CIDER: Boosting Memory-Disaggregated Key-Value Stores with Pessimistic Synchronization

Yuxuan Du
College of Computer Science and Artificial Intelligence, Fudan University
National Key Laboratory of Parallel and Distributed Computing, China
duyx23@m.fudan.edu.cn

Xuchuan Luo
College of Computer Science and Artificial Intelligence, Fudan University
National Key Laboratory of Parallel and Distributed Computing, China
xcluo23@m.fudan.edu.cn

Xin Wang
College of Computer Science and Artificial Intelligence, Fudan University
xinw@fudan.edu.cn

Yangfan Zhou
College of Computer Science and Artificial Intelligence, Fudan University
National Key Laboratory of Parallel and Distributed Computing, China
zyf@fudan.edu.cn

Jiacheng Shen
Duke Kunshan University
jc.shen@dukekunshan.edu.cn

ABSTRACT

Memory-disaggregated key-value (KV) stores suffer from a severe performance bottleneck due to their I/O redundancy issues. A huge amount of redundant I/Os are generated when synchronizing concurrent data accesses, making the limited network between the compute and memory pools of DM a performance bottleneck. We identify the root cause for the redundant I/O lies in the mismatch between the optimistic synchronization of existing memory-disaggregated KV stores and the highly concurrent workloads on DM. In this paper, we propose to boost memory-disaggregated KV stores with pessimistic synchronization. We propose CIDER, a compute-side I/O optimization framework, to verify our idea. CIDER adopts a *global write-combining* technique to further reduce cross-node redundant I/Os. A *contention-aware synchronization* scheme is designed to improve the performance of pessimistic synchronization under low contention scenarios. Experimental results show that CIDER effectively improves the throughput of state-of-the-art memory-disaggregated KV stores by up to 6.6× under the YCSB benchmark.

PVLDB Reference Format:
Yuxuan Du, Xuchuan Luo, Xin Wang, Yangfan Zhou, and Jiacheng Shen. CIDER: Boosting Memory-Disaggregated Key-Value Stores with Pessimistic Synchronization. PVLDB, 19(8): 1701 - 1714, 2026.
doi:10.14778/3811243.3811245

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dmemsys/CIDER.

1 INTRODUCTION

Memory-disaggregated key-value (KV) stores, *i.e.*, KV stores on the disaggregated memory (DM) architecture, are widely discussed in both academia and industry [24, 26, 27, 42, 43, 48, 63]. DM decouples CPU and memory resources from individual monolithic servers into independent compute and memory pools. Compute nodes in the compute pool access and modify data in the memory pool with high-performance networking, *e.g.*, remote direct memory access (RDMA) [6] and compute express link (CXL) [13]. Compared with traditional distributed KV stores, memory-disaggregated KV stores can achieve better elasticity and resource efficiency due to their enhanced flexibility in resource management.

To ensure data consistency when client threads[1] concurrently access and modify data in the memory pool, it is necessary that clients can efficiently synchronize their memory pool operations. Unfortunately, achieving efficient synchronization poses a significant challenge for memory-disaggregated KV stores, particularly under conditions of high skew and substantial concurrency. In general, two approaches are adopted to synchronize concurrent data accesses, *i.e.*, lock-free optimistic synchronization and lock-based pessimistic synchronization. Existing memory-disaggregated KV stores tend to employ optimistic synchronization schemes to avoid the high lock maintenance overhead [11, 33, 63]. However, optimistic synchronization approaches suffer from poor performance due to their severe I/O redundancy issues. Extensive redundant I/Os are generated since optimistic synchronization handles conflicting data accesses and modifications by iteratively retrying conflicting operations. The redundant I/Os quickly saturate the limited network bandwidth and IOPS of the memory pool, resulting in severe performance degradation. According to our preliminary experiments, existing memory-disaggregated KV stores suffer from more than 2.7× slowdown due to the wasted IOPS and bandwidth when synchronizing concurrent data accesses.

The primary cause of the I/O redundancy lies in the inadequacy of optimistic synchronization under the highly contended real-world workloads on DM. In this paper, we propose to enhance the performance of memory-disaggregated KV stores with lock-based pessimistic synchronization. Nonetheless, two significant challenges must be overcome to achieve high performance.

[1] In the rest of this paper, we use clients as an abbreviation for client threads.

Proceedings of the VLDB Endowment, Vol. 19, No. 8 ISSN 2150-8097.
doi:10.14778/3811243.3811245

1) Inter-node redundant data modifications limit system throughput. While pessimistic synchronization effectively reduces redundant I/Os during data synchronization, it falls short in addressing the I/O redundancy issues during concurrent data modifications. Specifically, when multiple clients concurrently modify the same data, the data modified by one client is frequently overwritten by others, causing redundant data modifications. Existing approaches mitigate this issue by proposing a local write combining scheme [26, 27] that consolidates redundant data modifications among multiple clients within their corresponding compute nodes. However, this approach only mitigates intra-node redundant data modifications, leaving inter-node I/O redundancy unresolved.

2) High lock overhead under low contention workloads. Although lock-based pessimistic synchronization performs well in highly contended scenarios, its efficiency declines under workloads with low contention. Specifically, locks have to be maintained in the memory pool to coordinate clients from multiple compute nodes. Additional remote memory accesses are required to acquire and release locks on the critical path of each data access operation. Operation latency significantly increases due to the high remote memory access overhead, *i.e.*, an order of magnitude higher than local memory accesses. System throughputs are thus compromised due to the higher per-operation I/O overhead.

We design CIDER, a compute-side I/O optimization framework, to enhance the performance of memory-disaggregated KV stores with pessimistic synchronization. CIDER adopts a distributed **M**ellor-**C**rummey-**S**cott (MCS) lock, the state-of-the-art lock mechanism designed for DM [20], to implement efficient pessimistic synchronization. Over the MCS lock, we further propose two techniques to address the above two challenges. First, to eliminate extensive cross-node redundant data modifications, CIDER employs a *global write-combining* technique. It organizes redundant and concurrent data modifications in a global queue and executes them through a single consolidated data modification. Second, to improve system performance under low contention scenarios, CIDER designs a *contention-aware synchronization* scheme. It dynamically identifies the contention level of the current workload on the client side and allows clients to dynamically switch between optimistic and pessimistic synchronization modes.

We implement CIDER from scratch and evaluate it with both micro-benchmarks and end-to-end evaluation under YCSB workloads [14]. For the micro-benchmark, we design a minimalistic object storage (termed as a pointer array), to quantify the pure performance advancement brought about by CIDER. For the end-to-end evaluations, we integrate CIDER into RACE [63] and SMART [27], two state-of-the-art memory-disaggregated KV stores that adopt hash- and tree-based indexes, respectively, to show the overall performance boost. Our evaluation results show that CIDER outperforms the state-of-the-art optimistic and pessimistic synchronization schemes on DM under our micro-benchmark by up to 6.7× and 2.0× in throughput, respectively. Furthermore, by integrating CIDER, RACE, and SMART achieve up to 5.1×, 6.6× higher throughput and 12.4×, 13.8× lower P99 latency under the write-intensive workload, respectively.

The contributions of this paper can be summarized in the following three aspects:

- We identify the performance issue with optimistic synchronization in existing memory-disaggregated KV stores with thorough experiments.
- We propose the idea of enhancing memory-disaggregated KV stores with lock-based pessimistic synchronization and design CIDER to address the challenges of achieving efficient pessimistic data synchronization on DM.
- We implement CIDER from scratch and evaluate it with extensive experiments. Our evaluation results show that CIDER boosts the throughput of state-of-the-art memory-disaggregated KV stores by up to 5.1× under the write-intensive workload.

2 BACKGROUND AND MOTIVATIONS

In this section, we first introduce the disaggregated memory architecture. We then introduce memory-disaggregated KV stores, focusing on their performance issues incurred by optimistic synchronization. Finally, we introduce existing lock-based pessimistic synchronization approaches on DM and conduct preliminary experiments to show the potential performance boost of adopting them on memory-disaggregated KV stores.

2.1 Disaggregated Memory

Disaggregated memory was proposed to attack the resource inefficiency issue due to the coupled allocation of CPU and memory in monolithic servers [4, 21, 23, 30, 36, 38, 41, 50]. DM decouples computing and memory resources from monolithic servers into independent compute and memory pools. Compute nodes (CNs) in the compute pool have abundant CPUs but only a limited amount of local memory to serve as the runtime cache. Memory nodes (MNs) in the memory pool host sufficient memory capacity but only a few weak CPU cores to execute lightweight management computation, *i.e.*, network connection management and memory allocation management. The compute and memory pools are connected with high-performance data center interconnect techniques, *e.g.*, RDMA [6] and CXL [13]. CNs can thus directly access and modify in-memory data on MNs, bypassing their weak CPUs. In this paper, without loss of generality, we assume CNs access MNs with one-sided RDMA operations, *i.e.*, RDMA_READ, RDMA_WRITE, atomic compare and swap (RDMA_CAS), and atomic fetch and add (RDMA_FAA). Other interconnect techniques that provide the same interfaces are compatible with our design.

2.2 Memory-Disaggregated Key-Value Stores

Many works port in-memory KV stores to DM to achieve better resource efficiency and elasticity [2, 24, 42, 43]. Existing memory-disaggregated KV stores are designed for simplicity and speed in accessing individual data with their keys [26, 27, 48, 63]. They organize stored objects as key-value (KV) pairs and provide clients with standard KV interfaces, *i.e.*, `SEARCH(K)`, `INSERT(K, V)`, `UPDATE(K, V)`, and `DELETE(K)` [25, 43, 63]. Complex operations, *e.g.*, transactions, are not usually supported. Meanwhile, clients are responsible for handling invalid operations, *i.e.*, any attempt to `INSERT` an existing key, or to `SEARCH`, `UPDATE`, or `DELETE` a non-existent key will not be executed, and the client will receive a return value indicating that the operation is invalid [32, 39]. A global index data structure is maintained in the memory pool to keep track of the mapping between the keys and values.

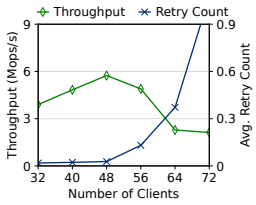

Figure 1: The throughput and retry count of the pointer array with optimistic synchronization under a highly-contented write-intensive workload.

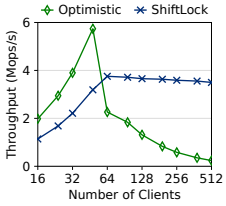

Figure 2: The throughput of the pointer array with optimistic and pessimistic synchronizations as a function of the number of clients.

Figure 3: The throughput of RACE with optimistic and pessimistic synchronizations as a function of the number of clients.

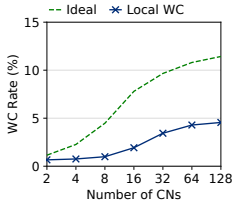

Figure 4: The actual WC rate of local WC and the theoretical upper bound as a function of the number of CNs.

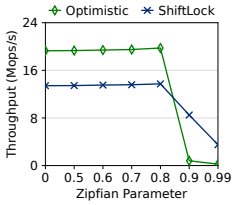

Figure 5: The throughput of optimistic and pessimistic synchronizations as a function of the skew factor (larger values mean more skew).

Memory-disaggregated KV stores are featured for their highly skewed and concurrent workloads [26, 27, 48, 63]. Efficiently synchronizing clients in the compute pool to concurrently access data in the memory pool is critical to achieving high performance. Existing approaches typically adopt an optimistic synchronization scheme. They store pointers of KV pairs in their global index and synchronize concurrent data accesses in a lock-free manner [11, 33, 63]. When performing SEARCH operations, clients on CNs first search for the target data pointer in the index with a series of RDMA operations. Then the KV pair is fetched according to the pointer with RDMA_READ. When performing INSERT, DELETE, and UPDATE operations (IDU), they first write the modified KV pair to a new location in the memory pool and then atomically modify the data pointer in the global index with RDMA_CAS. The atomicity of RDMA_CAS ensures that only one client can successfully modify the data pointer, and the values of pointers are always consistent. Consequently, SEARCH operations can always get the correct pointer and execute concurrently with IDU operations.

Compared with pessimistic synchronization, optimistic synchronization eliminates two additional remote memory accesses on acquiring and releasing locks in the memory pool. The operation latency can be reduced due to the reduced number of network round-trip times (RTTs) on the critical path. Overall system throughput can also be improved in low contention workloads due to the lower per-operation overhead of IDU operations.

However, under highly contented workloads, optimistic synchronization becomes a significant performance bottleneck. Specifically, when multiple clients concurrently IDU the same KV pair, only one thread can succeed due to the atomicity of RDMA_CAS. Other threads have to iteratively retry their operations until they successfully modify the data pointer. Suppose, in the worst-case, when n threads concurrently IDU the same pair and are acting in perfect synchrony, there will be n rounds of retry operations since only one thread can succeed in each round. However, during the process, $\frac{n(n-1)}{2}$ retry operations will be generated, causing $O(n^2)$ redundant operations. These retries quickly saturate the limited IOPS and bandwidth of the memory pool. System throughput is thus compromised since other operations are blocked by these redundant retries. This is a severe issue for practical memory-disaggregated KV stores since real-world workloads typically follow Zipfian distributions with a high skewness [7, 53]. The high skewness leads to severe contention during concurrent execution.

We integrate optimistic synchronization into a pointer array to illustrate its issues in high-contention scenarios. Specifically, the pointer array is composed of 60 million pointers. Each pointer stores the address of a unique KV pair in the memory pool. Clients in the compute pool perform SEARCH and UPDATE operations on the pointer array with optimistic synchronization. We evaluate the throughput and number of retried operations of the pointer array under a write-intensive workload with 50% SEARCH and 50% UPDATE. Requests are generated following a Zipfian distribution with $\theta = 0.99$ to reflect real-world workloads with skewed data access patterns. As shown in Figure 1, with less than 48 clients, the throughput of the pointer array scales linearly, and the number of retried operations stays low. This is because the pointer array is bottlenecked by the computing power of clients. With more than 48 clients, the throughput drops by up to 2.7× as the number of clients increases. This can be explained by the dramatic increase in the number of retried operations generated when handling conflicting UPDATE operations. These retry operations quickly saturate the RDMA network interface cards (RNICs) in the memory pool, preventing other operations from being normally executed.

2.3 Pessimistic Synchronization on DM

The spinlock is the most widely adopted pessimistic synchronization approach on DM. It is typically implemented by iteratively polling a remote memory address with atomic RDMA_CAS [26, 27, 48]. The key problem with spinlocks on DM lies in their expensive polling overhead incurred on the RNICs of the memory pool.

ShiftLock [20] is the state-of-the-art lock mechanism on DM. ShiftLock ports the **M**ellor-**C**rummey-**S**cott (MCS) lock [31, 40], which is originally designed for NUMA systems, to DM to relieve the network bottleneck of the memory pool. Specifically, ShiftLock maintains each lock as a linked list. All clients waiting for the same lock are queued in the same linked list, indicating the order of getting locks. For each lock, a lock entry is maintained in the memory pool to record the client ID at the tail of the list. When a client acquires a lock, it appends itself to the list by first executing an atomic get-and-set operation on the corresponding lock entry. The get-and-set operation is implemented with masked RDMA_CAS by setting the *compare_mask* to 0 to bypass the comparison in a normal RDMA_CAS operation [15, 20]. With the previous tail in the return value of the get-and-set operation, the client finishes the lock acquisition by notifying the previous tail about the insertion of a

new client. When a client releases a lock, it transfers the ownership of the lock to the next client by informing the following client in the list with a network request. The polling overhead on memory-side RNICs is thus shifted to client-side RNICs, which significantly improves system performance.

Motivation. In this paper, we propose to adopt lock-based pessimistic synchronization to enhance the performance of memory-disaggregated KV stores. To validate the potential of using pessimistic synchronization to address the limitations of optimistic synchronization under highly contended workloads, we incorporate the MCS lock of ShiftLock into KV stores with different index structures, *e.g.*, the pointer array and the RACE hash [63]. Figure 2 shows the throughput of the pointer array with optimistic and pessimistic synchronization under the same workload as in Section 2.2. With optimistic synchronization, the throughput of the pointer array significantly drops after reaching the peak at 48 clients. However, with pessimistic synchronization, *i.e.*, ShiftLock [20], the performance remains stable. As shown in Figure 3, a similar trend is observed on RACE hashing [63], indicating that pessimistic synchronization also works on real-world applications.

3 CHALLENGES

The performance of memory-disaggregated KV stores after adopting pessimistic synchronization is still suboptimal due to two challenges: 1) severe cross-node redundant data modifications and 2) high lock maintenance overhead. In this section, we introduce these two challenges in detail with thorough experimental analyses.

3.1 Inter-Node Redundant Data Modifications

Even though pessimistic synchronization effectively reduces redundant I/Os during data synchronization, substantial I/O redundancy still exists during concurrent IDU operations. Specifically, when multiple clients concurrently execute IDU operations on the same KV pair, all of them first write the modified KV pair to a new location in the memory pool with RDMA_WRITE. They then update the same pointer in the global index data structure with RDMA_CAS. Since the data pointer is protected by a lock, the update of the pointer happens sequentially. Most of the concurrent IDU operations in the process are wasted since they will be quickly overwritten by subsequent IDU operations to the same KV pair, resulting in wasted RDMA_WRITE and RDMA_CAS operations. Such a problem is particularly acute in production environments since real-world workloads are usually skewed [14].

Existing approaches adopt a local write-combining (WC) technique to reduce redundant I/Os incurred by concurrent IDU operations [26, 27]. The key idea is to consolidate concurrent IDU operations to the same KV pair in a small time window into one IDU operation. To detect redundant IDU operations during the time window, existing approaches associate a lock with each KV pair locally on each CN. The client that successfully acquires the local lock becomes the combiner and executes the IDU operation to the memory pool. Concurrent clients that issue IDU operations to the same KV pair are all blocked by the local lock and combined by the combiner. They directly return the combiner's result after successfully acquiring the lock since their operations are already executed. To ensure the correctness of the combined KV pair, existing approaches adopt a last-writer-wins conflict resolution scheme [27, 28]. A WC buffer is maintained locally on each CN for each KV pair. All blocked clients write their modified KV pair to the WC buffer by overwriting the KV pair written by previous clients. Consequently, the WC buffer always contains the KV pair written by the last writer. The combiner finally writes the KV pair to the memory pool, which indicates the value written by the last writer.

However, local WC can only handle redundant I/Os of IDU operations issued by clients within the same CNs. There are still extensive amounts of redundant I/Os generated by clients across CNs. To illustrate this issue, we integrate local WC to the pointer array described in Section 2.2 and evaluate its WC rate with increasing numbers of CNs. Since we only have access to a limited number of physical servers, we simulate a large number of virtual CNs by partitioning our servers. Specifically, each virtual CN consists of 4 CPU cores, and we run a client on each core, thus allowing a maximum of 4 concurrent clients on our CNs. The WC rate is calculated as $\frac{\#combined\_requests}{\#total\_requests}$. We also evaluate the redundant request rate to show an upper bound for the WC rate, *i.e.*, the ideal WC rate. The redundant request rate is calculated as the $\frac{\#combined\_requests + \#blocked\_requests}{\#total\_requests}$, where $\#blocked\_requests$ is the number of requests blocked by lock-based pessimistic synchronization. As shown in Figure 4, both the redundant request rate and the local WC rate increase with the number of CNs. Approximately 7% of redundant requests are not combined. These requests predominantly access hot keys and exhibit high latency. Since requests for hot keys must be serialized through lock acquisition, their throughput is constrained by latency. Consequently, this 7% of requests significantly degrades both system throughput and tail latency.

3.2 High Lock Maintenance Overhead

The second challenge lies in the high lock maintenance overhead. When leveraging lock-based pessimistic synchronization, clients have to acquire and release a lock before and after performing IDU operations. Both acquiring and releasing locks are remote memory accesses since locks are maintained in the memory pool to synchronize all clients in the compute pool. This incurs two additional RTTs on the critical path of each IDU operation. The system throughput is thus compromised in low contention workloads due to the higher per-operation latency and I/O overhead.

We verify the insufficiency of naively adopting pessimistic synchronization by evaluating the performance of the pointer array with optimistic and pessimistic synchronization under various contention scenarios. The contention can be affected by both the number of concurrent clients and the skewness of the workload.

First, under the same number of clients, the higher the skewness, the higher the contention. Figure 5 shows the performance of the pessimistic and optimistic pointer arrays with 512 clients under the write-intensive workload with 50% `SEARCH` and 50% `UPDATE`. We use different θs of the Zipfian distribution to control the skewness of the workload, *i.e.*, a larger θ indicates a more skewed workload. The throughput of pessimistic synchronization is only about 70% that of the optimistic approach with $\theta \leq 0.8$ due to the higher per-operation I/O overhead. However, under more skewed workloads, the throughput of pessimistic synchronization is up to 14× better since there are fewer redundant I/Os generated by the retry operations of optimistic synchronization.

Moreover, under the same skewness, the more clients, the higher the contention. Our previous results in Figure 2 show a similar trend,

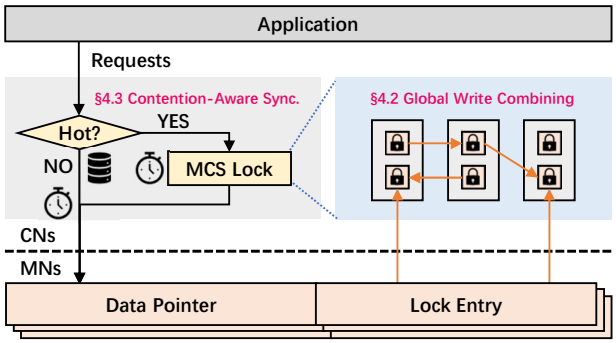

Figure 6: The overview of CIDER.

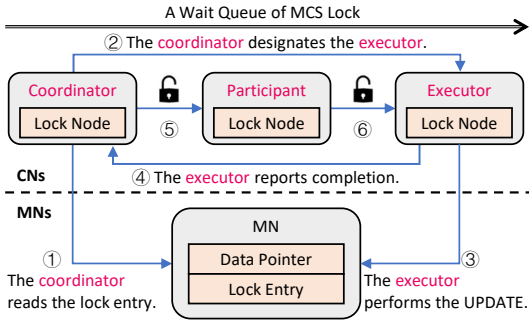

Figure 7: The workflow of global WC.

i.e., optimistic synchronization performs better with a small number of clients, while pessimistic synchronization performs better when the number of clients is larger.

4 THE CIDER DESIGN

4.1 CIDER Overview

We propose CIDER, a compute-side I/O optimization framework that leverages pessimistic synchronization to boost existing memory-disaggregated KV stores that conduct `INSERT`, `DELETE`, `UPDATE`, and `SEARCH` on data pointers as defined in Section 2.2. Figure 6 shows the overview of CIDER. CIDER adopts the distributed MCS lock in ShiftLock [20] to achieve pessimistic synchronization. We disable the reader-writer lock optimization of ShiftLock since it adds additional overhead on single-key accesses. To reduce the number of redundant `UPDATE` operations (**Challenge 1**), we propose a global write-combining (WC) technique that leverages MCS locks to detect and consolidate concurrent and redundant `UPDATE` operations across CNs. To improve pessimistic synchronization in low-contention scenarios (**Challenge 2**), we propose a contention-aware synchronization scheme that adaptively switches between pessimistic and optimistic synchronization schemes according to the contention level of individual keys. We introduce these two techniques in § 4.2 and § 4.3, respectively.

4.2 Global Write Combining

Global WC is proposed to reduce cross-node redundant I/Os by aggregating IDU requests from multiple CNs. Compared with local WC, the key challenge of global WC lies in efficiently detecting and correctly combining redundant IDU operations, *i.e.*, concurrent IDU operations that modify the same KV pair. One straightforward approach is to adopt a centralized coordination server that processes all IDU operations and combines redundant ones. However, the centralized server can become a severe performance bottleneck when the number of clients grows, making the system hard to scale to a large number of clients.

The key opportunity to address this issue lies in the MCS lock mechanism. For each KV pair, the MCS lock maintains a distributed wait queue across CNs, obviating the need for an additional centralized server to detect concurrent and redundant requests. Moreover, since the order of clients in the wait queue indicates the order of performing IDU operations, the operations can be safely combined with a last-writer-wins conflict resolution scheme [28, 43].

Based on this observation, we propose global WC over the MCS lock. In the rest of this section, we first introduce how CIDER achieves `UPDATE` operations in Section 4.2.1. `INSERT` and `DELETE` operations are separately introduced in Section 4.2.2 since they are handled differently to guarantee correctness.

4.2.1 Combining `UPDATE` *operations.* Figure 7 presents the workflow of global WC over `UPDATE` operations. Among CNs, clients concurrently executing `UPDATE` operations on the same KV pair are organized into the same wait queue with a lock node. Inside the queue, CIDER distinguishes three types of clients, *i.e.*, coordinators, participants, and executors. Coordinators are clients who initiate a global WC. Executors are clients that actually execute the operations with remote memory accesses. Finally, participants are other clients who are combined by the executor. On MNs, CIDER associates each data pointer with a lock entry to synchronize concurrent modifications to the corresponding KV pair.

Figure 8 shows the detailed structures of the lock node, lock entry, and data pointer. On CNs, each lock node contains a 64-bit `Next` field to indicate the next client in the wait queue, a 16-bit `Coordinator` field to record the coordinator of global WC, a 16-bit `Result` field to propagate the result of the combined operation, and a 32-bit `Locked` field to handover the ownership of the lock. Data pointers on MNs contain 64 bits with a 60-bit `Pointer` and a 4-bit `Version` to handle `DELETE` operations, which will be discussed later (§ 4.2.2). Lock entries serve as gateways for individual MCS locks. They contain a 60-bit `Tail` to store the client ID at the tail of the current wait queue and a 64-bit `Epoch` for fault tolerance. A 4-bit `Version` is also maintained to deal with `DELETE` operations.

Global WC starts when a client successfully acquires the lock. The client then checks if there are other `UPDATE` operations that can be combined before modifying the data pointer. It achieves this by examining if the `Next` field in its local lock node is modified by subsequent clients. If the `Next` field remains empty, the client becomes an executor and normally finishes the `UPDATE`. Otherwise, the client becomes a coordinator and starts combining subsequent operations. The coordinator first identifies the executor, *i.e.*, the client on the tail of the wait queue, by reading the lock entry on the MN (Step ①). The coordinator then notifies the executor to start combining operations by transferring ownership of the lock to the executor and sending it the coordinator's ID through modifying its `Coordinator` field (Step ②). After getting the ownership of the lock, the executor finishes the `UPDATE` operation by first writing

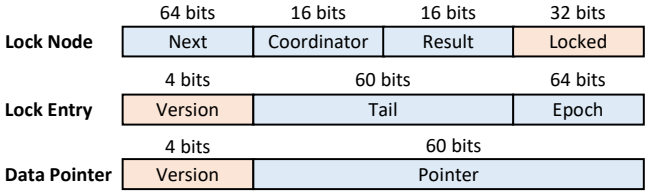

Figure 8: The structures of the lock node, lock entry, and data pointer.

the updated KV pair to a newly allocated location and then modifying the data pointer (Step ③). The executor then notifies the coordinator of the result of the combined UPDATE operations and transfers the ownership of the lock back (Step ④). The coordinator then skips its UPDATE operation, returns the result to upper-level applications, and transfers the ownership of the lock and the result along the wait queue (Step ⑤). The result of the combined UPDATE is conveyed by modifying the Result field in the lock node. Meanwhile, the Locked field is set to a special value 0x3 to indicate that the client's UPDATE is combined by the executor. All subsequent participants also skip their UPDATE operations and propagate the WC result through successive lock transfers when finding the Locked field is set to 0x3 (Step ⑥).

Compared with only adopting the standard MCS lock, global WC introduces an additional remote memory access overhead to identify the executor (Step ①) and two cross-CN communications (Steps ② and ④). However, it ensures that only a single UPDATE is actually executed on MN per combined batch, regardless of the number of requests in the batch. This design fundamentally eliminates cross-node redundant I/Os, especially I/Os on MNs. System performance can thus be improved under highly concurrent workloads with skewed access patterns.

4.2.2 Combining INSERT *and* DELETE *operations.* Handling INSERT and DELETE operations is difficult for global WC due to the mismatch between the targets of locks and those of global WC. Specifically, all concurrent operations modifying the same data pointer will be organized in the same wait queue, while only operations on the same KV pair can be combined. INSERT and DELETE operations can alter the key associated with the same data pointer, compromising the correctness of global WC.

CIDER ensures that operations in the same wait queue modify the same key by handling INSERT and DELETE operations differently. For INSERT operations, clients directly write the new KV pair to the memory pool and modify the empty pointer with RDMA_CAS without entering the MCS wait queue. For DELETE operations, CIDER ensures they are always the last operations on their corresponding wait queues. This ensures that all operations in the queue target the same KV pair and will be combined by the last DELETE.

CIDER leverages the version numbers in lock entries and data pointers to deal with DELETE operations. When performing a DELETE operation, a client first increments the version number in the lock entry when acquiring locks. The version number in the data pointer is incremented after the DELETE finishes. When performing UPDATE operations, clients first read the data pointer to get the version number and the address of the old KV pair. Then the client acquires the lock with the version number in the data pointer. The lock acquisition operation will be rejected when there is a mismatch between the two version numbers, indicating the target KV pair is already deleted. UPDATE operations initiated in between the start and finish of the DELETE operations are thus rejected from the wait queue, ensuring the correctness of global WC.

4.3 Contention-Aware Synchronization

As we discussed in Section 3, optimistic and pessimistic synchronizations exhibit complementary performance characteristics under workloads with different levels of contentions. This motivates us to design contention-aware synchronization to adapt synchronization schemes dynamically, matching the contention level of workloads.

A straightforward approach to achieve this is to adopt a centralized manager process. The manager samples requests from CNs to MNs to monitor the level of contention of the current workload. When the observed contention level changes significantly, it pauses new requests, switches the concurrency control protocol, and resumes operation. However, this centralized and coarse-grained approach poses several inherent limitations. First, there is a fundamental trade-off between the monitoring overhead and estimation accuracy, *i.e.*, higher sampling frequency improves the accuracy of load estimation but introduces additional system overhead. Second, switching between synchronization schemes requires pausing all clients, which impacts service stability and increases tail latency.

We propose to *dynamically decide synchronization modes with a decentralized and fine-grained arbitration mechanism* to avoid maintaining an additional centralized manager process. The key idea is similar to credit-based throttling [34]. Clients monitor the performance statistics of executing each IDU operation and assign each data pointer credits. A higher credit indicates a higher contention level. When executing IDU operations, clients adaptively decide to adopt optimistic or pessimistic synchronization according to the credit of the data pointer. Such a decentralized approach can effectively perceive data access patterns and adapt to workload change without the overhead induced in a centralized method.

Algorithm 1 shows the process of contention-aware synchronization, which involves changes only to the final step of the update operation, *i.e.*, the modification of data pointers. Each CN maintains two hash maps to track the contention level of workloads, *i.e.*, a *credit* map to record credits of each data pointer and a *retryRecord* map to record retry counts associated with each data pointer.

When updating a data pointer, a client first decides the synchronization mode by checking whether the target data pointer possesses any credits. If so, it consumes one credit and proceeds with pessimistic synchronization (Lines 3-4). In the pessimistic mode, global WC is adopted to reduce redundant data modifications. If the client becomes the executor of global WC, it needs to first write the KV and then update the pointer with an RDMA_CAS operation (Lines 9-12). In all other cases, the client will not hold the lock and is blocked until the combined request has been processed. Finally, the client dynamically adjusts the number of credits based on congestion assessment, which is determined by the batch size of global WC (Lines 13-16).

Otherwise, if the target data pointer does not contain any credit, it executes the operation with optimistic synchronization since the target key is infrequently accessed (Lines 18-19). Similarly, the client evaluates congestion by comparing the number of CAS retries

Algorithm 1: Contention-Aware Synchronization

Global Variables :
/* The following two variables are maintained on each CN. */
credit: A hash map recording the credit count.
retryRecord: A hash map tracking the number of retries.
Algorithm Inputs :
ptrAddr: The address of the target data pointer.
KV: The KV pair to be updated.
oldPtr: The value of the data pointer.

```
1  Function update(ptrAddr, KV, oldPtr):
2      if credit[ptrAddr] > 0 then
3          credit[ptrAddr] -= 1 ;
4          pessimisticUpdate(ptrAddr, KV, oldPtr)
5      else
6          optimisticUpdate(ptrAddr, KV, oldPtr)
7  Function pessimisticUpdate(ptrAddr, KV, oldPtr):
8      WCResult ← ptrAddr.tryLockAndWC();
9      if the client becomes the executor of global WC then
           /* Allocate a new memory block, write the KV pair to
              it with RDMA_WRITE, and return the address of
              the newly written KV.                          */
10         newPtr ← allocateAndWriteKV(KV);
           /* Update data pointer with RDMA_CAS, record and
              return the number of retries.                  */
11         nRetry ← CASPointer(ptrAddr, oldPtr, newPtr);
12         ptrAddr.unlock();
       /* In all other cases, ptrAddr will NOT lock.         */
13     if WCBatchSize > 1 then
14         credit[ptrAddr] += 2;
15     else
16         credit[ptrAddr] /= AIMD_FACTOR;
17 Function optimisticUpdate(ptrAddr, KV, oldPtr):
18     newPtr ← allocateAndWriteKV(KV);
19     nRetry ← CASPointer(ptrAddr, oldPtr, newPtr);
20     if nRetry >= HOTNESS_THRESHOLD &&
        retryRecord[ptrAddr] >= HOTNESS_THRESHOLD
        then
21         credit[ptrAddr] += INITTIAL_CREDIT;
22     retryRecord[ptrAddr] ← nRetry;
```

between the current and previous attempts, and adjusts credits accordingly (Lines 20-22).

To promptly detect contention level variations, we adopt the additive increase and multiplicative decrease scheme (AIMD) [12] to update credits. Specifically, we increase the credit after a successful global WC, while dividing the credit by an `AIMD_FACTOR` when no concurrent requests are detected for global WC (Lines 13-16). The `AIMD_FACTOR` is empirically set to 2, which is consistent with the original algorithm [12]. For optimistic synchronization, if two consecutive requests require two or more retries (*i.e.*, the `HOTNESS_THRESHOLD` is set to 2), we increase the credit by `INITIAL_CREDIT` (Lines 20-21), which is empirically set to 36.

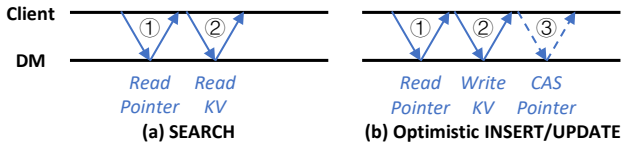

Figure 9: The workflows of `SEARCH`, `INSERT`, and `UPDATE` operations under the optimistic mode.

The contention-aware synchronization scheme enables fine-grained and seamless transitions between synchronization modes for individual KV pairs. For hot KV pairs, pessimistic synchronization with global WC eliminates redundant IDU operations by batching modifications and preventing unnecessary retries. For cold KV pairs, optimistic synchronization avoids the expensive lock maintenance overhead and reduces operation latency. Moreover, contention-aware synchronization can also improve the efficiency of global WC. The performance gain of global WC increases when combining a large batch of requests, since the fixed overhead of global WC can be amortized. Contention-aware synchronization boosts the performance of global WC by combining only IDU operations for hot KV pairs, which usually results in larger batches of operations being combined.

4.4 Put It All Together

CIDER is compatible to all memory-disaggregated systems with optimistic out-of-place modification in their index data structures [25, 27, 33, 63]. This section introduces the `SEARCH`, `INSERT`, `UPDATE`, and `DELETE` operations of CIDER in detail.

As shown in Figures 9 and 10, each operation begins by querying the index structure to locate the target data pointer (①). The final RDMA_FAA for each operation in Figure 10 is used for fault tolerance, which will be described in Section 4.6.

Search. CIDER enables lock-free `SEARCH` operations as shown in Figure 9a. The client first reads the data pointer via an RDMA_READ (①). It then performs an RDMA_READ on the memory location referenced by the pointer, where the KV pair is stored (②).

Insert. CIDER always conducts `INSERT` operations with optimistic synchronization as shown in Figure 9b. The client first writes the new KV data to a newly allocated memory region on the MN via an RDMA_WRITE (②). It then uses RDMA_CAS to atomically modify the data pointer to point to the new data (③).

Update. CIDER's optimizations primarily focus on the `UPDATE` operation. The client first selects the suitable synchronization mode based on the contention-aware synchronization scheme introduced in Section 4.3. In the optimistic mode, as shown in Figure 9b, the client writes the new KV data to a new memory region on the MN (②) and then executes an RDMA_CAS operation to atomically swap the old data pointer with the new memory address (③). If the RDMA_CAS fails, the client retries the update operation.

In the pessimistic mode, as shown in Figure 10a and 10b, the client acquires the MCS lock associated with the data pointer (②) and tries to participate in the global WC process introduced in Section 4.2. When participating in WC as the coordinator or participant, as shown in Figure 10b, the client directly returns the combined result upon lock release (③). Otherwise, as shown in Figure 10a, after

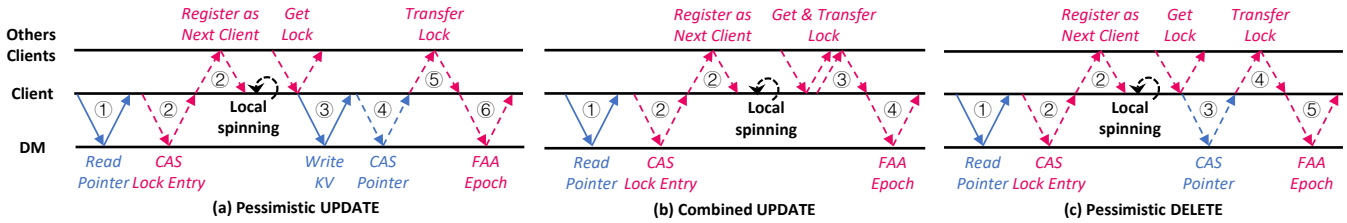

Figure 10: The workflow of the `UPDATE` and `DELETE` operations under the pessimistic mode. The solid lines indicate RDMA_READ and RDMA_WRITE, and the dashed lines indicate atomic RDMA_CAS and RDMA_FAA.

acquiring the lock, the client conducts the remote out-of-place update, *i.e.*, writes the new KV data (③) and performs an RDMA_CAS to update the pointer (④). After that, the client releases the lock (⑤). The client adjusts the number of credits associated with the data pointers according to local information, *e.g.*, the retry count, to enable the client to select the better synchronization mode during the next `UPDATE` operation.

Delete. CIDER always executes `DELETE` operations with pessimistic synchronization, as shown in Figure 10c. The client first acquires the MCS lock of the data pointer and increments the version number in the lock entry via an RDMA_CAS (②). It then atomically sets the data pointer to null and increments the version number in the data pointer via an additional RDMA_CAS (③). Finally, the client releases the MCS lock (④).

4.5 Correctness

4.5.1 The correctness of global WC. There are three key aspects that affect the correctness of global WC. First, all combined operations must target the same KV pair. CIDER ensures this by never combining `INSERT` operations and always ensuring `DELETE` operations are the executors. Second, all combined operations must be concurrent operations in a small time window. CIDER ensures the correctness by guaranteeing the same time window as the local WC. The time window of global WC is defined as the period for the first client, *i.e.*, the coordinator, to successfully acquire the lock and identify the last client in the wait queue, *i.e.*, the executor, which is the same as local WC [27]. All `UPDATE` operations issued by clients outside the time window will be queued behind the executor and combined by the next executor. Finally, all operations should be combined with a correct conflict resolution scheme. Global WC adopts a last-writer-wins [28] conflict resolution similar to that of local WC. This is guaranteed by always identifying executors as the last clients in the wait queue, *i.e.*, the last writer. Operations are thus correctly combined by only writing the value of the last writer to the memory pool.

4.5.2 The correctness of contention-aware synchronization. The correctness of contention-aware synchronization can be argued by correctly synchronizing `INSERT`, `DELETE`, `UPDATE`, and `SEARCH` operations in two synchronization modes. Specifically, since IDU operations always modify data pointers with atomic RDMA_CAS, `SEARCH` operations can always safely read the correct pointer with RDMA_READ and fetch the corresponding KV pair. When executing IDU operations in both pessimistic and optimistic modes, CIDER always uses RDMA_CAS to modify the data pointer. Consequently, IDU operations in different synchronization modes can also be correctly synchronized due to the atomicity of RDMA_CAS.

4.6 Discussions

Fault tolerance. The key fault-tolerance issue of CIDER lies in the reliance on the MCS lock [20]. Deadlocks can happen on client failures since the wait queue is constructed by organizing clients into a distributed linked list. CIDER adopts the same design as ShiftLock [20] to achieve fault tolerance. Specifically, ShiftLock assumes that operations that hold locks have a maximum duration. Consequently, deadlocks can be detected and repaired whenever a client is waiting too long. Similar to ShiftLock, CIDER associates each lock with a 64-bit `Epoch`, as shown in Figure 8. When releasing a lock, clients increment the lock's `Epoch` field by one via an RDMA_FAA. Deadlock is detected if the `Epoch` field of the lock exhibits no increase over a full maximum duration while clients are waiting for acquisition. The maximum duration is a configurable parameter, and we set the value to be the same as ShiftLock. Clients then report the deadlock to the MN, and the MN handles the failure by resetting the lock.

Generality. CIDER is designed to boost memory-disaggregated KV stores with optimistic synchronization. A large amount of existing memory-disaggregated KV stores can be optimized with CIDER since optimistic synchronization is widely adopted in the literature [25–27, 33, 63]. However, the applicability of CIDER is not limited to memory disaggregated KV stores. All applications on DM that adopt optimistic synchronization with out-of-place data modification can benefit from CIDER.

Fairness. CIDER has better fairness compared with only adopting optimistic synchronization. Specifically, optimistic synchronization on DM relies on atomic RDMA_CAS to resolve conflicts. However, commercial RNICs do not guarantee fairness for RDMA_CAS operations, *i.e.*, a client can repeatedly fail to modify a data pointer. CIDER improves this by introducing MCS lock-based pessimistic synchronization, which enforces clients to perform pointer modification in a FIFO order.

Metadata overhead. CIDER introduces additional metadata overhead on both CNs and MNs. On CNs, the metadata overhead is incurred by storing `credit` and `retryRecord` associated with data pointers. Specifically, for each KV pair, the metadata overhead is 8 bytes, *i.e.*, 4 bytes for `credit` and 4 bytes for `retryRecord`. CIDER reduces this overhead by recording `credit` and `retryRecord` only for hot KV pairs. On MNs, the metadata overhead is incurred by storing lock entries associated with data pointers. Specifically, for each key-value pair, the metadata overhead totals 24 bytes, which matches the overhead in ShiftLock [20]. This consists of 8 bytes for data pointers and 16 bytes for the lock structure. This metadata overhead is considered acceptable. For reference, popular key-value

Table 1: Workload specifications.

Workload	Write Ratio (IDU)	Read Ratio (SEARCH)
write-intensive	50%	50%
read-intensive	5%	95%
write-only	100%	/

stores such as Memcached [32] also require a minimum of 31 bytes of metadata per key-value pair [19].

5 EVALUATION

5.1 Experimental Setup

Testbed. We run our experiments on 8 physical servers on CloudLab [18]. Each machine has two 36-core Intel Xeon CPUs, 256 GB DRAM, and a 100 Gbps Mellanox ConnectX-6 NIC. All machines are connected by a 100 Gbps Ethernet switch. Following previous work [26, 27, 48], we configure one machine to serve as both CN and MN to save machine resources, aligning with previous work [27, 42] in the CN-MN ratio. Since we only have access to a limited number of physical servers, we emulate a large-scale cluster environment with up to 128 CNs by assigning 4 physical CPU cores to each CN, with each core running an independent client.

Workloads. We use YCSB workloads [14] with 8-byte keys and 8-byte values [25–27, 48], as it effectively captures the highly skewed and concurrent workloads characteristic of memory-disaggregated KV stores. Similar to previous work [5, 48, 49], we generate three types of workloads, *i.e.*, write-intensive, read-intensive, and write-only, where write operations include updating existing keys or inserting a new key if it does not exist. The write-intensive workload is consistent with that described in Section 2 and 3. We did not include range query workloads since KV stores typically do not support range queries (e.g., when using hashing). The detailed workload descriptions are listed in Table 1. Unless otherwise specified, all workloads follow the Zipfian distribution with a skewness parameter of 0.99, which is representative of real-world workloads [14].

Baselines. We compare CIDER with three synchronization schemes to show the efficacy of CIDER. We apply local WC [27] to all baselines to achieve better performance.

- Optimistic synchronization (O-SYNC): O-SYNC atomically modifies data pointers after writing the new KV data.
- CAS: Synchronizing data pointer modifications with the lock proposed by the Smart-framework [37], *i.e.*, a spinlock built on RDMA_CAS with truncated exponential backoff for contention mitigation.
- ShiftLock: Use ShiftLock [20] to synchronize data pointer modifications. We disable the reader-writer lock due to its poor performance under KV workloads.

Applications. We evaluate CIDER and the baselines on three KV stores with different index structures, *i.e.*, a pointer array, *RACE* [63], and *SMART* [27], respectively. The pointer array is used to show the pure performance of data pointer modifications, where each pointer corresponds to an individual key. *RACE* and *SMART* are two memory-disaggregated KV stores, which index KV pairs using the hash table and the radix tree, respectively. They adopt the optimistic synchronization with out-of-place KV modification to support variable-length keys and values, thus benefiting from the CIDER design. For each application, we populate 60 million KV items before the evaluation and use their default configurations.

Due to space limitations, some experimental results are included in the full version of the paper [17].

5.2 Micro-Benchmarks

We first use the pointer array application as a micro-benchmark to evaluate the pure performance gain of CIDER's design.

5.2.1 Overall performance. Figures 11a and 12a show the throughput and latency of the pointer array under the write-intensive workload. Under low concurrency with fewer than 64 clients, O-SYNC performs the best since only a few retries are generated due to conflicting IDU operations. In this case, CIDER can achieve comparable performance with O-SYNC since contention-aware synchronization switches CIDER to the optimistic mode. However, under high concurrency with more than 64 clients, O-SYNC experiences a dramatic performance collapse due to the severe I/O redundancy of optimistic synchronization. In this case, CIDER scales well and achieves up to 6.7× higher throughput and 4.2× lower P99 latency compared with O-SYNC. This is because CIDER switches into the pessimistic mode, where the MCS lock reduces redundant I/O by queuing conflicting IDU operations in a global wait queue and performing the operations sequentially. Furthermore, CIDER outperforms ShiftLock by up to 2.0× in throughput and 1.9× in P99 latency since global WC further reduces the inter-node redundant I/Os, which achieves a higher combining efficiency than the local WC design. Finally, CAS outperforms O-SYNC when the number of clients exceeds 384, indicating the inefficiency of optimistic synchronization under high-concurrency workloads. However, it is inferior to ShiftLock and CIDER because it still incurs I/O redundancy due to unsuccessful retries.

Figures 11b and 12b show the throughput and latency under the read-intensive workload. All baselines and CIDER exhibit a similar performance in throughput. CIDER's latency is comparable to O-SYNC, as both approaches utilize optimistic synchronization without additional lock maintenance overhead. In contrast, ShiftLock and CAS incur two additional RTTs to 5% of write requests and thus increase the P99 latency by up to 2.1×. The results verify the effect of the contention-aware synchronization of CIDER.

Figures 11c and 12c show the throughput and tail latency under the write-only workload. The results are similar, where CIDER achieves 6.5×, 4.1×, 2.0× higher throughput and 6.1×, 6.3×, 1.5× lower P99 latency compared with O-SYNC, CAS and ShiftLock.

As shown in Figure 13, we evaluate how workload skewness affects the performance of the pointer array under the write-intensive workload. CIDER performs best under both the uniform workload and highly skewed workloads. O-SYNC shows a good performance in the uniform workload while having the poorest performance in highly skewed workloads, *i.e.*, when the skewness is larger than 0.8. This is because the I/O redundancy issue becomes more severe under highly skewed workloads. ShiftLock and CAS perform better than O-SYNC in highly skewed workloads, since the pessimistic synchronization avoids I/O retries. However, they perform worse in the uniform workload due to the overhead of lock operations.

Figure 14 quantifies the ratio of requests using pessimistic synchronization under write-intensive workloads. Ideally, requests with severe contention conflicts, *i.e.*, requests whose retry count

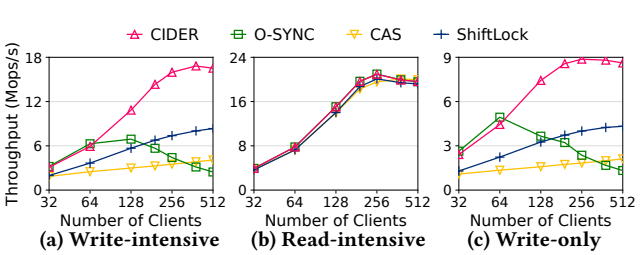

Figure 11: The throughput comparison on a pointer array.

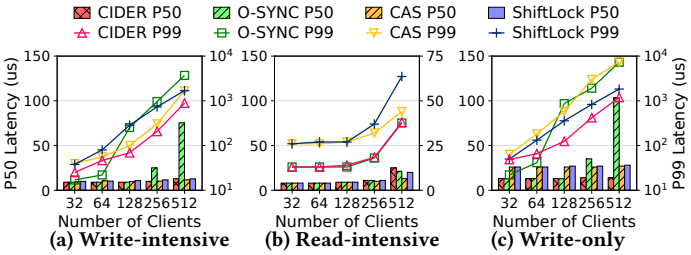

Figure 12: The latency comparison on a pointer array.

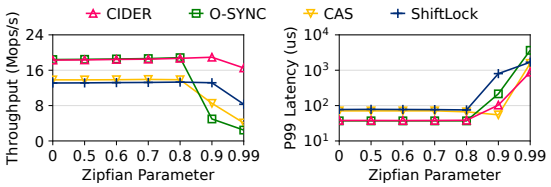

Figure 13: The throughput and latency of CIDER and baseline approaches as a function of the skewness factor (larger values mean more skew).

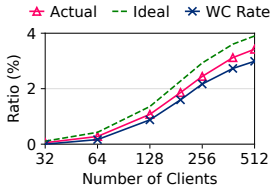

Figure 14: The analysis of the ratio of pessimistic synchronization.

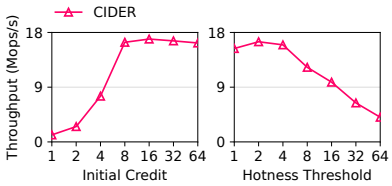

Figure 15: The impact of contention-aware synchronization parameters on throughput.

exceeds HOTNESS_THRESHOLD, should adopt pessimistic synchronization. We define the proportion of such requests as the ideal pessimistic ratio, which is 4% with 512 clients. Among these requests, CIDER can accurately identify 88% as suitable for pessimistic synchronization. Among requests using pessimistic synchronization, 87% are combined, effectively reducing redundant operations.

5.2.2 Parameter Selection. Through experiments with 512 clients under our write-intensive workload, we justify our parameter selection, *i.e.*, INITIAL_CREDIT as 36 and HOTNESS_THRESHOLD as 2. The left part of Figure 15 indicates that a small INITIAL_CREDIT causes the transition to code keys to become too sensitive. Setting the values > 8 stabilizes the throughput. The right part of Figure 15 demonstrates that a higher HOTNESS_THRESHOLD imposes stricter criteria for the transition to hot keys, shifting system behavior closer to optimistic synchronization. Setting the threshold to 2 achieves the optimal.

5.3 End-to-End Evaluation

5.3.1 RACE. RACE [63] is a memory-disaggregated KV store that indexes KV pairs with a lock-free hash table. *RACE* employs optimistic synchronization by using RDMA_CAS to modify data pointers. We integrate CIDER into *RACE* by first associating each slot entry with a lock entry to enable *RACE* to support pessimistic synchronization.

Performance under write-related workloads. Figures 16a and 17a show the throughput and latency of *RACE* under the write-intensive workload. Compared with O-SYNC, CAS and ShiftLock, CIDER brings 5.1×, 3.0×, 1.5× higher throughput and 12.4×, 6.4×, 6.2× lower P99 latency to *RACE*, respectively. Even with local WC to reduce redundant I/Os, RACE still exhibits throughput degradation beyond 128 clients and performs worse under high contention. This limitation stems from cross-CN concurrent requests that generate excessive redundant retry I/O, which cannot be eliminated by local WC. The throughput improvement of CIDER is less significant for *RACE* (5.1×) compared with that of the pointer array (6.7×). This is because *RACE* requires additional RDMA_READs to fetch remote hash buckets from the MN, making it bandwidth-bound. Compared with ShiftLock, CIDER exhibits a 1.9× increase in P50 latency with 512 clients, because requests using optimistic synchronization experience higher contention.

Figures 16c and 17c show the results under the write-only workload. Compared with O-SYNC, CAS and ShiftLock, CIDER brings 6.9×, 4.1×, 2.0× higher throughput and 7.6×, 7.9×, 1.9× lower P99 latency to *RACE*. Besides, compared with ShiftLock, CIDER achieves a 1.9× reduction in P99 latency on *RACE*, which is less than the reduction under the write-intensive workload (6.2×). This is because the write-only workload generates higher concurrency, resulting in a longer wait queue in the global WC as well as more RTTs to unlock participants along the queue.

Performance under the read-intensive workload. All methods exhibit comparable throughput and latency, as seen in Figures 16b and 17b. For O-SYNC, the 5% write ratio is insufficient to cause an IOPS bottleneck. For CAS and Shiftlock, RACE's two-choice design introduces higher read overhead across the 95% reads, masking the lock overhead from the 5% writes.

5.3.2 SMART. We also integrate CIDER into *SMART* [27], an advanced KV store on DM that uses the adaptive radix tree to index KV pairs. To support variable-length KV data, *SMART* adopts optimistic synchronization with out-of-place updates.

Performance under write-related workloads. Figures 18a and 19a show the performance of *SMART* with CIDER under the write-intensive workload. With O-SYNC, *SMART* experiences a performance collapse with more than 128 clients due to the redundant retries on modifying data pointers. On the contrary, *SMART* indexes with CAS Lock, ShiftLock, and CIDER scale well as the number of clients grows. Compared with O-SYNC, CAS, and ShiftLock, CIDER brings 6.6×, 2.8×, 2.0× higher throughput and 13.8×, 2.8×, 4.7× lower P99 latency to *SMART*, respectively, since CIDER reduces more redundant I/Os through global WC rather than local WC. Figures 18c and 19c show the performance under the write-only

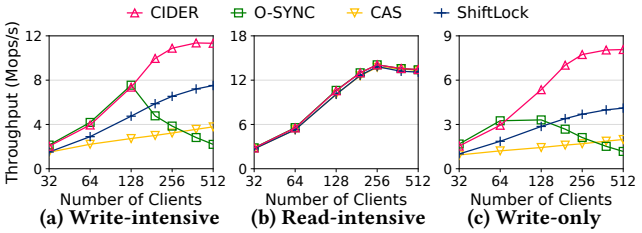

Figure 16: The end-to-end throughput on *RACE*.

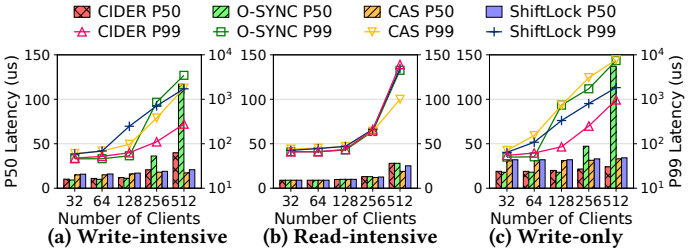

Figure 17: The end-to-end latency on *RACE*.

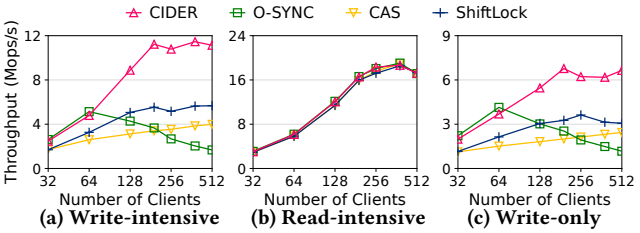

Figure 18: The end-to-end throughput on *SMART*.

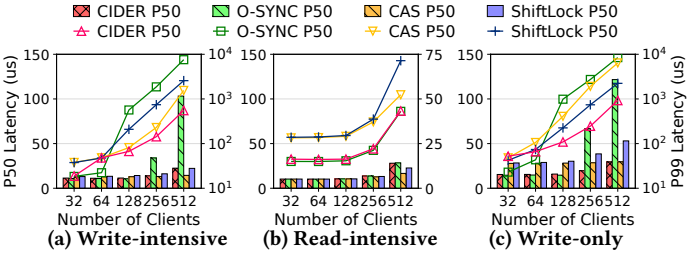

Figure 19: The end-to-end latency on *SMART*.

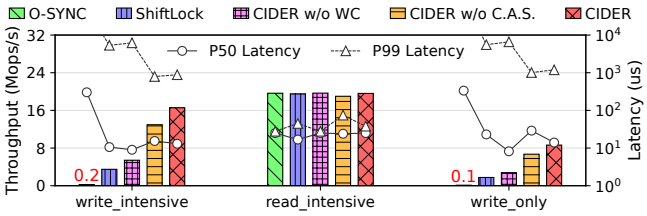

Figure 20: The factor analysis of overall performance on CIDER. "C.A.S." represents contention-aware synchronization.

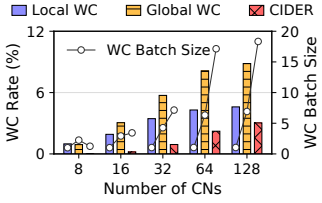

Figure 21: The efficiency comparison of different WC mechanisms.

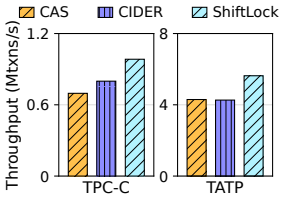

Figure 22: Performance under transactions.

workload. The evaluation results are similar, where CIDER outperforms O-SYNC, CAS and ShiftLock by up to 5.7×, 2.7×, and 2.2× in throughput, respectively.

The throughput gains brought by CIDER under write-only workloads are lower for *SMART* (5.7×) than for the pointer array (6.5×). This is because local WC provides more benefits in tree indexes than in the pointer array. Specifically, local WC can reduce intra-node redundant I/Os not only for data modifications but also for tree traversals. Since the global WC relies on the MCS lock, it can only reduce redundant I/Os on the memory region protected by the lock. Thus, it cannot combine tree traversals like the local WC. Additional tree traversal occurs during write operations, consequently imposing more significant performance overhead under write-only workloads. However, CIDER still performs the best compared with baselines since its capability of reducing inter-node I/O redundancy.

Performance under the read-intensive workload. Figures 18b and 19b show the performance of *SMART* under the read-intensive workload. The results are similar to those on a pointer array.

5.4 Factor Analysis for CIDER Design

Finally, we investigate how our design improves performance in terms of throughput, latency, and the internal WC efficiency.

5.4.1 Throughput and latency. Figure 20 presents the factor analysis for the techniques in CIDER. We disable local WC for both O-SYNC and ShiftLock to show the performance gain purely introduced by global WC. The experiment is conducted on a pointer array using 512 clients. Without loss of generality, we discuss the performance improvement under the write-intensive workload.

+ Contention-aware synchronization (CIDER w/o WC). Compared with O-SYNC, contention-aware synchronization achieves a 22× improvement in throughput as it prevents redundant retries on hot keys. Compared with ShiftLock, we achieve 1.6× higher throughput, owing to reduced operational overhead on cold keys. However, since the P99 latency is dominated by highly contended hot keys, no significant latency optimization is observed.

+ Global write combining (CIDER w/o C.A.S.). Compared with ShiftLock, the global WC improves throughput by 3.7× and reduces P99 latency by at least 5.4× since it significantly reduces redundant I/Os by combining `UPDATE` operations, alleviating IOPS bottleneck on the MN. The P50 latency remains unaffected, as the majority of requests do not induce severe I/O redundancy and therefore do not benefit from the global WC technique.

CIDER. Integrating both global WC and contention-aware synchronization, CIDER demonstrates superior performance compared with using either technique in isolation.

5.4.2 WC efficiency. We evaluate the *WC rate* and the *WC batch size* of local WC, global WC, and CIDER in Figure 21. The *WC rate* is defined in Section 2.3, *i.e.*, the proportion of IDU operations that are combined by WC techniques. The *WC batch size* is defined as the average number of requests in each combined batch.

In terms of the WC rate, global WC has a 1.9× higher WC rate than local WC, demonstrating a higher upper bound for combining efficiency. However, despite having the highest WC rate, the throughput of global WC is still less than CIDER since its small average WC batch size makes the communication overhead of conducting WC higher for clients. CIDER can achieve a larger average WC batch size thanks to the contention-aware synchronization. Specifically, CIDER employs optimistic synchronization for cold keys to minimize locking overhead, while preserving global WC batching for hot keys, where the benefits are more substantial, improving the actual combining efficiency.

5.5 Distributed Transactions

We evaluate the performance of CIDER in a transactional system using the TPC-C [44] and TATP [1] benchmarks. Our setup follows the same parameter configurations as ShiftLock [20]. Specifically, we use six servers, *i.e.*, one lock server and five client servers, each hosting 64 clients. The system implements the typical two-phase locking (2PL) protocol, where clients acquire all required locks before execution and release them afterward. Clients simulate transaction execution via busy-waiting.

We disable CIDER's contention-aware synchronization and the WC techniques as they violate the 2PL and transaction atomicity, respectively. We compare against two baselines: CAS lock and ShiftLock [20], which implements reader-writer locks.

The left part of Figure 22 shows TPC-C results. Due to the adoption of the MCS lock, CIDER reduces redundant I/O compared with CAS lock, achieving 1.1× higher throughput. ShiftLock achieves 1.2× higher throughput than CIDER since its reader-writer lock design supports concurrent reads. The right part of Figure 22 presents TATP results. With lower contention, CAS lock and CIDER perform similarly. ShiftLock achieves 1.3× higher throughput due to its reader-writer lock design.

6 RELATED WORK

Disaggregated Memory. DM is a next-generation data center architecture that is widely discussed in both academia and industry. Existing works on DM can be categorized into DM systems and DM applications. *DM systems* focus on achieving high-performance and transparent execution of applications on memory-disaggregated data centers. Existing works span multiple levels, including hardware design [21, 23, 50], operating systems [3, 8, 41, 45, 62], and software runtimes [9, 10, 29, 38, 46, 47, 56]. *DM applications* refer to a bottom-up approach that builds native applications directly over memory-disaggregated data centers. Many applications have been ported to DM, *e.g.*, cache systems [42, 60], transaction systems [58, 59], databases [22, 61], KV stores [24–27, 43, 48, 63].

The most closely related work is Smart-framework [37], an I/O optimization framework on DM. It improves throughput via thread-aware resource allocation and reduces CAS retry overhead with adaptive backoff. Unlike it, CIDER focuses on addressing the redundant I/Os incurred by optimistic synchronization with the global WC and contention-aware synchronization.

Memory-Disaggregated Key-Value Stores. Network I/O is the key bottleneck for memory-disaggregated KV stores. Existing approaches focus on improving the I/O efficiency of KV stores on DM in a bottom-up manner, *i.e.*, by tailoring data structures and algorithms to reduce I/O sizes and numbers between CN and MNs. Specifically, RACE [63] is an extendible hash table with a lock-free concurrency control scheme. SMART [27] proposes a radix tree design to avoid the read amplifications of B+ trees. It further presents the read-delegation and write-combining technique to reduce redundant I/Os on DM. CHIME [26] is a hybrid index combining B+ trees and hopscotch hashing to reduce read amplifications of B+ trees. FUSEE [43] adopts a two-level memory management technique, reducing frequent remote allocation I/Os. Different from these approaches, CIDER is a general optimization method that can be applied to a large body of index data structures. All index data structures that employ optimistic synchronization with out-of-place data modification can be optimized with CIDER. Hence, CIDER is orthogonal to these data structures and algorithm designs.

RDMA-Based Lock Management. RDMA has attracted increasing research attention in terms of distributed lock management, which can be classified into two types, *i.e.*, centralized and decentralized lock management. *Centralized lock management* [55, 57] relies on a central server for granting locks, which is unfriendly to DM due to the limited compute capability at the memory side. *Decentralized lock management* [16, 35, 51, 52, 54] leverages one-sided RDMA verbs to bypass the CPU bottleneck.

Existing memory-disaggregated KV stores typically conduct lock acquisitions via RDMA_CAS with a fail-and-retry strategy [25–27, 48, 58]. The CAS-based lock will rapidly saturate the limited IOPS upper bound of memory-side RNICs, resulting in poor scalability. ShiftLock [20] proposes an RDMA-based MCS lock design to address this issue, which can also be applied to DM. Different from existing approaches, CIDER proposes a global WC design on top of the MCS lock to further reduce redundant data modifications.

7 CONCLUSIONS

In this paper, we identify that current memory-disaggregated KV stores face significant performance bottlenecks due to redundant I/O operations when synchronizing concurrent data accesses. This issue stems from a fundamental mismatch between their optimistic synchronization schemes and the highly contended workloads prevalent in these systems. To address this, we propose to adopt pessimistic synchronization strategies to enhance performance. Based on this idea, we design and implement CIDER, a compute-side I/O optimization framework, with two key techniques, *i.e.*, global write combining and contention-aware synchronization. Our evaluation demonstrates that CIDER significantly improves the throughput of leading memory-disaggregated storage systems by up to 6.6× under the write-intensive workload.

ACKNOWLEDGMENTS

We sincerely thank our anonymous shepherd and reviewers for helping us improve our paper. This work is supported by the National Natural Science Foundation of China (Project No. 62472101) and the Open Fund of PDL (Project No.WDZC20245250106). Jiacheng Shen and Xin Wang are corresponding authors.

REFERENCES

[1] 2025. TATP Benchmark. https://tatpbenchmark.sourceforge.net/. Accessed: 2025.

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. uBFT: Microsecond-Scale BFT using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 862–877. https://doi.org/10.1145/3575693.3575732

[3] 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, April 27-30, 2020*. ACM, 14:1–14:16. https://doi.org/10.1145/3342195.3387522

[4] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K. Aguilera. 2023. Logical Memory Pools: Flexible and Local Disaggregated Memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks, HotNets 2023, Cambridge, MA, USA, November 28-29, 2023*. ACM, 25–32. https://doi.org/10.1145/3626111.3628201

[5] 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, August 7-10, 2023*. ACM, 695–704. https://doi.org/10.1145/3605573.3605576

[6] InfiniBand Trade Association. Accessed: 2025. Enabling the Modern Data Center – RDMA for the Enterprise. https://www.infinibandta.org.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[8] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering OS memory optimizations in the presence of disaggregated memory. In *ISMM '22: ACM SIGPLAN International Symposium on Memory Management, San Diego, CA, USA, 14 June 2022*. ACM, 1–14. https://doi.org/10.1145/3520263.3534650

[9] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 79–92. https://doi.org/10.1145/3445814.3446713

[10] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 77–95. https://www.usenix.org/conference/osdi24/presentation/chen-lei

[11] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

[12] Dah-Ming Chiu and Raj Jain. 1989. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Networks* 17 (1989), 1–14. https://doi.org/10.1016/0169-7552(89)90019-6

[13] CXL Consortium. Accessed: 2025. Compute Express Link. https://www.computeexpresslink.org.

[14] 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, June 10-11, 2010*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[15] NVIDIA Corporation. Accessed: 2025. Advanced Transport. https://docs.nvidia.com/networking/display/ofedv502180/advanced+transport.

[16] Ananth Devulapalli and Pete Wyckoff. 2005. Distributed Queue-Based Locking Using Advanced Network Features. In *34th International Conference on Parallel Processing (ICPP 2005), 14-17 June 2005, Oslo, Norway*. IEEE Computer Society, 408–415. https://doi.org/10.1109/ICPP.2005.34

[17] Yuxuan Du, Xuchuan Luo, Xin Wang, Yangfan Zhou, and Jiacheng Shen. 2026. CIDER: Boosting Memory-Disaggregated Key-Value Stores with Pessimistic Synchronization. arXiv:2604.03007 [cs.DC] https://arxiv.org/abs/2604.03007

[18] 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 *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[19] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. USENIX Association, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan

[20] Jian Gao, Qing Wang, and Jiwu Shu. 2025. ShiftLock: Mitigate One-sided RDMA Lock Contention via Handover. In *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*. USENIX Association, 355–372. https://www.usenix.org/conference/fast25/presentation/gao

[21] 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, 28 February 2022 - 4 March 2022*. ACM, 417–433. https://doi.org/10.1145/3503222.3507762

[22] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

[23] 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, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 488–504. https://doi.org/10.1145/3477132.3483561

[24] 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. https://www.vldb.org/pvldb/vol15/p4023-lee.pdf

[25] 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, February 21-23, 2023*. USENIX Association, 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

[26] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*. ACM, 110–126. https://doi.org/10.1145/3694715.3695959

[27] 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, July 10-12, 2023*. USENIX Association, 553–571. https://www.usenix.org/conference/osdi23/presentation/luo

[28] N.A. Lynch and A.A. Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 272–281.

[29] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. 2024. DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 97–115. https://www.usenix.org/conference/osdi24/presentation/ma-haoran

[30] Hasan Al Maruf, Yuhong Zhong, Hongyi Wang, Mosharaf Chowdhury, Asaf Cidon, and Carl A. Waldspurger. 2023. Memtrade: Marketplace for Disaggregated Memory Clouds. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2 (2023), 41:1–41:27. https://doi.org/10.1145/3589985

[31] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65. https://doi.org/10.1145/103727.103729

[32] Memcached Development Team. 2025. Memcached: a distributed memory object caching system. https://memcached.org/. Accessed: 2025.

[33] Xinhao Min, Kai Lu, Pengyu Liu, Jiguang Wan, Changsheng Xie, Daohui Wang, Ting Yao, and Huatao Wu. 2024. SepHash: A Write-Optimized Hash Index On Disaggregated Memory via Separate Segment Structure. *Proc. VLDB Endow.* 17, 5 (2024), 1091–1104. https://www.vldb.org/pvldb/vol17/p1091-lu.pdf

[34] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. 2021. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 212–227. https://doi.org/10.1145/3477132.3483576

[35] Sundeep Narravula, A. Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. 2007. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*. IEEE Computer Society, 583–590. https://doi.org/10.1109/CCGRID.2007.58

[36] 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, April 23-26, 2018*. ACM, 16:1–16:12. https://doi.org/10.1145/3190508.3190537

[37] 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, 27 April 2024- 1 May 2024*. ACM, 351–367. https://doi.org/10.1145/3617232.3624857

[38] 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, Virtual Event, November 4-6, 2020*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[39] Salvatore Sanfilippo and Redis Ltd. 2025. Redis. https://redis.io. Accessed: 2025.

[40] Michael L. Scott. 2013. *Shared-Memory Synchronization*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00499ED1V01Y201304CAC023

[41] 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, October 8-10, 2018*. USENIX Association, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[42] 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, October 23-26, 2023*. ACM, 675–691. https://doi.org/10.1145/3600006.3613144

[43] 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, February 21-23, 2023*. USENIX Association, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[44] Transaction Processing Performance Council. 2025. TPC Benchmark C (TPC-C). https://www.tpc.org/tpcc/. Accessed: 2025.

[45] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. 2022. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 352–367. https://doi.org/10.1145/3492321.3519569

[46] 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, Virtual Event, November 4-6, 2020*. USENIX Association, 261–280. https://www.usenix.org/conference/osdi20/presentation/wang

[47] 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, July 11-13, 2022*. USENIX Association, 35–53. https://www.usenix.org/conference/osdi22/presentation/wang

[48] 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, June 12 - 17, 2022*. ACM, 1033–1048. https://doi.org/10.1145/3514221.3517824

[49] Qing Wang, Youyou Lu, and Jiwu Shu. 2025. Designing an Efficient Tree Index on Disaggregated Memory. *Commun. ACM* 68, 05 (2025), 92–100. doi: 10.1145/3709647

[50] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*. USENIX Association, 277–292. https://www.usenix.org/conference/fast21/presentation/wang

[51] 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, October 8-10, 2018*. USENIX Association, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[52] 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, October 4-7, 2015*. ACM, 87–104. https://doi.org/10.1145/2815400.2815419

[53] Juncheng Yang, Yao Yue, and KV Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 191–208.

[54] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 1571–1586. https://doi.org/10.1145/3183713.3196890

[55] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*. ACM, 126–138. https://doi.org/10.1145/3387514.3405857

[56] Daniel Zahka and Ada Gavrilovska. 2022. FAM-Graph: Graph Analytics on Disaggregated Memory. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 81–92. https://doi.org/10.1109/IPDPS53621.2022.00017

[57] Hanze Zhang, Ke Cheng, Rong Chen, and Haibo Chen. 2024. Fast and Scalable In-network Lock Management Using Lock Fission. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 251–268. https://www.usenix.org/conference/osdi24/presentation/zhang-hanze

[58] 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, July 10-12, 2024*. USENIX Association, 801–819. https://www.usenix.org/conference/osdi24/presentation/zhang-ming

[59] 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, February 22-24, 2022*. USENIX Association, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[60] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2021. Redy: Remote Dynamic Memory Cache. *Proc. VLDB Endow.* 15, 4 (2021), 766–779. https://www.vldb.org/pvldb/vol15/p766-zhang.pdf

[61] Qizheni Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p6-zhang-cidr20.pdf

[62] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1345–1359. https://doi.org/10.1145/3514221.3517856

[63] 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, July 14-16, 2021*. USENIX Association, 15–29. https://www.usenix.org/conference/atc21/presentation/zuo