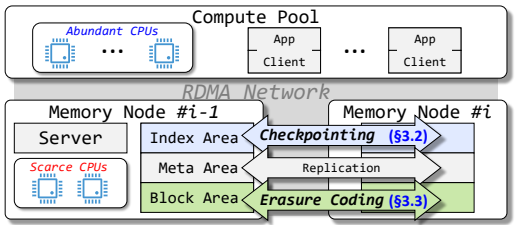


Figure 2. The overview of Aceso.

Challenge 1: ***Performance interference for KV requests due to checkpointing transmission.*** We evaluate the throughputs of four types of requests (**IN**SERT, **UP**DATE, **SE**ARCH, and **DE**LETE) during periodically transmitting different sizes of the index checkpoints in MNs, as shown in Figure 1(b). We observe that with the increase of the checkpoint size, the throughput of KV requests decreases significantly. With the checkpoint size growing to 512MB, the throughput of SEARCH decreases by about 25%. This is because the transmission of the index checkpoints seizes the network bandwidth when executing the KV requests.

Challenge 2: ***Losing index updates during the checkpointing interval.*** When a failure occurs, using checkpoints can only recover the index to a previous state, leading to the loss of recently committed KV pairs and thus compromising the data consistency of the KV store.

Challenge 3: ***Slow space reclamation of erasure coding in DM.*** In replication-based KV stores on DM, the old KV pair can be overwritten directly by a new one via one-sided writes. However, with erasure coding, synchronous updates to the parity are required during the overwriting process to maintain strong consistency. Since one-sided verbs cannot perform complex encoding calculations, updating parity relies on the weak compute power of MN CPUs, which may easily become the performance bottleneck.

Challenge 4: ***Long memory node recovery time with erasure coding.*** In replication-based KV stores on DM, recovering data from a crashed MN only requires reading its replica. However, with erasure coding, the recovery of lost data necessitates decoding calculation, which significantly increases the time taken for recovery.

3 The Aceso Design

3.1 Overview

Figure 2 shows the overview of Aceso, primarily composed of clients and memory nodes (MNs). Clients provide a common API [36, 63, 69] (INSERT, UPDATE, SEARCH, and DELETE) for upper-level applications to access KV pairs. All KV requests from applications are directly executed by clients through one-sided RDMA verbs without involving the CPUs on MNs. The memory space of each MN is divided into three areas: the *Index Area* storing the index, the *Meta Area* storing the metadata, and the *Block Area* storing memory blocks that hold KV pairs and the parity data. Each MN includes a server

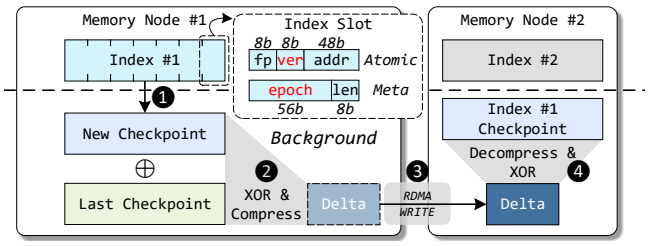


Figure 3. The differential checkpointing workflow.

responsible for coarse-grained management tasks such as space allocation, index checkpointing, and erasure coding. Every group of n MNs forms a *coding group*, working independently to achieve fault tolerance. The dynamic expansion of MNs can be achieved by adding a new *coding group* or enlarging the size of a particular *coding group* combined with data migration, which is not the focus of this paper. The subsequent sections of this paper will focus on the design of a single *coding group*.

To achieve the fault tolerance of the *Index Area*, Aceso leverages differential checkpointing with versioning-based recovery (§3.2) to address the **performance overhead** in current replication-based approaches. To achieve the fault tolerance of the *Block Area*, Aceso leverages offline erasure coding with delta-based space reclamation (§3.3) to reduce the **space overhead**. As for the metadata in the *Meta Area*, due to its small size and infrequent modifications, simple replication via `RDMA_WRITE`s is sufficient. Finally, to deal with the extended recovery time, Aceso adopts a tiered recovery scheme to minimize user disruption (§3.4).

3.2 Differential Checkpointing for Index

Aceso adopts RACE hashing [94] for the index, similar to prior works [62, 63]. It employs an out-of-place modification scheme, where each write request (`INSERT`, `UPDATE`, `DELETE`) creates a new KV pair and atomically updates the index slot to reference it. Aceso ensures strong consistency, with the atomic index update serving as the commit point. For the fault tolerance of the index, Aceso employs checkpointing. However, transmitting checkpoints consumes bandwidth, leading to a decrease in the performance of KV requests (**Challenge 1**). Additionally, the checkpoints do not record the most recently committed KV pairs, failing to satisfy strong consistency (**Challenge 2**). To tackle these challenges, Aceso adopts a differential checkpointing approach to reduce the bandwidth consumption (§3.2.1), and a versioning-based recovery approach to ensure strong consistency (§3.2.2) and efficient index recovery (§3.2.3).

3.2.1 Differential Checkpointing. To reduce the bandwidth consumed by transmitting checkpoints, Aceso employs a differential checkpointing approach [17, 32], where only the differences (*i.e.*, the delta) between consecutive checkpoints are transferred. Since the amount of index data that clients can modify in a given timeframe is limited by the maximum IOPS, this approach significantly reduces bandwidth consumption.

Figure 3 shows the workflow of index checkpointing in Aceso, which can be divided into four main steps: generating a new index checkpoint (❶), performing XOR between the new checkpoint and the previous one (`Last Checkpoint`) and compressing the resulting delta (❷), writing the compressed delta to the neighboring MN via `RDMA_WRITE` (❸), and finally, the neighboring MN decompresses the delta and XORs it with the previous checkpoint to update it (❹). Aceso currently employs the LZ4 compression algorithm due to its high performance, but other compression algorithms can also be applied, which is orthogonal to our work.

Each MN's server periodically executes the above workflow in the background to propagate its local index to neighboring MNs, which is synchronized by a leading server periodically sending trigger signals to other servers via RPC. The index contents can be modified concurrently while generating the new checkpoint. Because the server CPU only reads the index during checkpointing, while clients modify the index through atomic verbs (*e.g.*, `RDMA_CAS`), which are then transformed into PCIe read-modify-write (RMW) transactions [33] to ensure atomicity, the generated index checkpoint will not have partially modified index slots.

3.2.2 Slot Versioning. After the loss of an MN's index, the recovery process first reads the latest index checkpoint. However, the index checkpoint merely represents the state of the index at a previous moment, with many recent modifications yet to be applied. Therefore, Aceso will scan all recently created KV pairs to reapply them to the recovered index, ensuring each index slot points to the latest KV pair. To achieve this, Aceso assigns a *Slot Version* to each index slot, which is increased by 1 each time the slot is updated. The updated *Slot Version* is also recorded in the new KV pair. Consequently, the KV pairs corresponding to a slot are ordered based on increasing *Slot Versions*, from old to new. Therefore, repairing the index slot requires only retaining the KV pair with the highest *Slot Version*.

Index Slot Structure. To accommodate the *Slot Version*, Aceso makes a minor expansion to the structure of the index slot, as shown in Figure 3. Aceso extends the 8B index slot from RACE Hashing [94] to 16B. The first 8B *Atomic* field is modified atomically by each write request via `RDMA_CAS`. The remaining 8B *Meta* field stores infrequently changing information. The *Atomic* field contains an 8-bit *fingerprint* (*fp*) for storing the 8-bit hash of the key to accelerate key searching and a 48-bit *address* (*addr*) representing the global address of the KV pair in the memory pool. The *Meta* field contains an 8-bit *length* (*len*) representing the size of the KV pair in units of 64B. The *length* is positioned in *Meta* as we anticipate the KV pair size of the same key will not change frequently. Besides, In Aceso, each KV pair's header records the KV length. If a client detects a mismatch between the

Algorithm 1 Slot versioning's algorithm

```
1:  procedure UPDATE(KV_new)
2:      Atom_old, Meta_old ← QUERY_INDEX(KV_new.key)
3:      Atom_new, Meta_new ← Atom_old, Meta_old
4:      Atom_new.ver ← Atom_old.ver + 1
5:      if Meta_old.epoch ∧1 = 1 then
6:          return          ▷ Meta is locked by another client, retry
7:      if Atom_old.ver = 0xFF then
8:          Meta_new.epoch ← Meta_old.epoch + 1
9:          cas ← CAS_INDEX(Meta_old, Meta_new)    ▷ Lock Meta
10:         if cas = FAIL then
11:             return      ▷ Meta is locked by another client, retry
12:         Meta_old        ← Meta_new
13:         Meta_new.epoch ← Meta_new.epoch + 1
14:     KV_new.ver, KV_new.epoch ← Atom_new.ver, Meta_new.epoch
15:     Atom_new.addr ← WRITE_KV(KV_new)
16:     cas ← CAS_INDEX(Atom_old, Atom_new)        ▷ Commit KV
17:     if cas = FAIL then
18:         WRITE_VER(Atom_new.addr, −1)           ▷ Invalidate KV
19:     if Atom_old.ver = 0xFF then
20:         CAS_INDEX(Meta_old, Meta_new)          ▷ Unock Meta
```

KV pair size and the *length* in the index slot, it updates the *length* with a single RDMA_WRITE.

Slot Version. Compared to the original index slot in RACE Hashing, Aceso adds an 8-bit *version* (*ver*) in the *Atomic* field and a 56-bit *epoch* in the *Meta*. These two values form a logical 64-bit *Slot Version*, where the lower 8 bits are the *version*, and the upper 56 bits are the *epoch*. The *Slot Version* exists not only in the index slot but also in the header of each KV pair. Algorithm 1 exemplifies how Aceso maintains these *Slot Versions* through processing UPDATE requests. Each time a client modifies the *Atomic* field of a slot via RDMA_CAS, the 8-bit *version* is incremented by 1 (Line 4). When the *version* rolls over from 255 (0xFF) to 0, the client performs extra operations to modify the *epoch*. First, the client tries to lock the *Meta* field by incrementing the *epoch* by 1 via RDMA_CAS, marking it as odd to indicate the lock state (Line 9). Then, the client proceeds with the CAS operation on the *Atomic* field to commit the KV pair as usual (Line 16). Upon completion, the client increments the *epoch* by 1 to mark it as even to unlock (Line 20). In this manner, each update to the *Meta* occurs after every 256 updates to the *Atomic*, a frequency which causes negligible impact on the performance of KV requests. Under normal circumstances where the *Meta* field is not locked, the client's direct attempt to CAS the *Atomic* field may fail. In such cases, it sets the *Slot Version* of its newly written KV pair to -1, marking it as invalid (Line 18). In this way, Aceso ensures no two KV pairs of one slot have the same *Slot Version*. It also simplifies Aceso's index recovery process as it only needs selecting the KV pair with the highest *Slot Version* for each index slot, ensuring that the most up-to-date KV pair is inserted into the corresponding slot.

Remarks. ***(1) The numeric range.*** Assigning a 64-bit *Slot Version* to each slot is sufficient. Even under a 100% UPDATE workload on a single key (*e.g.*, 4MOps/s), it would take at least hundreds of years to exhaust the numeric range. ***(2) The client failure handling.*** In the event of a client failure after locking the *Meta* field, other clients updating the same key will retry after a timeout (*e.g.*, 500us). They increment the current *epoch* by 2 (to the next odd number) and attempt to CAS the *Meta* to re-lock. This procedure ensures client failure will not cause Aceso to be blocked. ***(3) The consistency in recovery.*** During concurrent updates to the same index slot, there may temporarily exist multiple KV pairs with the same incremented *Slot Version*. If a failure occurs at this point and then the index slot needs to be recovered, it is possible to select a different KV pair (not the one with a successful commit). However, strong consistency is still maintained. This is because the KV requests for previously failed commits have not yet returned to the upper applications. Therefore, even if these requests are reprocessed, it does not violate linearizability [27].

3.2.3 Index Versioning.

To ensure each index slot eventually points to the KV pair with the highest *Slot Version* after index recovery, it is crucial to reapply all KV pairs without omission. A naive approach is scanning all KV pairs in the memory pool, which would significantly increase index recovery time. To address this, in addition to the *Slot Version*, each MN in Aceso stores a 64-bit *Index Version* at the end of the index. We now explain how the *Index Version* operates.

First, after each round of checkpointing, the *Index Version* of the index is incremented by 1. During the period when the *Index Version* of the index is i, the *Index Version* in the checkpoint is $i-1$. Second, each client manages its own coarse-grained (*e.g.*, 2MB) memory blocks, which are allocated from MNs when its space is insufficient. Besides, Aceso employs an out-of-place modification mechanism where each write request will append a new KV pair to a block. Each memory block has a metadata record in the MN's *Meta Area* (Figure 2). When a memory block is filled, the client notifies the corresponding server to write the current *Index Version* to the metadata record of that memory block. This ensures that each filled memory block is associated with the *Index Version* at the moment of filling, while unfilled memory blocks have an *Index Version* of 0. Consequently, during index recovery from the checkpoint, memory blocks with *Index Versions* lower than the index checkpoint can be skipped, as the KV pairs in these blocks are already applied. This approach effectively narrows the range of KV pairs that need to be scanned, thereby speeding up index recovery.

Summary. To integrate the concepts presented so far, an example process of index recovery is illustrated in Figure 4. There are three MNs. MN 1 stores an index and two memory blocks. During the period when the *Index Version* is 3, a slot is updated by the client via CAS, increasing its *Slot Version*

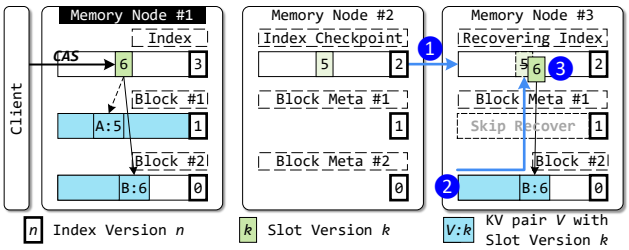


Figure 4. The index recovery process.

from 5 to 6, and the pointer in the slot shifts from KV pair A to KV pair B. MN 2 holds MN 1's index checkpoint and the metadata replica. However, the checkpoint's *Index Version* remains at 2, with its slot still pointing to KV pair A (*Slot Version* 5). Now MN 1 encounters a failure, and MN 3 is tasked with recovery. It first reads the metadata replica and the index checkpoint from MN 2 (❶). Block 1's *Index Version* is lower than the checkpoint's, thus it is skipped. Block 2, with an *Index Version* of 0 indicating it's among the newest blocks, is recovered via Aceso's erasure decoding mechanism (§3.3), followed by scanning KV pairs within it (❷). KV pair B, with a *Slot Version* of 6, surpasses the checkpoint's and is thus reinserted into the index slot (❸). Locating a KV pair's index slot follows a similar process as normal KV request handling, involving a key-based lookup (*e.g.*, hashing). If a KV pair is found not to belong to the index of the crashed MN, it is skipped. Through this process, Aceso accurately recovers each index slot and skips old memory blocks to ensure the fastest possible recovery of KV store functionality (§3.4.1).

3.3 Offline Erasure Coding for KV Pairs

Aceso employs erasure coding for KV pairs to minimize the space overhead. To perform erasure coding efficiently, Aceso adopts an XOR-based erasure code and designs specialized metadata structures (§3.3.1). To prevent erasure coding from impacting the performance of KV requests, Aceso employs an offline mechanism to remove erasure coding from the critical path of KV request execution (§3.3.2). However, this still poses a challenge in reclaiming space occupied by obsolete KV pairs, as synchronous updates to the parity data are needed (**Challenge 3**). To address this, Aceso adopts a delta-based mechanism, leveraging the linearity property of the erasure code (§3.3.3).

3.3.1 Memory Layout. Here, we introduce the memory layout in Aceso that involves erasure coding. First, the *Block Area* is divided into memory blocks at a fixed granularity (*e.g.*, 2MB) and erasure coding is then performed on these coarse-grained memory blocks, enabling efficient computation. To keep track of the status involved in erasure coding, the *Meta Area* maintains a metadata record for each memory block.

X-Code. In the *Block Area*, Aceso utilizes an XOR-based erasure code, X-Code [82], for its superior performance over the GF-based code, as previously discussed and demonstrated in our experiments (§4.5). Due to space constraints, we do

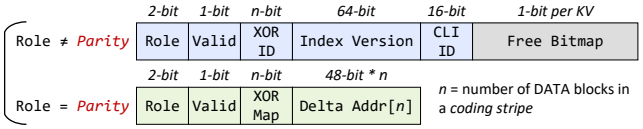


Figure 5. The structure of the metadata record.

not display the complete encoding and decoding processes of X-Code, but its basic operation is simply to generate parity data by XORing multiple blocks together. We refer to these blocks involved in XOR as DATA blocks, and the resulting parity data as the PARITY block. A PARITY block, together with these DATA blocks, forms a *coding stripe*. Multiple *coding stripes* are interleaved within a single *coding group* to attain a certain degree of load balance. Specifically, each block of a *coding stripe* is distributed across different MNs, with each MN in a *coding group* storing both PARITY blocks and DATA blocks. In a *coding group*, X-Code can tolerate up to two node failures. This level of fault tolerance, equivalent to that of three-way replication, is deemed sufficient in our study. Other erasure codes that satisfy the linearity (§3.3.2) can also be incorporated into Aceso depending on the required levels of fault tolerance and performance.

Memory Block Metadata. In the *Meta Area*, the structure of a metadata record is shown in Figure 5. The 2-bit *Role* indicates the type of the memory block. In Aceso, memory blocks are categorized into four types: FREE, DATA, PARITY, and DELTA. FREE represents unallocated blocks, DATA blocks store KV pairs, PARITY blocks store the parity data that are generated by multiple DATA blocks via erasure coding, and DELTA blocks store the delta for DATA blocks in the offline erasure coding (§3.3.2) and the space reclamation (§3.3.3) mechanisms. The 1-bit *Valid* indicates the validity of the block's data, as it may temporarily be unavailable due to failures. ***For non-*PARITY *blocks,*** the *n*-bit *XOR ID* represents its sequential location in its *coding group*. The 64-bit *Index Version* are copied from the index when the block is filled (§3.2.3). The 16-bit *CLI ID* records the ID of the client to which the block is allocated. The *Free Bitmap* keeps track of the validity statuses of KV pairs, which is used in Aceso's space reclamation mechanism (§3.3.3). KV pairs within a memory block have the same size, and memory blocks are grouped into different size classes to accommodate variable-length KV pairs, similar to slab allocators [4, 18]. ***For* PARITY *blocks,*** the *n*-bit *XOR Map* records, for each specific bit, whether the corresponding DATA block of the *coding group* has undergone erasure coding. The *n*×48-bit *Delta Addr* stores the address of each DATA block's corresponding DELTA block.

3.3.2 Offline Erasure Coding. To avoid performance degradation of KV requests, Aceso adopts an offline erasure coding mechanism, where KV pairs are initially written into MNs and subsequently batch-encoded by MN CPUs in the background. Figure 6 exemplifies this process. Block 2, 3, and 4 belong to the same *coding stripe*, with Block 2 being a

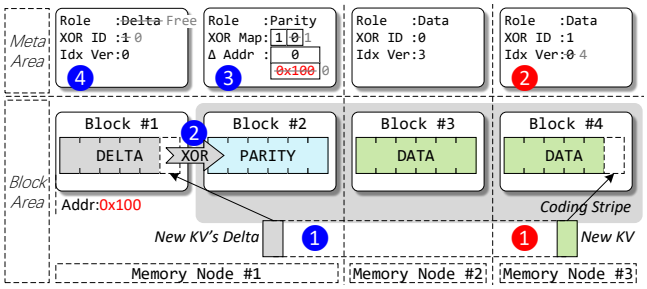


Figure 6. The offline erasure coding process.

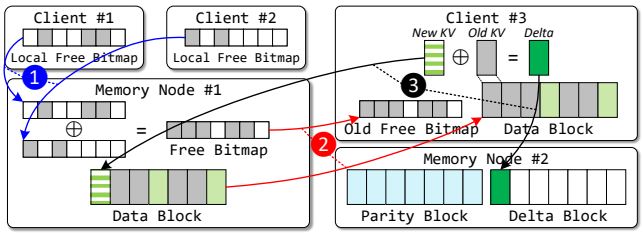


Figure 7. The space reclamation process.

PARITY block, Block 3 being the first DATA block (*XOR ID* 0), and Block 4 being the second DATA block (*XOR ID* 1). Block 1 is a DELTA block, serving as a temporary placeholder on MN 1 for storing the delta (§3.3.3) of the unfilled Block 4.

The Encoding Process. When the client processes a write request (INSERT, UPDATE, DELETE), it writes the KV pair to the DATA block (Block 4) and the delta to the DELTA block (Block 1) via RDMA_WRITEs (❶). The delta is generated either by XORing the old KV pair with the new one for overwrites (§3.3.3), or it is simply the new KV pair itself for non-overwrites. If the client encounters a failure during writes, it will perform recovery upon restart to maintain consistency (§3.4.2). When the DATA block reaches its capacity, the client will notify the servers (MN 1 and 3) via RPC. The server holding the DATA block (MN 3) records the current *Index Version* (*e.g.*, 4) into the metadata record of the DATA block. The server holding the PARITY block (MN 1) first encodes the DELTA block into the PARITY block via erasure coding (❷), and then it sets the corresponding bit in the *XOR Map* of the PARITY block to 1 and clear the address in the *Delta Addr* (△ *Addr*) (❸), before physically freeing the DELTA block (❹). Note that during the above process, if one MN encounters a failure, the other alive MNs will not halt but continue executing the left steps to ensure consistency.

The Decoding Process. When an MN experiences a failure, the lost DATA block is reconstructed via erasure decoding. The server tasked with recovery first locates the PARITY block based on the layout of *coding stripe*s, then retrieves its metadata record. Using the *XOR Map* and *Delta Addr* information therein, it identifies all remaining DATA blocks and DELTA blocks that should participate in the decoding process. For example, in Figure 6, suppose MN 3 fails before Block 4 is filled, the server should read Block 1, 2, and 3 together via RDMA_READ. Then, it encodes the DELTA block (Block 1) into the PARITY block (Block 2) to update it. Subsequently, it reconstructs the lost DATA block using the updated PARITY block and the DATA block (Block 3) through erasure decoding. Note that the XOR-based erasure code (*i.e.*, X-Code) simplifies the recovery of a lost DATA block with just one XOR operation involving all DATA, DELTA, and PARITY blocks. However, if a GF-based erasure code (*e.g.*, RS code) were used, recovery becomes a two-step process where the DELTA blocks must first be encoded into the PARITY blocks.

3.3.3 Delta-based Space Reclamation.

As described in §2.5, erasure coding poses a greater challenge in reclaiming obsolete KV pairs. To address this, Aceso employs a delta-based space reclamation mechanism. This mechanism leverages the linearity of the erasure code [8, 42], which means updates to PARITY blocks can be achieved by adding (*i.e.*, XORing) the delta of the altered DATA blocks, rather than rebuilding the PARITY blocks. Therefore, when overwriting old KV pairs, clients only need to write the delta to the MNs containing the PARITY blocks. The delta is then encoded into the PARITY blocks following the same steps as in the offline encoding mechanism (§3.3.2), enabling efficient updates.

Figure 7 exemplifies this process, which consists of three steps. First, clients maintain the *Free Bitmap* of each DATA block (❶). The bitmap indicates the validity of each KV pair within the block, with all bits initially set to 0. When a client updates a key, the corresponding bit of the old KV pair will be set to 1, denoting expiration. Clients collect the free bitmap information that needs updating and periodically send it to the servers via RPC for bulk updates. Second, when the server detects that the obsolete KV pairs in a DATA block exceed a certain threshold (*e.g.*, 75%), and the free space of the MN is below a certain threshold (*e.g.*, 25%), it triggers the reclamation process for that block (❷). Upon a client's next DATA block allocation request, the old block is reassigned and marked as reused. The server first creates a local copy of the old DATA block as a backup in case of the client failure (§3.4.2). It then sends the old *Free Bitmap* to the client and resets the *Free Bitmap* and the *Index Version* in the metadata record, as this block reverts to an unfilled state and these obsolete KV pairs will be overwritten by new ones. The client then reads the entire reused DATA block into its local memory. Subsequently, when the client overwrites an obsolete KV pair based on the old *Free Bitmap*, it calculates the delta by XORing the old and new KV pairs (❸). Then, it writes the new KV pair to the DATA block and the delta to the DELTA block. Once the reused DATA block refills, the local copy of the old DATA block is freed, and the DELTA block is encoded into the PARITY block, leveraging the erasure code's linearity.

In this manner, Aceso achieves overwriting of old KV pairs and updating of the parity data with the additional overhead of reading the entire reused DATA block. As shown in Figure 1(a), the main bottleneck for write requests is the IOPS bound rather than bandwidth. Therefore, the impact of

this additional read is minimal. As evidenced in §4.4, under a workload of 100% UPDATEs, the performance impact of space reclamation does not exceed 5%.

3.4 Failure Handling

As mentioned in §2.1, Aceso ensures fault tolerance against both CN and MN fail-stop failures that cause memory data loss, while Byzantine failures are not considered. In this framework, Aceso is designed to handle an arbitrary number of CN crashes and up to two MN crashes within each *coding group*. Similar to prior works [36, 63], Aceso assumes a master node to manage failures. In regular operations, it operates a membership service [25], allowing it to swiftly receive notifications of node failures. The fault tolerance of the master node (*e.g.*, state machine replication [15, 31, 71]) is out of scope.

To prevent prolonged recovery time from erasure coding impacting user experience in the event of MN failures (**Challenge 4**), Aceso leverages a tiered recovery scheme to prioritize the restoration of critical data (*e.g.*, the index) for the fastest possible recovery of KV store functionality (§3.4.1). For other types of failures, Aceso also ensures effective recovery (§3.4.2, §3.4.3).

3.4.1 Memory Node Crashes. In the event of MN failures, the master first disseminates this information to all clients. Clients currently executing SEARCH requests on the affected MNs will be interrupted. Clients executing INSERT, UPDATE, or DELETE requests will bypass any steps involving failed MNs, but will proceed with normal operations on the functional MNs until the current requests are completed.

When an MN encounters a failure, the master initiates the recovery process by starting a new server on an idle MN. The MN recovery adopts a tiered approach, first the *Meta Area*, then the *Index Area*, and finally the *Block Area*. Once the *Index Area* is recovered, most of the functionality can be restored, thus minimizing user disruption. ***(1) For the Meta Area,*** which is protected via replication, the new server directly reads from the neighboring MN to recover it. ***(2) For the Index Area,*** the server first reads the index checkpoint from the neighboring MN to recover the index to an older version. Then, it reads or recovers the recent DATA blocks (with an *Index Version* of 0 or greater than or equal to the checkpoint's) through the decoding mechanism introduced in §3.3.2. Finally, the server scans these blocks' KV pairs, updating each index slot to point to the KV pair with the highest *Slot Version*. ***(3) For the Block Area,*** only DATA blocks with an *Index Version* less than that of the checkpoint remain unrecovered. Similar to the previous steps, these blocks are recovered via erasure decoding. Note that PARITY blocks will be gradually recovered in the background after the MN recovery, as they are not critical. DELTA blocks will not be recovered, as they will be encoded into the PARITY blocks during their recovery.

Fast Functionality Recovery and Degraded Search. When an MN crashes, KV requests to the affected index range are blocked. Once the *Index Area* is recovered, write requests (INSERT, UPDATE, DELETE) recover to normal performance, while the read requests (SEARCH) recover to degraded performance (*i.e.*, 53% of normal throughput, §4.4), as clients may need to recover the lost KV pairs on demand via erasure decoding. After the *Block Area* is recovered, read performance returns to normal. In this way, upon completion of the *Index Area* recovery, MN can restore most of its functionality. Given the rapid recovery time of the *Index Area*, the impact of MN failure on user experience is minimal (a delay of 1-2 seconds is generally acceptable [54, 55]).

Remarks. ***(1) The Two-stage Pipelining.*** During the recovery of lost blocks, Aceso overlaps RDMA reads and decoding computations to accelerate the recovery in a two-stage pipeline manner. ***(2) The Two-MN Failure.*** In X-Code [82], when two MNs in the same *coding group* fail concurrently, recovering block A might require block B from another failed node. Leveraging XOR's additive property, the server just reads all required blocks for the recovery of both A and B, then executes a single XOR operation to get block A.

3.4.2 Compute Node Crashes. When CNs fail, the affected clients will restart on other functional CNs. These clients need to recover their management of memory blocks and recover data consistency. Specifically, during recovery, these clients query the servers via RPC to retrieve the addresses of previously allocated blocks based on the *CLI ID* field in the block metadata records. They then read the unfilled DATA and DELTA blocks (with an *Index Version* of 0). By iterating through each KV pair slot in these blocks, the clients determine whether it has been written to, thereby restoring fine-grained memory management and preventing memory leaks.

Each failed client's last KV pair may be in an intermediate state, such as having modified the DATA block without updating the corresponding DELTA block, causing data inconsistency. To address this, Aceso embeds a pair of 2-bit *Write Version*s at the beginning and end of each KV pair and its delta, set to '01' or '10' during writes; when overwriting an old KV pair, the two versions toggle (*e.g.*, from '01' to '10'). During the aforementioned iteration process, the client also checks the *Write Version*s of each KV pair and its delta. If their *Write Version*s are non-zero and identical, it confirms the KV pair is fully written and consistent (since RDMA writes are sequential [76, 93]). If not, the delta is cleared to 0, and the KV pair is restored to 0 or the old KV pair (if overwritten) based on the local copy of the old DATA block (§3.3.3).

3.4.3 Mixed Crashes. When both MNs and CNs fail in quick succession, Aceso first restarts all failed clients to recover data consistency in parallel among the unfilled blocks in the surviving MNs, and then it initiates the recovery processes for the crashed MNs.

3.5 Optimization and Discussion

3.5.1 Local Index Cache. In Aceso, the original 8B index slot is extended to 16B, which leads to read amplification issues. Specifically, querying the RACE Hashing index [94] requires reading multiple index slots, resulting in additional data reads per query. To mitigate this, similar to prior works [39, 48, 63], each CN in Aceso maintains an index cache in its local memory. However, Aceso caches not only the address of the KV pair (*i.e.*, the value of the index slot) but also the address of the index slot. During a cache-accelerated read operation, Aceso reads both the KV pair and the current index slot based on the two addresses. If the current index slot remains unchanged, the read KV pair is valid, as in Aceso, modifying the index slot serves as the commit point. Otherwise, it reads the new KV pair based on the new index slot. Only if the index slot's address changes (*e.g.*, due to index resizing) does Aceso query the index. This approach mitigates the read amplification issues, while significantly improving performance compared to the previous caching strategy that only caches the address of the KV pair.

3.5.2 Other Optimizations. Similar to prior works [29, 63, 77, 78, 80, 93], Aceso incorporates numerous general optimization techniques to achieve its peak performance. During MNs' memory region registration, address alignment and huge page technologies are employed to enhance memory access speed. The RPC between the clients and servers is implemented via RDMA unreliable datagram (UD) QPs [30]. Coroutines are enabled in clients to hide RDMA polling overhead. Additionally, several RDMA-related optimizations are applied in CNs, such as using inline writes for small write requests, employing doorbell batching to reduce PCIe overhead in transmitting RDMA requests, and implementing selective signaling to reduce completion queue polling overhead.

3.5.3 Generality of Techniques in Aceso. Some techniques in Aceso can also be applied to other DM-based KV stores. Specifically, the offline erasure coding mechanism can be adopted in KV stores using an out-of-place modification scheme to reduce space overhead. The strategy that caches both the addresses and values of index slots can be adopted to reduce the frequency of querying the index from the root. The differential checkpointing mechanism can be applied to other indexes utilizing 8B slots (the granularity of `RDMA_CAS`) to ensure fault tolerance with appropriate modifications.

3.5.4 When It Comes to CXL. Since CXL [45] is still evolving, particularly with features such as atomic instructions in CXL 3.0, this paper focuses on RDMA-based DM. However, the techniques discussed in Aceso can serve as a reference for future KV stores on CXL-based DM. For example, the data can be distributed across different CXL memory devices (with different fault domains), and these CXL memory devices can be empowered with additional compute power by leveraging physically-nearby CXL accelerators (*i.e.*, CXL type-1 devices) to perform erasure coding.

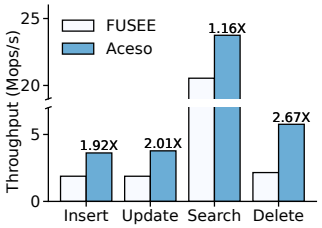


Figure 8. Throughputs on microbenchmarks.

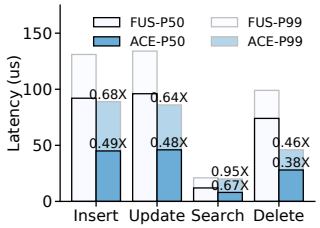


Figure 9. Latencies on microbenchmarks.

4 Evaluation

4.1 Experimental Setup

Testbed. We run all experiments within 28 physical machines (23 CNs and 5 MNs) on the Apt cluster of CloudLab [16]. Each machine, including the MN and CN, is equipped with a 56Gbps Mellanox ConnectX-3 IB RNIC connected to a 56Gbps Mellanox SX6036G switch. The MN machine has two 8-core Intel E5-2650v2 CPUs and 64 GB DRAM, while the CN machine has an 8-core Xeon E5-2450 CPU and 16GB DRAM. We assign 4 CPU cores per MN, one for RPC serving, one for erasure coding, one for checkpoint sending, and one for checkpoint receiving. Given that MN CPUs handle only coarse-grained tasks with low CPU usage, and their count is independent of the number of clients, we consider this configuration reasonable.

Workloads. We use both microbenchmarks and macrobenchmarks to evaluate the effectiveness of Aceso. In the microbenchmarks, keys across different clients are unique, ensuring no concurrent conflicts. For macrobenchmarks, we use YCSB synthetic workloads [9] and real-world workloads (*i.e.*, Twitter's key-value traces [84]). In the YCSB workloads, we use four core workloads labeled as A (50% `SEARCH`, 50% `UPDATE`), B (95% `SEARCH`, 5% `UPDATE`), C (100% `SEARCH`), and D (95% `SEARCH`, 5% `INSERT`). Each core workload consists of one million keys distributed according to the default Zipfian distribution (*i.e.*, $\theta = 0.99$). In the Twitter workloads, we use three traces collected from a compute cluster (*i.e.*, `COMPUTE`), a transient cluster (*i.e.*, `TRANSIENT`), and a storage cluster (*i.e.*, `STORAGE`).

Baseline. We compare Aceso with FUSEE [63], a state-of-the-art fault-tolerant KV store on DM. FUSEE ensures fault tolerance through replication. Each KV write request (`INSERT`, `UPDATE` and `DELETE`) creates n (the replication factor) KV pairs and requires at least n CAS operations to modify the index. FUSEE also includes client-side caches, which we enable with default parameters. We don't compare with Clover [69], because its partially disaggregated architecture limits its performance, making a fair comparison difficult. Additionally, we exclude DINOMO [36] as its design does not consider MN failures, and MicroEC [40] as it only benefits large objects.

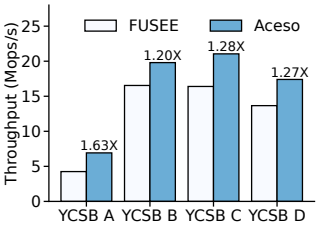


Figure 10. Throughputs on YCSB workloads.

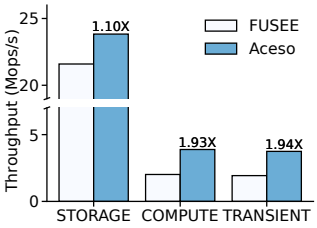


Figure 11. Throughputs on Twitter workloads.

Setup. In our experiments, unless otherwise specified, the number of clients is 184, evenly distributed across 23 CNs, and the KV size is 1024B, which closely aligns with real-world [7, 9, 13]. The total memory pool size is 240GB, distributed evenly among 5 MNs, and the memory block size is 2MB. In Aceso, the *coding group* size is 5 and the index checkpointing interval is 500ms. In FUSEE, the replication factor is set to 3, providing the same fault tolerance level as Aceso to ensure fairness.

4.2 Microbenchmarks

In this subsection, we employ microbenchmarks to evaluate the latencies and throughputs of different KV requests (INSERT, UPDATE, SEARCH, DELETE) across Aceso and FUSEE.

Throughput. Figure 8 shows the throughputs of different KV requests for the two KV stores, where the numbers above Aceso's bars indicate coefficients normalized to FUSEE's. The significant improvement in write requests (INSERT, UPDATE, DELETE) is attributed to Aceso's index checkpointing mechanism, which avoids the overhead of synchronously maintaining index replicas. Among these, DELETE requests show the most pronounced improvement, reaching up to 2.67×, due to the zero-length value of the written KV pairs used solely for logging, thus making them particularly responsive to optimizations aimed at reducing redundant I/O operations. On the other hand, the modest improvement in SEARCH requests by Aceso is due to its use of index slot addresses in the local index cache, which helps avoid unnecessary index queries.

Latency. Figure 9 shows the P50/P99 latencies of different KV requests. Aceso achieves significant reductions in latencies compared to FUSEE, *i.e.*, up to 62% in P50 latencies and 54% in P99. This improvement stems from Aceso's ability to address concurrent conflicts with just one CAS operation, whereas FUSEE, with its three-replicated index, requires at least three CAS operations.

4.3 Macrobenchmarks

YCSB Throughput. Figure 10 shows the throughputs of the two KV stores on four YCSB workloads. For read-intensive workloads (*i.e.*, YCSB B, C, D), Aceso, with its local index cache using both values and addresses of index slots, outperforms FUSEE (up to 1.28×) that only utilizes values in its local index cache. For write-intensive workloads (*i.e.*, YCSB A, 50% UPDATE), Aceso significantly outperforms (1.63×) as

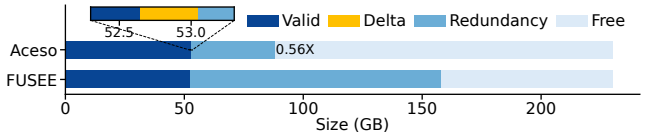


Figure 12. Memory distribution.

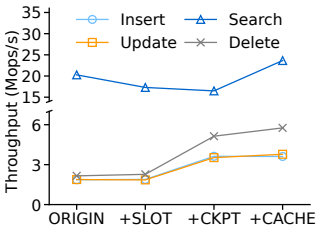


Figure 13. Factor analysis.

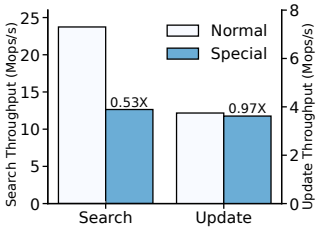


Figure 14. Throughputs of degraded SEARCH and UPDATE with space reclamation.

the index checkpointing mechanism greatly enhances the efficiency of resolving concurrent conflicts.

Twitter Throughput. Figure 11 shows the throughputs of the two KV stores on different Twitter workloads (*i.e.*, STORAGE, COMPUTE, TRANSIENT). The storage cluster deals with data stored on slower devices, resulting in a predominance of read requests in the STORAGE workload. The compute cluster primarily stores data generated by calculations, which can undergo frequent modifications, while the transient cluster manages short-lived data with frequent insertions and deletions. Thus, both the COMPUTE and TRANSIENT workloads consist of a significant proportion of write requests. In the STORAGE workload, Aceso slightly outperforms FUSEE (1.10×). However, in the write-intensive COMPUTE and TRANSIENT workloads, Aceso outperforms FUSEE by up to 1.94×. This evaluation offers insights into Aceso's real-world performance, particularly its exceptional performance in write-intensive scenarios.

4.4 System-level Analysis

Memory Distribution. Figure 12 shows the memory distribution of the two KV stores after 184 clients each write 300,000 KV pairs, totaling approximately 52.6 GB of data. The sizes of the *Meta* and *Index Area* are omitted, as they are the same in both Aceso and FUSEE and fit within two preallocated regions per MN. *Valid* refers to the space occupied by valid KV pairs, *Redundancy* refers to the space occupied by the backup replicas in FUSEE and the parity data in Aceso, and *Delta* refers to the space occupied by DELTA blocks in Aceso. The result shows Aceso, employing erasure coding, reduces space overhead by approximately 44% compared to FUSEE. Additionally, the extra space occupied by DELTA blocks in Aceso is negligible (about 0.5GB, or 1% of DATA blocks), demonstrating that the offline strategy minimally impacts the space-saving effectiveness of erasure coding.

Factor Analysis. Figure 13 shows the performance impact of several key designs in Aceso and shows the step-by-step

Table 2. Impact of erasure code.

	Meta Area	Index Area							Block Area			
Erasure Code	Read Meta (ms)	Read Ckpt (ms)	Recover LBlock (ms)	LBlock Count (#)	Read RBlock (ms)	RBlock Count (#)	Scan KV (ms)	KV Count (#)	Recover OldLBlock (ms)	OldLBlock Count (#)	Total Time (ms)	Test Tpt (GB/s)
XOR	1.3	17	176	284	401	1194	95	1732607	2417	3371	3109	20.6
RS	1.4	17	289	297	403	1202	104	1779418	2959	3304	3775	12.6

Table 3. The average utilization of 4 cores in an MN.

Core ID	CPU 1	CPU 2	CPU 3	CPU 4
Usage	3.8%	41.9%	29.1%	43.1%

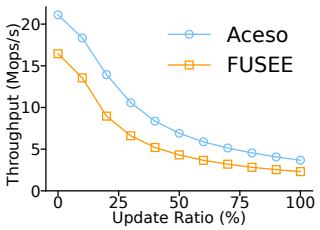


Figure 15. Throughputs under different UPDATE ratios.

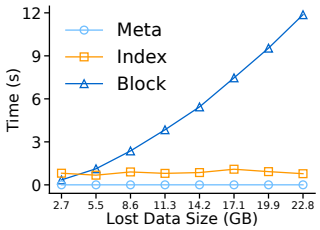


Figure 16. Recovery time under different lost data sizes.

evolution from FUSEE to Aceso. The baseline configuration, *ORIGIN*, corresponds to FUSEE. *+SLOT* extends index slots from 8B to 16B. *+CKPT* switches from index replication to index checkpointing. Finally, *+CACHE* enhances the local index cache to utilize slot addresses in addition to slot values, representing the full version of Aceso.

The result reveals that extending the slot length (*+SLOT*) increases bandwidth consumption, reducing the performance of the bandwidth-bound read requests (SEARCH), while having minimal impact on IOPS-bound write requests. The index checkpointing (*+CKPT*) reduces the number of CAS operations for write requests, significantly boosting performance, although read performance slightly declines due to a portion of bandwidth consumed by checkpointing. Finally, the additional utilization of slot addresses in Aceso's local index cache (*+CACHE*) conserves bandwidth, leading to improved read performance.

Degraded Search. To demonstrate the performance of degraded SEARCH, we have all clients concurrently write KV pairs from microbenchmarks to MNs. After 10 seconds, we shut down one MN and initiate its recovery, recovering only the index while leaving most old *DATA* blocks lost. In this scenario, we test the degraded SEARCH performance. The result is shown on the left side of Figure 14. *Normal* refers to the standard SEARCH's throughput, while *Special* refers to that of the degraded SEARCH, which is 0.53× the standard. Given Aceso's short recovery time (Figure 16), we consider this throughput acceptable.

Space Reclaimed Update. The right side of Figure 14 displays the impact of space reclamation on UPDATE requests. *Normal* refers to the standard UPDATE's throughput (without overwriting), while *Special* refers to that of the space reclaimed UPDATE (with overwriting). From the result, the throughput of space reclaimed UPDATE is nearly identical to that of standard UPDATE (0.97×), indicating that the performance impact of Aceso's space reclamation is negligible.

MN CPU load. Table 3 shows the average utilization of the 4 cores in an MN (CPU 1 for RPC serving, CPU 2 for erasure coding, CPU 3 for checkpoint sending, and CPU 4 for checkpoint receiving) when the index size per MN is 256MB, and all clients concurrently write KV pairs from microbenchmarks. The utilization of each core is below 50%, and since the utilization is independent of the number of clients, we consider this acceptable. Even with weaker MN compute power, the KV request performance remains unaffected. It only slightly increases the checkpoint interval and the index recovery time (see Figure 18).

4.5 Sensitivity Analysis

Impact of Update Ratio. Figure 15 shows the throughputs of YCSB workloads across various UPDATE-SEARCH ratios. As the proportion of UPDATE increases, the throughputs of both Aceso and FUSEE decline because UPDATE issues more I/O operations. However, Aceso achieves better throughput across all ratios due to its more advanced mechanisms.

Impact of Erasure Code. We follow the same process in the *Degraded Search* test (§4.4) to trigger an MN recovery, but this time we recover all three areas of the MN. Table 2 provides a breakdown analysis of the MN recovery in Aceso, revealing the difference between RS and XOR-based code (X-Code). The table displays the respective time for reading the metadata replica (*Read Meta*), reading the latest index checkpoint (*Read Ckpt*), recovering new (*new* refers to blocks with an *Index Version* of 0 or greater than or equal to the checkpoint) local blocks (*Recover LBlock*), reading new remote blocks from other MNs (*Read RBlock*), scanning KV pairs in these new blocks (*Scan KV*), and recovering old local blocks (*Recover OldLBlock*). It also shows the overall recovery time (*Total Time*) and the bandwidths (*Test Tpt*) of RS and XOR-based code when generating one 2MB PARITY block from six 2MB DATA blocks (simulating three DATA and three DELTA blocks), using a performance test program from the ISA-L library [10]. The result clearly shows the XOR-based code outperforming the RS code in erasure coding stages like *Recover LBlock* and *Recover OldLBlock*, saving 38% and 18% time, respectively. Other stages take a similar amount of time due to identical procedures. Overall, the XOR-based code reduces total recovery time by 18%. The *Test Tpt* shows XOR has a 68% higher throughput than RS. Therefore, Aceso opts for the XOR-based code.

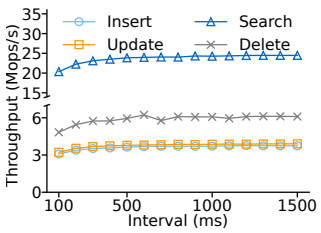


Figure 17. Throughputs under different checkpoint intervals.

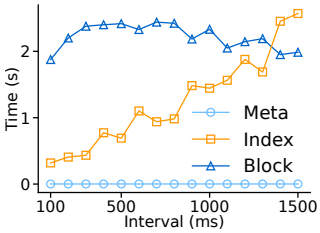


Figure 18. Recovery time under different checkpoint intervals.

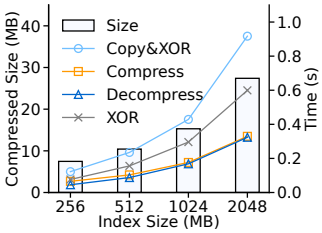


Figure 19. Impact of different index sizes.

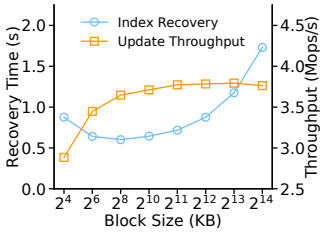


Figure 20. Impact of different block sizes.

Impact of Lost Data Size. Based on the previous setup, we adjust the timing of the MN crash to vary the lost data size, demonstrating its impact on MN recovery time. As shown in Figure 16, the *Meta Area* recovery time stays constant, given the unchanging total metadata size. Due to Aceso's index checkpointing mechanism, the *Index Area* recovery time is also unaffected (all within 1 second), as it only scans new KV pairs since the last checkpoint, with their total size staying relatively constant. The *Block Area* recovery time is proportional to the lost data size (approximately at a rate of 2GB/s). Nevertheless, Aceso's ability to resume normal write operations and degraded read operations after the *Index Area* recovery minimizes the impact on user experience.

Impact of Checkpoint Interval. Based on the same setup in the *Degraded Search* test (§4.4), we vary the checkpoint interval to observe its impact on Aceso's performance and recovery time. As shown in Figure 17, the checkpoint interval has minimal impact on performance, with only a slight decrease observed at 100ms. Figure 18 shows that longer intervals mainly increase the *Index Area* recovery time, as longer intervals require scanning more KV pairs to recover the index. The *Block Area* recovery time is slightly shortened as more blocks are already recovered during the index recovery. To mitigate the impact on system performance and shorten the *Index Area* recovery time, the checkpoint interval is set to 500ms in our experiments. It's worth noting that in read-intensive scenarios, the *Index Area* recovery time would be shorter due to fewer KV pairs needing to be scanned.

Impact of Index Size. To assess the impact of the index size, we allocate the index of each MN to different sizes and preload keys to achieve a load factor of around 0.75. Clients then concurrently execute `UPDATE` requests from microbenchmarks. Figure 19 shows the compressed index checkpoint sizes and the time taken for a single thread to execute each checkpointing step under different index sizes. *Size* refers to the compressed checkpoint size, *Copy&XOR* refers to generating a new checkpoint and XORing it with the previous one to generate the delta, *Compress* and *Decompress* refer to compressing and decompressing the delta via LZ4, and *XOR* refers to XORing the decompressed delta with the previous checkpoint. The result indicates that even for large indexes (*e.g.*, 2GB), the compressed checkpoint size is small (*e.g.*, 27MB), highlighting the importance of Aceso's differential checkpointing. The time for each step scales with index size. If a large index size causes checkpointing to take more than 500ms per round, the interval dynamically increases. We assume a maximum index size per MN of below 2GB, ensuring, as shown in Figure 18, that the *Index Area* recovery time does not reach 3 seconds. For larger index sizes, the extended *Index Area* recovery time can be alleviated by distributing coding stripe recovery tasks across multiple CNs, similar to RAMCloud [54], which we leave for future work.

Impact of Block Size. We vary the memory block size from 16KB to 16MB to observe its impact on the index recovery time and the performance of `UPDATE` requests. As shown in Figure 20, for the index recovery time, as the block size increases, the index recovery time initially decreases and then increases. The poor performance at small block sizes is due to severe data fragmentation affecting the two-stage pipelining of the RDMA reads and XOR computation during recovery. Conversely, larger block sizes result in larger unfilled blocks (the *Index Version* of the block is 0) to recover. For the `UPDATE` throughput, larger block sizes result in higher throughput as the frequency of server accesses required for block allocation decreases. Taking these factors into consideration, we set the default block size in Aceso to 2MB.

5 Related Work

Disaggregated Memory. Current research on disaggregated memory (DM) spans a wide range, encompassing underlying hardware [23, 26, 35, 38, 43, 50, 65], operating system kernels [2, 24, 56, 61, 70, 75, 79, 86], user-level programming libraries [49, 58, 60, 73, 74, 92], and user-level applications [21, 36, 48, 51, 63, 66, 89, 94]. Aceso is an advanced DM design tailored for the important cloud application, the KV store, which can also benefit from the low-level optimizations of prior works.

KV Stores on DM. To advance the development of KV stores on DM, various efforts have been undertaken. Many studies focus on optimizing the index structures, such as tree [39, 48, 76] and hash indexes [14, 80, 94]. Others, such as Clover [69], focus on reducing monetary and energy costs, DINOMO [36] focus on utilizing cache to enhance performance, and FUSEE [63] focus on a fully disaggregated architecture to unleash potential performance. Finally, Aceso is a KV store on DM with a focus on fault tolerance, achieving significant improvements in performance and space efficiency.

Fault Tolerance on DM. Some prior works have employed erasure coding in DM-based memory swapping systems, such as Hydra [37] and Carbink [92]. However, the scenario that their designs target substantially differs from that of DM-based KV stores. For instance, in DM-based memory swapping systems, the data in memory nodes is not shared between multiple compute nodes, without considering concurrent access. In contrast, Aceso carefully integrates erasure coding with checkpointing within the DM-based KV store and presents new optimization techniques to address their challenges on DM.

6 Conclusion

This paper presents Aceso, a DM-based KV store that combines checkpointing for the index and erasure coding for KV pairs to ensure fault tolerance. Aceso addresses the challenges to fully harness the benefit of this hybrid fault-tolerant mechanism, including a differential checkpointing scheme to reduce the bandwidth consumption, a versioning approach to recover lost index updates on failures, a delta-based space reclamation mechanism to reclaim obsolete KV pairs, a tiered recovery scheme to mitigate the extended recovery time caused by erasure coding. Compared to the state-of-the-art replication-based KV store on DM, Aceso achieves up to 2.7× throughput improvement, up to 54% tail latency reduction and 44% memory space savings.

7 Acknowledgments

We sincerely thank our anonymous shepherd and reviewers for their constructive comments and suggestions. We are also grateful to CloudLab [16] for the infrastructure support. Moreover, we thank Jiacheng Shen for his help clarifying inquiries. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK14218522). Pengfei Zuo (pengfei.zuo@huawei.com) and Ming-Chang Yang (mcyang@cse.cuhk.edu.hk) are the corresponding authors.

References

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association, 599–616.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece*. ACM, 14:1–14:16.

[3] Hang An, Fang Wang, Dan Feng, Xiaomin Zou, Zefeng Liu, and Jianshun Zhang. 2023. Marlin: A Concurrent and Write-Optimized B+-tree Index on Disaggregated Memory. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA*. ACM, 695–704.

[4] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994 Technical Conference, Boston, Massachusetts, USA, Conference Proceeding*. USENIX Association, 87–98.

[5] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. 1993. The primary-backup approach. *Distributed systems* 2 (1993), 199–216.

[6] Edouard Bugnion, Vitaly Chipounov, and George Candea. 2013. Lightweight Snapshots and System-level Backtracking. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA*. USENIX Association.

[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA*. USENIX Association, 209–223.

[8] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. 2014. Parity logging with reserved space: towards efficient updates and recovery in erasure-coded clustered storage. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA*. USENIX, 163–176.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA*. ACM, 143–154.

[10] Intel Corporation. 2017. Storage acceleration with ISA-L. https://storageconference.us/2017/Presentations/Tucker-1.pdf.

[11] NVIDIA Corporation. 2023. RDMA Aware Networks Programming User Manual. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/.

[12] William R. Dieter and James E. Lumpp Jr. 2001. User-Level Checkpointing for LinuxThreads Programs. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, Boston, Massachusetts, USA*. USENIX, 81–92.

[13] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies, FAST 2021*. USENIX Association, 33–49.

[14] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA*. USENIX Association, 401–414.

[15] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. 2023. Fast One-Sided RDMA-Based State Machine Replication for Disaggregated Memory. *ACM Trans. Archit. Code Optim.* 20, 2 (2023), 31:1–31:25.

[16] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA*. USENIX Association, 1–14.

[17] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA*. USENIX Association, 929–943.

[18] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[19] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. 2011. Diskreduce: Replication as a prelude to erasure coding in data-intensive

scalable computing (cmu-pdl-11-112). *Carnegie Mellon University Parallel Data Laboratory, Tech. Rep.* (2011).

[20] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[21] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. 2023. Citron: Distributed Range Lock Management with One-sided RDMA. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA*. USENIX Association, 297–314.

[22] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet L. Wiener. 2014. Fast database restarts at facebook. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA*. ACM, 541–549.

[23] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling With CXL. *IEEE Micro* 43, 2 (2023), 48–57.

[24] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA*. USENIX Association, 649–667.

[25] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. 2022. uKharon: A Membership Service for Microsecond Applications. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA*. USENIX Association, 101–120.

[26] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: a hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland*. ACM, 417–433.

[27] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[28] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA*. USENIX Association, 15–26.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA*. USENIX Association, 437–450.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA*. USENIX Association, 185–201.

[31] Antonios Katsarakis, Vasilis Gavrielatos, M. R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland*. ACM, 201–217.

[32] Kai Keller and Leonardo Bautista-Gomez. 2019. Application-Level Differential Checkpointing for HPC Applications with Dynamic Datasets. In *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2019, Larnaca, Cyprus*. IEEE, 52–61.

[33] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. 2023. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA*. USENIX Association, 31–48.

[34] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[35] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Koblenz, Germany*. ACM, 488–504.

[36] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. *Proc. VLDB Endow.* 15, 13 (2022), 4023–4037.

[37] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. 2022. Hydra : Resilient and Highly Available Remote Memory. In *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA*. USENIX Association, 181–198.

[38] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada*. ACM, 574–587.

[39] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA*. USENIX Association, 99–114.

[40] Qiliang Li, Liangliang Xu, Yongkun Li, Min Lyu, Wei Wang, Pengfei Zuo, and Yinlong Xu. 2024. Enabling Efficient Erasure Coding in Disaggregated Memory Systems. *IEEE Trans. Parallel Distributed Syst.* 35, 1 (2024), 154–168.

[41] Runhui Li, Yuchong Hu, and Patrick P. C. Lee. 2017. Enabling Efficient and Reliable Transition from Replication to Erasure Coding for Clustered File Systems. *IEEE Trans. Parallel Distributed Syst.* 28, 9 (2017), 2500–2513.

[42] Xiaolu Li, Zuoru Yang, Jinhong Li, Runhui Li, Patrick P. C. Lee, Qun Huang, and Yuchong Hu. 2021. Repair Pipelining for Erasure-coded Storage: Algorithms and Evaluation. *ACM Trans. Storage* 17, 2 (2021), 13:1–13:29.

[43] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), Austin, TX, USA*. ACM, 267–278.

[44] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA*. IEEE Computer Society, 189–200.

[45] Compute Express Link. 2023. Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/.

[46] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA*. IEEE Computer Society, 2884–2892.

[47] Xuhao Luo, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. SplitFT: Fault Tolerance for Disaggregated Datacenters via Remote Memory Logging. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece*. ACM, 590–607.

[48] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA*. USENIX Association, 553–571.

[49] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA*. ACM, 92–107.

[50] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada*. ACM, 742–755.

[51] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC 2013, San Jose, CA, USA*. USENIX Association, 103–114.

[52] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. 2006. Recording shared memory dependencies using strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA*. ACM, 229–240.

[53] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal*. ACM, 16:1–16:12.

[54] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal*. ACM, 29–41.

[55] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3 (2015), 7:1–7:55.

[56] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA*. USENIX Association, 181–198.

[57] Irving S. Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[58] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. 2024. Scaling Up Memory Disaggregated Applications with SMART. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA*. ACM, 351–367.

[59] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick P. C. Lee. 2024. ELECT: Enabling Erasure Coding Tiering for LSM-tree-based Storage. In *22nd USENIX Conference on File and Storage Technologies, FAST 2024, Santa Clara, CA, USA*. USENIX Association, 293–310.

[60] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association, 315–332.

[61] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA*. USENIX Association, 69–87.

[62] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany*. ACM, 675–691.

[63] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA*. USENIX Association, 81–98.

[64] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia*. ACM, 1–15.

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

[66] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. 2021. CoRM: Compactable Remote Memory over RDMA. In *SIGMOD '21: International Conference on Management of Data*. ACM, 1811–1824.

[67] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece*. ACM, 30:1–30:14.

[68] Infiniband trade association. 2023. InfiniBand. https://www.infinibandta.org/.

[69] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020*. USENIX Association, 33–48.

[70] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China*. ACM, 306–324.

[71] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA*. USENIX Association, 91–104.

[72] Dirk Vogt, Armando Miraglia, Georgios Portokalidis, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. 2015. Speculative Memory Checkpointing. In *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada*. ACM, 197–209.

[73] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association, 261–280.

[74] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA*. USENIX Association, 35–53.

[75] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA*. USENIX Association, 161–179.

[76] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In

SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA. ACM, 1033–1048.

[77] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association, 117–135.

[78] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA*. USENIX Association, 233–251.

[79] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. 2022. KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA*. USENIX Association, 121–136.

[80] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA*. ACM, 87–104.

[81] Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. 2023. TreeSLS: A Whole-system Persistent Microkernel with Tree-structured State Checkpoint on NVM. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany*. ACM, 1–16.

[82] Lihao Xu and Jehoshua Bruck. 1999. X-Code: MDS Array Codes with Optimal Encoding. *IEEE Trans. Inf. Theory* 45, 1 (1999), 272–276.

[83] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. 2024. IONIA: High-Performance Replication for Modern Disk-based KV Stores. In *22nd USENIX Conference on File and Storage Technologies, FAST 2024, Santa Clara, CA, USA*. USENIX Association, 225–241.

[84] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3 (2021), 17:1–17:35.

[85] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. 2017. Erasure coding for small objects in in-memory KV storage. In *Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR 2017, Haifa, Israel*. ACM, 14:1–14:12.

[86] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy*. ACM, 266–282.

[87] Jeremy Zawodny. 2009. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine* 79, 8 (2009), 1–10.

[88] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA*. USENIX Association, 167–180.

[89] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA*. USENIX Association, 801–819.

[90] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA*. USENIX Association, 51–68.

[91] Yiying Zhang, Jian Yang, Amir Saman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey*. ACM, 3–18.

[92] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA*. USENIX Association, 55–71.

[93] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proc. ACM Manag. Data* 1, 2 (2023), 131:1–131:26.

[94] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021*. USENIX Association, 15–29.