Cache Coherence Over Disaggregated Memory

Ruihong Wang
Purdue University
wang4996@purdue.edu

Jianguo Wang
Purdue University
csjgwang@purdue.edu

Walid G. Aref
Purdue University
aref@purdue.edu

ABSTRACT

Disaggregating memory from compute offers the opportunity to better utilize stranded memory in cloud data centers. It is important to cache data in the compute nodes and maintain cache coherence across multiple compute nodes. However, the limited computing power on disaggregated memory servers makes traditional cache coherence protocols suboptimal, particularly in the case of stranded memory. This paper introduces SELCC; a Shared-Exclusive Latch Cache Coherence protocol that maintains cache coherence without imposing any computational burden on the remote memory side. It aligns the state machine of the shared-exclusive latch protocol with the MSI protocol, thereby ensuring both atomicity of data access and cache coherence with sequential consistency. SELCC embeds cache-ownership metadata directly into the RDMA latch word, enabling efficient cache ownership management via RDMA atomic operations. SELCC can serve as an abstraction layer over disaggregated memory with APIs that resemble main-memory accesses. A concurrent B-tree and three transaction concurrency control algorithms are realized using SELCC's abstraction layer. Experimental results show that SELCC significantly outperforms RPC-based protocols for cache coherence under limited remote computing power. Applications on SELCC achieve comparable or superior performance over disaggregated memory compared to competitors.

PVLDB Reference Format:
Ruihong Wang, Jianguo Wang, and Walid G. Aref. Cache Coherence Over Disaggregated Memory. PVLDB, 18(9): 2978 - 2991, 2025.
doi:10.14778/3746405.3746422

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ruihong123/SELCC.

1 INTRODUCTION

Memory disaggregation has emerged as a significant trend in cloud databases in both academia [20, 27, 28, 38, 41, 47, 48, 53] and industry [8, 44, 49]. An important motivation behind disaggregated memory is to utilize the substantial amounts of stranded memory [14, 15, 31, 36, 46] in cloud data centers. Stranded memory refers to memory that is inaccessible due to all the available cores being allocated to virtual machines [46]. Memory disaggregation addresses this issue by accessing the stranded memory via high-speed networks, physically decoupling the memory resources from compute servers. By establishing disaggregated memory over stranded memory, cloud providers can significantly enhance memory utilization and reduce the total cost of ownership (TCO).

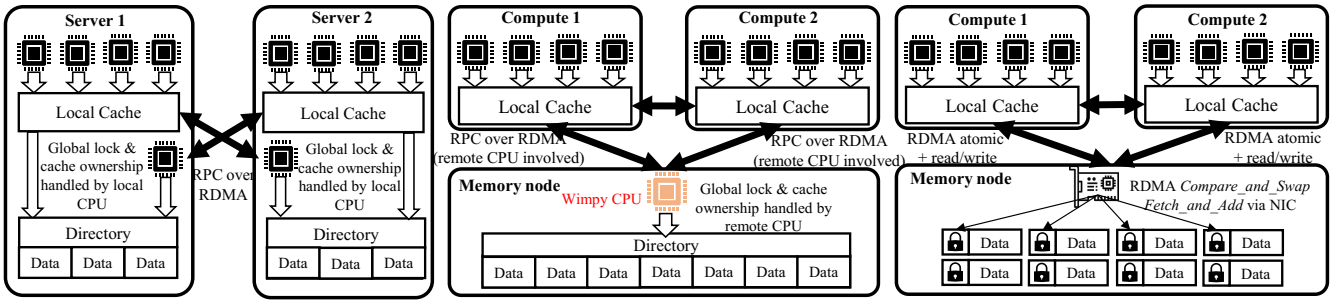
One-Sided RDMA: Key to Efficient Memory Disaggregation. The unique feature of disaggregated memory is that memory nodes have *very limited* computing power. This constraint necessitates a more efficient data access method that is provided by Remote Direct Memory Access technology (RDMA), particularly one-sided RDMA operations. One-sided RDMA allows data transfer to fully bypass the CPU on remote memory, achieving low latency and minimal use of remote computing resources. In contrast, traditional RPC-based access schemes become inefficient for disaggregated memory, especially when the computing power on memory nodes is limited or, at times, nonexistent.

The Cache Coherence Problem over Disaggregated Memory. Memory disaggregation enables sharing the main memory among multiple compute nodes. This advancement drives the next generation of multi-primary architectures [24, 44] that resolve conflicts among multiple writers through disaggregated memory. The key challenge in designing multi-primary systems over disaggregated memory is *maintaining cache coherence between the compute nodes*. Given that RDMA latency is approximately 10 times slower than main-memory access, compute-side caching is necessary as it can effectively reduce these round trips by leveraging locality. However, multiple copies of data across compute nodes introduce consistency challenges, necessitating a robust software-level cache-coherence protocol to ensure data integrity.

Limitations of Existing Cache-Coherence Solutions. Existing cache-coherence protocols over RDMA, e.g., GAM [7], ScaleStore [50], and the solutions in PolarDB MP [44] and GaussDB [24] are all RPC-based protocols, which rely on the computing resources in memory nodes. As in Figure 1a, these protocols have been designed for distributed shared-memory systems, where compute and memory resources are co-located (i.e., not disaggregated). In these systems, a server can utilize abundant computing power to process cache ownership management and resolve access conflicts. However, these protocols become suboptimal when applied to disaggregated memory, particularly when the memory pool is established over stranded memory. RPC requests suffer due to the limited computing power in memory nodes (Figure 1b). Thus, there is a pressing need for a native cache coherence protocol over disaggregated memory that bypasses the CPU on the remote memory.

Challenges. This paper investigates *how to maintain cache coherence among multiple compute nodes while adhering to the one-sided access scheme between compute nodes and the disaggregated memory.* There are many challenges: (1) Managing cache ownership distributively via one-sided RDMA is extremely challenging and is fundamentally different from RPC-based protocols. Simply leveraging RDMA read and write to manage the cache directory is inefficient due to the introduced RDMA round trips. (2) Optimizing the protocol to minimize RDMA round-trips and bandwidth consumption is

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746422



(a) RPC-based CC protocols are natively designed for distributed shared memory, where CPU resources are symmetrically distributed.

(b) RPC-based CC protocols over disaggregated shared memory can be suboptimal, due to the wimpy computing power on the memory node.

(c) SELCC protocol is natively designed for disaggregated memory, which maintains cache coherence without involving the CPUs on memory nodes.

Figure 1: RPC-based cache coherence protocols vs. SELCC protocol

non-trivial. (3) Maintaining fair data access among multiple compute nodes is complicated, especially given that the new protocol bypasses the processing of remote memory.

Our Approach. This paper presents the Shared-Exclusive Latch-based Cache-Coherence protocol (SELCC), a new protocol for the cache coherence problem over disaggregated memory. By introducing lazy latch-release and invalidation messages, the one-sided RDMA shared-exclusive latch protocol can be upgraded to address the cache-coherence problem with sequential consistency. It embeds the cache ownership information into RDMA latch words, allowing both the latch and cache ownership to be managed within a single RDMA atomic operation. To optimize performance, SELCC prioritizes local ownership handover over global ownership handover, employs batched processing for cache eviction, and minimizes the bandwidth for dirty data flush back. Additionally, SELCC enhances fairness by incorporating priority into invalidation messages. In Figure 1c, SELCC operates without involving remote memory computing power, making it particularly suited for memory disaggregation. Note that in scenarios where remote memory has computing resources, they can be used for other functionalities (rather than handling cache coherence), such as operator pushdown, which can result in significant performance gains [27, 40, 42].

Contributions. The contributions of this paper are as follows. (1) We introduce SELCC, a new one-sided RDMA latch protocol that supports RDMA access atomicity and cache coherence across multiple compute nodes (Section 3–5). Compared to other cache-coherence protocols, SELCC is agnostic to whether or not CPU compute power is available at the disaggregated memory side. (2) We present an API for SELCC that serves as an abstraction layer over disaggregated memory. We demonstrate the usefulness of this API by realizing a concurrent B-tree and three transaction concurrency control algorithms (Section 6). The proposed API simplifies realizing databases over disaggregated memory. While prior research works optimize indexes and transaction engines for disaggregated memory, these efforts have been studied independently. Integrating them into a single unified database is difficult due to their differing approaches to data synchronization. With SELCC, we can build indexes and transaction management directly on the same SELCC layer without worrying about cache coherence.

Open-source. SELCC is available at https://github.com/ruihong123/SELCC (around 29,200 LOC).

2 BACKGROUND

RDMA Technology. Remote Direct Memory Access (RDMA) is a high-speed inter-memory communication method with low latency. It allows direct access to the memory of a remote node [19]. RDMA bypasses the host operating system when transferring data to avoid extra data copies. RDMA's kernel-bypassing and low-latency features make it applicable to high-performance data centers [1, 3, 8, 49]. *ibverbs* is a C++ library for RDMA programming that provides low-level implementation of RDMA primitives. There are five types of primitives in *ibverbs*: RDMA send, RDMA receive, RDMA write, RDMA read, and RDMA atomic. RDMA write, RDMA read, and RDMA atomic are one-sided RDMA primitives that directly access the remote server's memory without involving the remote server's CPU. Two-sided RDMA primitives (including RDMA send and RDMA receive) involve both sides of the compute and memory servers. RDMA atomic includes two primitives: *RDMA compare and swap* (`RDMA_CAS`) and *RDMA fetch and add* (`RDMA_FAA`). These primitives ensure the atomicity of a group of operations on data of at most 8 bytes. Additionally, `RDMA_CAS` and `RDMA_FAA` can be leveraged to implement shared-exclusive latch over RDMA (SEL), guaranteeing atomicity among RDMA reads and writes [51].

Cache-Coherence Protocols. Cache coherence is a concept in multiprocessor systems ensuring that multiple data copies in various CPU caches remain consistent [11]. In multiprocessor systems, consistency is ensured via hardware-level cache-coherence protocols. In disaggregated memory systems, hardware-level protocols are not present. Thus, a software-level cache-coherence mechanism is needed when local caches are deployed in compute nodes.

Existing cache-coherence protocols [4, 7, 9, 26, 34, 50] over RDMA have been originally designed for distributed shared memory systems, where each object has a main copy stored in its home node. These protocols maintain cache-coherence using methods similar to those for multiprocessor systems, e.g., MSI, MESI, and MOESI [11], and utilize state machines to manage different ownership types. Indexes over disaggregated memory caches only the index metadata, e.g., B-tree internal nodes or hash directory) [28, 39, 53]. These approaches are effective, but have limitations: (1) These caches are typically limited in size and cannot be adjusted to match the available local memory capacity. (2) Metadata caching is typically specific to particular data structures, limiting its generality.

Thus, there is a significant need for a general cache coherence protocol that eliminates the need for computing over remote memory.

3 SYSTEM OVERVIEW

This paper addresses optimizing DBMSs for disaggregated memory [24, 41, 44], where compute and memory are decoupled, and multiple compute nodes share a common memory pool (Figure 2). The memory pool consists of multiple memory servers. Data is addressed via an 8-byte global pointer *(NodeID, offset)*, where *NodeID* is the memory server identifier, and *offset* is the offset within server memory. In Figure 2, the disaggregated memory space is split into blocks of configurable sizes, referred to as Global Cache Lines (GCLs). GCL is the main data manipulation unit between the compute and memory nodes, and has 3 components: A one-sided global latch word (8 bytes), a user-defined header, and the data region. The global latch word is a crucial and ensures one-sided RDMA access atomicity and cache coherence. The user-defined header is application-specific, similar to page headers in traditional disk-based databases. Finally, the data region stores data objects, e.g., tuples for data tables and key-value pairs for indexes.

When a GCL is accessed, the system first checks local cache. If the GCL is not found or its ownership state is incorrect, the system attempts to acquire the corresponding RDMA latch and fetch the latest GCL using `RDMA_CAS` and `RDMA_Read` within a single RDMA round trip. If the lock acquisition fails, which indicates a conflicting copy on another compute node, invalidation messages will be initiated to force the current owner to transfer global ownership as well as the latest copy (More on this in Section 4.4). SELCC exposes a simple API to applications that allows users to bypass the complexities of RDMA programming. Many data structures and algorithms for monolithic servers can be migrated onto SELCC seamlessly (Section 6), as the RDMA access atomicity and cache coherence problem has already been resolved underneath this API.

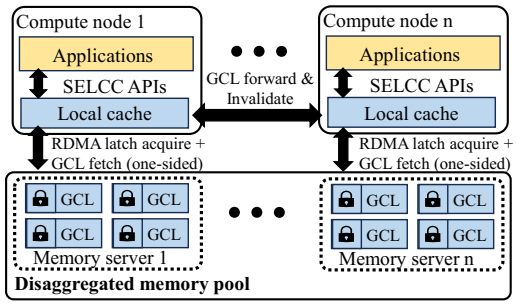


Figure 2: System overview

4 THE SELCC PROTOCOL

We introduce the SELCC Protocol; **S**hared-**E**xclusive **L**atch-based **C**ache **C**oherence protocol (SELCC). It guarantees cache coherence and atomicity for concurrent RDMA reads and writes. We follow the established design practice in [19, 51] to ensure both correctness and efficiency.

4.1 Protocol Overview

Main Idea. The SELCC protocol draws inspiration from the one-sided Shared-Exclusive Latch protocol (SEL) [51] and the Modified, Shared, Invalid protocol (MSI) [11]. SEL uses RDMA atomic operations to ensure the atomicity of RDMA accesses, enabling concurrent RDMA reads through the use of shared state. MSI is a widely adopted cache coherence protocol in multiprocessor systems. It manages the cache ownership states (Modified, Shared, and Invalid) via a state machine for consistent data reads and writes (Figure 3c).

Interestingly, we observe that the semantic meanings of MSI states are analogous to those of SEL. In SEL, the *Exclusive* state indicates a locally modified copy, the *Shared* state indicates a locally shared copy, and the *Latch Off* state indicates an invalid local copy. However, the conditions triggering state transitions differ significantly. In SEL, compute nodes eagerly release an RDMA latch once local access is complete, leading to immediate invalidation of data copies (Figure 3a). In contrast, MSI has a lazy invalidation strategy, where cache states are invalidated only upon receiving bus signals from other processors. **By mapping cache states to latch states and synchronizing SEL's state machine with that of MSI's, we can resolve the cache-coherence problem.**

To align the state machines, SELCC introduces two key mechanisms: lazy latch-release and invalidation messages (PeerRd, PeerWr), as in Figure 3b. When a compute node acquires the latch, the fetched copy is stored in local cache. Unlike traditional latch mechanisms, SELCC defers the release of the latch until either another compute node accesses the same GCL or the corresponding cache frame is evicted. This deferred latch-release allows SELCC's state machine to align with the MSI, as in Figure 3b and 3c. When a compute node fails to acquire the global latch, an invalidation message is issued, prompting the current owners to release the latch.

SELCC Flow. Data access in SELCC is divided into three phases. (1) The accessing thread searches local cache for the target Global Cache Line (GCL), leveraging access locality to minimize RDMA round trips (Section 4.2). (2) If no valid cache frame is found, the thread employs one-sided RDMA to retrieve the latest copy from disaggregated memory (Section 4.3). (3) If conflicting cache copies are found in other compute nodes, e.g., an exclusive copy in another node, the conflict is resolved by invalidation messages (Section 4.4).

Challenges. The key challenge lies in efficiently managing ownership using one-sided RDMA while ensuring it incurs no additional round trips compared to RPC-based solutions. Each phase of the process presents specific challenges: (1) How to effectively leverage metadata in local cache frames to minimize remote accesses? (2) How to efficiently acquire ownership in remote memory through one-sided RDMA operations? (3) How to correctly and efficiently transfer ownership and the latest GCL copies across compute nodes? This challenge becomes particularly complex in the presence of varying conflict scenarios, each of which requires a tailored design.

4.2 Local GCL Access via Cache

The local cache frame not only stores a copy of the Global Cache Line (GCL) but also maintains an ownership memo and a local shared-exclusive latch. The ownership memo records the acquired global ownership (Shared, Modified, or Invalid) on this data copy. The latter accesses can verify whether they are permitted without

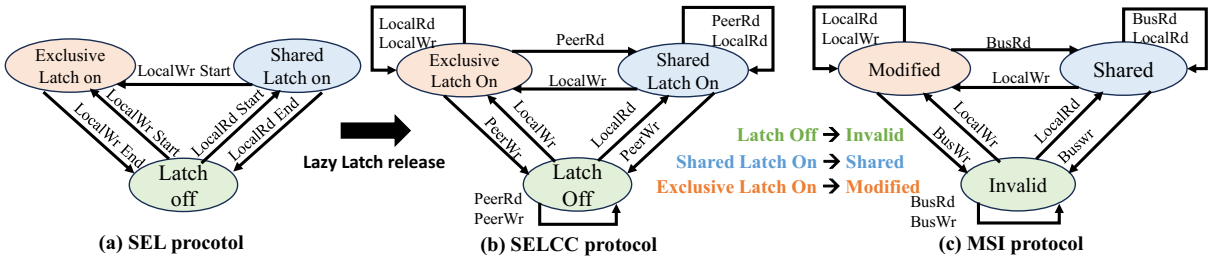


Figure 3: State machines for SELCC, SEL, and MSI protocols

involving any RDMA operation. The local latch synchronizes concurrent accesses to the cache frame between local accessing threads and invalidation message handlers.

When accessing a GCL, the thread searches local cache for the target GCL. In case of a cache miss, the thread fetches the data and acquires ownership from the disaggregated memory. For a cache hit, the thread acquires the local latch, and then checks against the ownership memo. If the ownership satisfies the current access type, the thread can directly access data via local cache. Else, RDMA operations are invoked to acquire the appropriate ownership (Section 4.3.2) and the latest GCL copy. Ownership verification rules are given in Table 1, where rows represent the type of local access, and columns represent the ownership states in the memo. Once the corresponding ownership is acquired and local cache access is completed, the accessing thread releases the local latch, while leaving the global ownership memo unchanged.

Table 1: Verification rules for data access

	Modified	Shared	Invalid / cache miss
Reader	✓	✓	Require Shared own.
Writer	✓	Require own. upgrade	Require Modified own.

4.3 Remote GCL Access via One-Sided RDMA

When no valid copy of the target GCL is present in local cache, the accessing thread must retrieve the latest GCL via one-sided RDMA and update global ownership in the disaggregated memory. In RPC-based protocols, there are message handling threads in the disaggregated memory, managing the cache ownership directory of GCLs. However, maintaining this ownership directory via one-sided RDMA is challenging, particularly when no additional RDMA round-trip is expected compared to the RPC-based solution.

4.3.1 ***Embeded Ownership in RDMA Latch Word.*** In SELCC, we propose to embed cache ownership within the RDMA latch words, allowing both cache ownership and RDMA latch to be managed via one RDMA atomic operation. As in Figure 4, the latch word is a 64-bit field, the maximum size supported by RDMA atomic operations. We divide the 64 bits into 2 parts: (1) An ID of the exclusive latch holder (6 bits), and (2) A bitmap of IDs recording the concurrent Shared latch holders (58 bits).[1]

[1]This protocol can support up to 58 compute nodes. With multi-cores on each compute node, a system using SELCC can support thousands of cores.

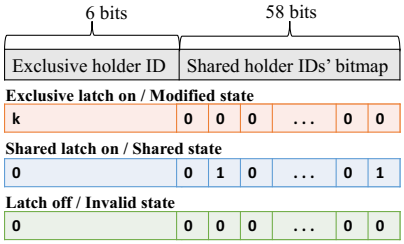


Figure 4: RDMA latch words in SELCC

This new structure allows the RDMA latch to simultaneously track both shared and exclusive latch holder IDs. For instance, if a node with ID A holds a modified copy of a GCL, say g, the latch word associated with g is represented as $(A, 000...0)$, where A is the exclusive holder ID and the reader bitmap is cleared. When two nodes, say A and B, concurrently hold shared copies of g, the latch word is represented by $(0, (1 << A)|(1 << B))$, where no exclusive holder exists, and the bit positions for A and B are set to 1 in the reader bitmap. If there is no valid cached copy in the compute nodes, the RDMA latch is set to $(0, 00..0)$. This approach has two benefits: **(1) No additional RDMA round trips are needed to maintain the cache directory, (2) Atomicity of directory changes is ensured**. If lock acquisition fails, a compute node gets the latest cache holders' IDs via the RDMA atomic operation's return value, enabling the determination of invalidation message recipients.

4.3.2 ***Acquiring Ownership via One-Sided RDMA.*** With embedded ownership in an RDMA latch word, acquiring ownership globally is equivalent to acquiring the corresponding RDMA latch from disaggregated memory. Based on the type of access and the current ownership mode in the cache frame, there are three types of ownership requests implemented via one-sided RDMA.

Acquiring "Modified" Ownership. To acquire *Modified* ownership, a writer atomically compares the latch word with $(0, 00..0)$, and swaps it with $(NodeID, 00..0)$. If the `RDMA_CAS` fails, the prior latch word is returned to the compute node that parses the shared/exclusive latch holder IDs and sends them invalidation messages.

Acquiring "Shared" Ownership. To acquire shared ownership, a reader atomically fetches the latch word and sets its own position in the bitmap using `RDMA_FAA` (with the add operand set to $(1 << NodeID)$). The reader checks the return value of `RDMA_FAA` to determine if a writer is currently holding the latch. If so, the acquisition fails, and the reader resets its bit in the bitmap using

another RDMA_FAA. Then it sends an invalidation message based on the holder ID of the exclusive latch.

Upgrading "Shared" to "Modified" Ownership. When a writer finds that the target GCL is cached in "Shared" state, the compute node needs to upgrade the global latch from shared to exclusive. First, the compute node attempts to compare and swap the global latch words from $(0, 1 << NodeID)$ to $(NodeID, 00..0)$. If two nodes simultaneously upgrade the same global latch, both nodes will continuously fail at the CAS operation. This issue is analogous to the deadlock problem for lock upgrade within a single machine, and is resolved using a typical fallback approach. After several failed attempts to upgrade the latch, SELCC resorts to releasing the shared latch and then acquiring the exclusive latch.

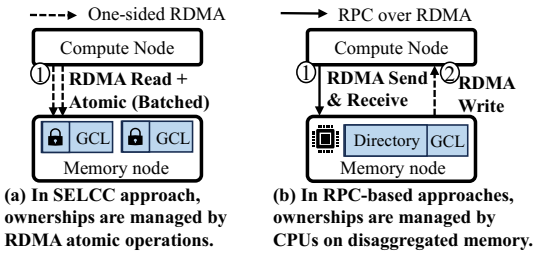


Figure 5: Different approaches for remote access

As in Figure 5a, SELCC issues RDMA read and atomic operations to atomically retrieve the latest data copy and modify the latch state (ownership). Since RDMA read and atomic operations are conducted in a batch, this procedure consumes only one RDMA round-trip. In contrast to the RPC-based solution (Figure 5b), SELCC reduces latency, as RPC-based protocols typically require two RDMA operations issued separately on both sides.

4.4 Resolving Conflicted GCL Accesses

When a thread fails to acquire the global latch for a GCL via RDMA, it issues an invalidation message (Section 5.1) to prompt the current owner to either relinquish or transfer its ownership, as well as the data copy, and, if necessary, flush back the dirty GCL. There are three types of conflict scenarios: (1) A writer on the sender side invalidates a modified copy on the receiver side. (2) A reader on the sender side invalidates a modified copy on the receiver side. (3) A writer on the sender side invalidates one or more shared copies on the receiver sides. Each scenario must be handled differently.

4.4.1 ***Case 1: A Writer on the Sender Side Invalidates a "Modified" Copy on the Receiver Side.*** Refer to Figure 6 for illustration. Node A attempts to **write** a GCL while Node B holds a copy in the "**Modified**" state. To resolve this conflict, the naive approach is that the message receivers simply release the global latch and actively flush back the dirty data to disaggregated memory. Meanwhile, the message senders repeatedly attempt to acquire the latch and retrieve the latest data from disaggregated memory (Figure 6a), following the procedure outlined in Section 4.3.2. However, this naive approach is inefficient for two reasons: (1) The retry mechanism consumes a large amount of RDMA bandwidth between compute and memory nodes. (2) Compared to RPC-based solutions (Figure 6c), it takes four RDMA round-trips, higher than three in RPC-based solutions. To optimize it, we introduce two new techniques: *Global Ownership Handover* and *GCL Forwarding*.

Global Ownership Handover allows the receiver of an invalidation message to transfer ownership directly to the requester through one atomic RDMA_FAA operation, rather than requiring the sender to repeatedly attempt latch acquisition (Figure 6b) [17, 18]. For example, if Node A (Writer) issues an invalidation message to Node B ("Modified" state), where A and B are the Node IDs, ownership can be atomically transferred from Node B to Node A by adding the value $(A - B, 00..0)$ onto the latch word via RDMA_FAA.

GCL Forwarding enables a message sender to retrieve the latest GCL copy directly from the message receiver, bypassing the need to fetch it from disaggregated memory. The sender includes its local buffer address in the invalidation message, allowing the receiver to write back the latest GCL via an RDMA write. Additionally, if GCL ownership is transferred between nodes as a "Modified" copy, the dirty GCL does not need to be flushed back to disaggregated memory, as the following owner always acquires the latest copy from the last exclusive owner in a compute node.

As in Figure 6b, SELCC requires only three RDMA round trips to handle this cases, which is the same as RPC-based solutions. Furthermore, SELCC fully bypasses remote processing, making it effective over even stranded memory. Notably, the victim of an invalidation message can immediately forward the dirty GCL to the sender's local buffer without waiting for the ownership handover round-trip (RDMA_FAA). Although this design may result in outdated invalidation messages being sent by compute nodes, outdated messages can simply be discarded upon detection.

4.4.2 ***Case 2: A Reader on the Sender Side Invalidates a "Modified" Copy on the Receiver Side.*** Refer to Figure 6 for illustration. Node A attempts to **read** a GCL while Node B holds a copy in the "**Modified**" state. This scenario is generally handled in a manner similar to Case 1, with minor adjustments for global ownership handover and GCL forwarding. As in Figure 6b, when the current owner in the "Modified" state receives an invalidation message, it atomically modifies the latch word by adding $(-B, (1 << A)|(1 << B))$ via RDMA_FAA. This operation clears the modified ownership for Node B and assigns shared ownership to both Nodes A and B. Additionally, in the same RDMA round trip, the latest GCL is flushed back to the disaggregated memory together (unlike Case 1), ensuring that future concurrent readers can acquire the latest "Shared" copy from the disaggregated memory rather than fetch it from another compute node. Finally, the latest GCL is forwarded to the sender.

4.4.3 ***Case 3: A Writer on the Sender Side Invalidates One or More "Shared" Copies on the Receiver Sides.*** Refer to Figure 7 for illustration. Node B attempts to **write** a GCL, while Nodes A and C hold copies in the "**Shared**" state. Unlike the previous case, GCL forwarding and ownership handover cannot be applied here, as atomic ownership transfer among multiple nodes with shared copies is impossible. For example (Figure 7a), consider a scenario where Node B (writer) identifies shared copies on Nodes A and C. A and C cannot collectively transfer the "Shared" ownership to B in the latch word atomically. Thus, one node must take the responsibility as a leader to manage ownership on behalf of the

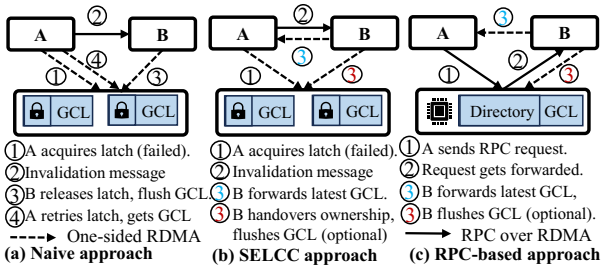


Figure 6: Different approaches to resolving conflict when a writer/reader invalidates a copy in "Modified" state.

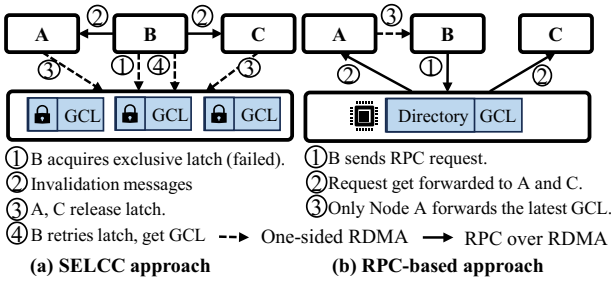


Figure 7: Different approaches to resolving conflict when a writer invalidates one/many "Shared" copies.

others. Even if Node *A* is designated as the leader, it cannot determine whether another node, e.g., *D*, has successfully acquired shared ownership during the message transmission. In these cases, *D*'s "Shared" ownership is overlooked by *A* (the leader), potentially leading to corruption of the RDMA latch word. As a result, our design in SELCC is as follows. Refer to Figure 7. The invalidation message prompts each victim node to release its shared lock while the sender continues attempting to acquire the exclusive latch. Compared to an RPC-based solution (Figure 7b), SELCC may have one extra RDMA round-trip, but given that it fully bypasses the need for remote compute power, this trade-off is acceptable.

5 OPTIMIZATIONS

We implement SELCC into the compute-side cache over disaggregated memory. The compute-side cache is a lightweight hash table with the LRU replacement policy and is sharded to support high concurrency. Several instantiation challenges remain unaddressed: (1) Efficiently implementing invalidation messages across compute nodes, (2) Optimizing cache eviction to minimize its impact on read and write operations, and (3) Avoiding latch starvation to maintain fairness among the compute nodes.

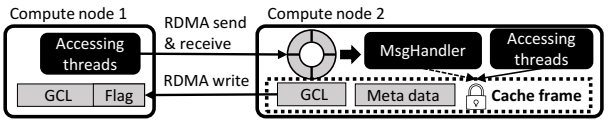


Figure 8: Invalidation message

5.1 Efficient Invalidation Messages

Invalidation messages, implemented via RPC over RDMA, play a critical role in coordinating global cache-line (GCL) ownership in SELCC. Invalidation messages contain key information, e.g., the global address of the target GCL and the case of invalidation (Section 4.4). As in Figure 8, the RPC request is sent via `RDMA_send`, to the receiver side maintaining a ring buffer to capture incoming messages via RDMA receive. The background message handlers in the receiver node process the invalidation messages by releasing/handing over the global latch. After processing, the handler sends an acknowledgment along with the latest GCL copy to a local buffer on the sender's side via RDMA write. Importantly, in SELCC, RPC is used only between compute nodes, while the communication between compute and memory layers are strictly one-sided. SELCC adheres to the compute-free design principle for memory nodes, distinguishing it fundamentally from RPC-based protocols.

The efficiency and robustness of SELCC are supported by two key design choices: (1) Prioritizing local accesses over invalidation message processing, and (2) Implementing a message drop-and-resend mechanism to handle scenarios where invalidation cannot be processed immediately.

5.1.1 ***Lower Priority for Processing Invalidation Messages.*** In scenarios where invalidation messages are handled concurrently with front-end accessing threads, synchronization is required for message handlers to manage access to cache frame metadata. A straightforward approach is to acquire the local exclusive latch before processing the invalidation messages. However, this can lead to two problems: (1) the message handler can get blocked if the local accessing thread holds the local latch for an extended period, preventing the processing of subsequent invalidation messages, and (2) such a design assigns equal priority to local accessing threads and global accessing threads for handing over the data ownership. In workloads with high contention, this can lead to frequent ownership transfers among compute nodes, significantly increasing read/write latency and generating a large volume of invalidation messages.

To address these issues, SELCC adopts a design that assigns lower priority to invalidation message handlers compared to local accessing threads. This is achieved through the use of the `try_lock`, which attempts to acquire the local latch without blocking the handler. If `try_lock` fails, the handler either defers or drops the message and proceeds to the next one. This approach ensures that global ownership transfers have lower priority than local ownership transfers within the same node, thus reducing latency and conserving RDMA bandwidth. However, prioritizing local access over global access can potentially prevent invalidation messages from taking effect under highly skewed workloads, potentially leading to global starvation on other compute nodes. The solution to this starvation problem is presented in Section 5.3.

5.1.2 ***Message Dropping and Resending Mechanism.*** Another important design is the message drop and resending mechanism. Invalidation messages can be dropped by RPC handlers under three specific conditions: (1) the cached entry has already been invalidated by other compute nodes, (2) the target cache line has been evicted, or (3) the target GCL is currently being accessed by a local thread. When a message is dropped, the receiver writes a

"dropped" flag to the end of the reply buffer in the sender side via RDMA (Figure 8). Upon receiving this flag, the message sender retries the global latch to update its view of valid cache copies and adjusts the targets of subsequent invalidation messages accordingly. To prevent network saturation due to excessive message resending, the protocol enforces a time interval $T = \frac{C \times RTT}{N_r}$ between each resend, where C is an empirical constant, RTT is the round-trip time for RDMA atomic operations, and N_r is the number of global latch retries. This interval also plays a crucial role in maintaining fairness among compute nodes, as will be discussed in Section 5.3

5.2 Efficient Cache Eviction

Cache eviction is also an important part of SELCC. In SELCC, cache eviction is managed by background threads, which monitor the length of the GCL free lists and initiate eviction when the list size falls below a predefined threshold. We employ the Least Recently Used (LRU) replacement policy to select cache frames for eviction. The cache eviction involves two main steps: releasing ownership and, if necessary, flushing back dirty data. Releasing ownership is equivalent to releasing the global latch. When releasing an exclusive latch, the compute node atomically fetches and decrements the latch word by $(NodeID, 000...0)$, while any dirty GCLs are flushed back using RDMA write within the same RDMA round trip. We do not adopt the method from [51] for releasing the exclusive latch via `RDMA_CAS`, as this can lead to spurious failures due to concurrent read lock operations, potentially resulting in livelock. For releasing the shared latch, the compute node resets its corresponding bit in the bitmap using `RDMA_FAA`.

However, having the background thread release latches and flush back dirty data for every cache frame is not efficient because: (1) each GCL eviction requires at least one RDMA round trip, and if the background thread cannot keep up with eviction requests, additional RDMA round trips may be added to the critical path of read and write over SELCC; and (2) the bandwidth consumed by flushing dirty GCLs is non-trivial. To address these challenges, we next introduce two optimizations in SELCC.

5.2.1 ***Batched Processing for GCL Eviction.*** To enhance the efficiency of GCL eviction, we process evictions in batches. The eviction worker continuously monitors the length of the GCL free list l_f and compares it against a predefined threshold L. If $l_f < L$, the eviction worker selects $L - l_f$ victims from the back of the LRU list. The selected GCLs are then grouped according to their node ID, which represents the memory node they belong to. This allows all RDMA evictions within the same group to be processed in a single batched RDMA work request. Importantly, the eviction worker does not need to wait for RDMA operation completion and can immediately proceed to the next batch of victims. However, this approach presents two technical challenges: (1) The local cache frames for dirty GCL cannot be immediately reused until the RDMA operation for flushing is complete. (2) There is a limit on the total number of outstanding RDMA operations per queue L_{out}.

To address these issues, we equip a RDMA-registered ring buffer for every queue pair connection to remote memory. The ring buffer size is set to match the maximum number of outstanding RDMA work requests per queue pair, L_{out}. Before executing RDMA operations, the eviction worker attempts to acquire an available slot from the ring buffer and copies the payload from the cache frame into this buffer slot. If no slot is available, the worker must busy-wait for the completion of previous RDMA operations. Upon receiving n completion notifications from the completion queue, the ring buffer's tail is advanced by n positions. This design ensures that the number of outstanding RDMA requests always be within the threshold L_{out}, and the cache frames can be immediately recycled once the dirty GCL is copied to the ring buffer.

5.2.2 ***Dynamic Dirty Boundaries for GCL Flushing .*** To reduce RDMA bandwidth during GCL eviction, it is essential to minimize the payload size for flushing dirty GCLs. The GCL header maintains two boundaries, d_l and d_h, which define the address range encompassing all modifications made since the GCL was fetched into the local cache. Initially, these boundaries are both set to zero, indicating that no modifications have occurred. When the first modification over GCL comes in, the d_l and d_h will be initialized as the start and end addresses of this modification. Afterwards, whenever a new modification arrives, d_l will be updated as $min(d_l, start)$ and d_h will be updated as $max(d_h, end)$. When flushing a GCL, the eviction thread checks the dirty boundaries and only writes back the data within the specified range, rather than the entire GCL, thereby reducing RDMA bandwidth consumption.

5.3 Fairness

Fairness is a significant challenge for the SELCC protocol, as it is based on a shared-exclusive spinlock. A server may experience starvation if it fails to acquire the latch repeatedly. Next, we present the root causes of latch starvation over SELCC and propose relevant solutions accordingly. Due to the two-level hierarchy of the system, two root causes of latch starvation can be identified, each requiring distinct resolution techniques.

Root cause 1: asymmetric local latch acquisition. As stated in Section 5.1.1, to minimize the volume of invalidation messages traffic, front-end accessing threads have higher priority than invalidation message handlers when acquiring the local latch. A compute node can experience global latch starvation for a particular data object if a peer compute node with a valid copy continuously receives local access requests from multiple threads for that data object. In this scenario, the local accessing threads continuously hold the local latch, causing the invalidation message handler's `try_Lock` requests to fail continuously, leading to global latch starvation.

Root cause 2: asymmetric global latch acquisition. It is unnecessary to have symmetric hardware configurations across all the compute nodes. Consequently, some compute nodes with weak CPU or network may experience latch starvation due to the low frequency of RDMA latch retries. Additionally, if there are continuous global read requests for a particular data object, a write request for that data object may struggle to acquire the exclusive latch because peer compute nodes continuously hold the shared latch, preventing the writer from obtaining the exclusive latch.

5.3.1 ***Handling Local Latch Starvation.*** To address local latch starvation, we implement a lease mechanism that forces the compute node to release the global latch when a data object has been continuously accessed by local front-end threads for an extended period. To interrupt these continuous local accesses at an appropriate time, two counters, the read access counter (R_c) and the write access

counter (W_c), are maintained in each cache entry. These counters are activated only when an invalidation message is dropped due to the ongoing local access and is deactivated when a thread acquires the latch without spinning, indicating that the data is no longer heavily accessed. The counters are incremented by 1 when a local access waits for the latch. Synthetic access times for the cache entry are calculated by $A_{times} = \frac{R_c}{P} + W_c$, where P represents the number of front-end threads on the compute node. When the synthetic access times A_{times} exceed a predefined threshold γ, the local thread proactively release the global latch and reset the counters.

5.3.2 ***Handling Global Latch Starvation.*** To handle global latch starvation for asymmetric hardware among compute nodes, we adopt a priority aging mechanism, originally devised to solve the starvation problem in CPU scheduling [35]. In SELCC, each invalidation message is assigned a priority that is positively correlated to the number of retries a compute node has conducted for a particular RDMA latch. The global ownership handover mechanism introduced in Section 4.4.1 takes starvation priority as a key factor in determining the next owner of the data. The exclusive latch holder, receiving invalidation messages from all conflicting servers, acts as a centralized decision-maker for global latch ownership transfer. During the continuous local access (Section 5.3.1), the invalidation message handler keeps receiving invalidation messages from other compute nodes and memorizes those messages as well as their priority. Upon handing over the global ownership, it is deliberately transferred to the sender with the highest priority. Furthermore, as in Section 5.1, there is a manually injected time interval between each retry for a particular latch. This interval decreases as the priority of latch acquisition increases. Thus, compute nodes with prolonged wait times are more likely to successfully acquire the latch through more frequent latch retries.

To address global write starvation induced by continuous global reads, the protocol employs spin-waiting and priority-matching mechanisms. Specifically, when a high-priority invalidation message is detected on reader nodes, a flag is set in the corresponding local cached frame, forcing subsequent global readers to spin for a predetermined duration. The spin duration is proportional to the starved writer's priority. This design creates a sufficiently large time window during which no concurrent reader holds the targeted shared latch, allowing a concurrent writer to preempt latch ownership. Although this approach mitigates the starvation, readers still achieve higher throughput compared to writers. To further balance performance between read and write, when write starvation is detected, its priority is recorded in the cache frame. This GCL cache frame can only be invalidated by a global reader with equal or greater priority. By this priority-matching mechanism, the performance of global readers and writers can be balanced given skewed workloads (See Figure 12 b).

6 SELCC AS ABSTRACTION LAYER

As cache coherence is addressed in SELCC, we observe that it can be used to build an abstraction layer to simplify building databases over disaggregated memory. This is because many existing database data structures and algorithms can be easily migrated, as the issues of RDMA access atomicity and cache coherence have already been resolved within the abstraction layer.

6.1 Programming Interface

SELCC exposes a straightforward interface to upper-level applications (Table 2). Users can allocate or deallocate global cache lines by calling `Allocate/Free`. Each data access is conducted via the local cache, and has to be protected by an SELCC latch that consists of a local latch in the cache frame and a global latch in the remote memory. The acquisition of a SELCC latch (`SELCC_SLock/SELCC_XLock`) ensures that both the local and global latch are obtained, thereby guaranteeing access atomicity and cache coherence across compute nodes. Upon acquisition, the API returns a cache handle pointing to the local copy of the target GCL. Due to the lazy latch-release introduced in Section 4.1, the release of the SELCC latch (`SELCC_SUnLock/SELCC_XUnLock`) only ensures the immediate release of the local latch while deferring the release of the global latch until another compute node accesses the same GCL. Additionally, SELCC provides APIs for global atomic operations that can be utilized to generate global timestamps or sequential numbers.

Table 2: The API of SELCC

API	Input	Output	Description
Allocate/Free	NA	gaddr	Allocate/ free a GCL
SELCC_SLock/ SELCC_XLock	gaddr	handle	Acquire the shared/exclusive permission of the target GCL.
SELLC_SUnlock/ SELCC_XUnlock	handle	NA	Release the shared/exclusive permission of the target GCL.
Atomic	gaddr, args	latch word	Conduct RDMA atomic operation on the given global address.

6.2 Consistency Model

SELCC guarantees a high level of consistency: sequential consistency. Every compute node observes operations from different compute nodes in the same sequential order [5, 21, 33] Sequential consistency is essential for a general cache framework because many applications, e.g., banking and financial services rely on strong consistency to provide reliable and accurate services to users. The primary reason for SELCC achieving sequential consistency is its latch-based design combined with synchronized invalidation messages. The latch, which is acquired before reads or writes, acts as a barrier, preventing reordering of operations within a thread. The invalidation messages mechanism ensures that any conflicting read or write can only proceed after the corresponding invalidation message has been processed. This mechanism guarantees that, before a compute node modifies data in disaggregated memory, it must invalidate all existing cache copies, forcing all subsequent reads to fetch the most up-to-date data from disaggregated memory. Consequently, when a compute node releases an SELCC exclusive latch, all other nodes can simultaneously be able to observe the write, establishing a total order of writes that is consistent across the system. By default, SELCC ensures sequential consistency. However, its consistency guarantees can be relaxed to enhance performance. For example, write operations can be buffered in a local queue and returned immediately, allowing the execution order of reads and writes to be relaxed. These buffered write requests can then be processed in batches, enabling multiplexing of invalidation messages and data transfers, which reduces RDMA bandwidth consumption. By carefully setting deadlines for processing the buffered write

requests, SELCC can achieve varying levels of consistency, ranging from sequential consistency to eventual consistency.

6.3 Index Support

By leveraging SELCC APIs, disaggregated indexes can be implemented as easily as the single-node indexes. For example, we implemented a concurrent B-link tree with SELCC in approximately 1,200 lines of code. More importantly, SELCC APIs allow developers to adapt existing single-node index codebases to disaggregated memory with just a few hundred lines of modifications. For instance, we adapted a B-tree codebase [6] from a single-server setup to disaggregated memory with only 430 lines of changes, which is less than 20% of the original codebase. The migration process involves four key steps: (1) allocate index blocks using `SELCC::Allocate`, ensuring each block is aligned with the GCL size; (2) replace local pointers with global pointers; (3) substitute mutex-based latches with SELCC latches; and (4) create an index catalog in a GCL to globally store the root node of the tree-structured index. However, not all index types can be migrated with such minimal modifications. Data structures without block organization (e.g., bw-tree [22] and skip list [29]) may require a reorganization to reduce RDMA round trips. We evaluate the concurrent B-link-tree's performance using the YCSB benchmark (Figure 14), finding it achieves competitive performance compared to state-of-the-art B-trees over shared disaggregated memory.

6.4 Transaction Support

We can migrate existing concurrency control algorithms to the disaggregated architecture by leveraging the abstraction layer enabled by SELCC as follows. (1) Tuples should be properly organized into GCLs. (2) Local shared-exclusive latches are replaced with `SELCC_XLock`/`SELCC_SLock` locks. (3) Algorithms that require monotonic timestamps utilize `Atomic` provided by SELCC API to perform RDMA Fetch-and-Add (FAA) operations on a global timestamp generator to obtain monotonically increasing timestamps.

We implement three types of algorithms over SELCC: Two-Phase Locking with no wait strategy (2PL), Timestamp Ordering (TO), and Optimistic Concurrency Control (OCC). Tuples are organized in a heap style, meaning they are placed in GCLs in chronological order of insertion. To ensure atomicity of tuple accesses, these accesses must be protected by `SELCC_XLock`/ `SELCC_SLock` locks. For two-phase locking, SELCC latches on the GCLs are reused for locking purposes, minimizing the RDMA round trips required by the transaction concurrency control. Since all transactions are executed within the same compute node via RDMA, transaction support over SELCC does not require two-phase commit protocols.

7 EXPERIMENTAL EVALUATION

.

Platform. Experiments are mostly conducted on a cluster of 16 nodes in Cloudlab [13]. The chosen instance type is c6220 which features two Xeon E5-2650v2 processors (8 cores, 2.6GHz) and 64GB (8GB X 8) of memory per node. The cluster is interconnected using 56 Gbps Mellanox ConnectX-3 FDR Network devices. Each server runs Ubuntu 18.04.1, and the NICs are driven by Mellanox OFED-4.9-0.1.7. The 16 servers are divided into two groups: 8 compute servers and 8 memory servers. Asymmetrical compute and memory resources are allocated on these two types of servers. The compute servers can utilize all the CPU cores but have a limited local cache (8GB by default). The memory agents on the memory servers can access all the memory but are restricted to a very limited number of CPU cores (1 core by default) using the `numactl` command.

7.1 Evaluating SELCC

Baselines. To show the efficiency of SELCC, we compare SELCC against three competitors over disaggregated memory: (1) ***GAM*** [7], an RPC-based cache-coherence protocol designed for distributed shared memory. We test GAM with different consistency models: total store order consistency (TSO) and sequential consistency (SEQ). (2) ***ScaleStore*** [50], state-of-the-art RPC-based protocol designed for distributed shared memory over distributed shared SSDs. (3) ***SEL*** [51], a one-sided access framework that operates without compute-side caching. While it employs the shared-exclusive latch (SEL) to ensure RDMA access atomicity, it does not have the cache coherence problem.

Benchmarks. We test the competitors by a micro-benchmark tool [7] that allows for adjustments in sharing ratios, read/write ratios, data skewness and access locality. The accesses in the micro-benchmark directly targets the global address of GCL. Each compute server issues 16 Million accesses over 24 million allocated GCLs (2KB per GCL , 48GB in total). The overall throughput with different read ratios 95% (Read intensive), 50% (Write intensive), 0% (Write only) are tested.

7.1.1 ***Evaluating the Scalability of SELCC.*** To evaluate the scalability of the SELCC protocol, we run the benchmark under a uniformly distributed workload while varying the number of compute nodes. The total number of nodes is temporarily increased to 32 for this experiment. We scale the number of memory nodes in proportion to the number of compute nodes. Furthermore, we increased the local cache size to 16GB to more clearly reveal the overhead associated with cache invalidation. We compare SELCC under various sharing ratios (*sr*), following the methodology in [7, 12, 44]. The sharing ratio (*sr*) indicates the percentage of allocated data accessible by all compute nodes, while the remainder is accessed privately. When the sharing ratio is zero, the system essentially operates as a sharding-based system over disaggregated memory.

Figure 9 shows the experimental results. The point values represent the overall throughput, while the bar values indicate the proportion of operations requiring invalidation messages. For the read-intensive workloads, SELCC demonstrates great scalability regardless of the sharing ratio, as there is very little cache coherence overhead introduced in the system. For write-intensive and write-only workloads, SELCC scalability deteriorates with increased shared data ratio (Figure 9c and d). The reason is that a higher shared data ratio increases the likelihood of two compute nodes caching the same data, resulting in a higher volume of invalidation messages. Compared to the fully partitioned SELCC (0% shared ratio), the fully shared SELCC (100% shared) shows a 37.7%/33.2% performance degradation at 16 nodes in write-intensive and write-only workloads, respectively. Compared to the single compute node deployment, the 16-node SELCC increases throughput by 10.4×/9.83×, corresponding to the write-intensive and write-only workloads.

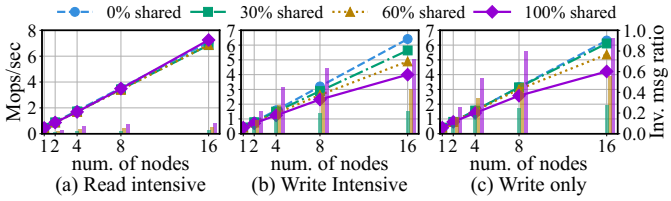


Figure 9: Scalability of SELCC over multiple compute nodes.

7.1.2 ***Evaluating the Workloads with Access Locality.*** Workloads with access locality are scenarios where the local cache provides significant benefits. To illustrate the performance benefits, we modify the uniformly distributed workload such that each operation accesses the same GCL as the preceding one with 50The benchmark is executed with 8 compute nodes, 100% sharing ratio, and varying numbers of threads across the nodes. Compared with SEL, SELCC shows significant performance gains in all workloads (Figure 10a), with improvements up to 1.85× at 64 threads. Compared with GAM (TSO), GAM (SEQ) and ScaleStore, SELCC demonstrates higher performance , achieving up to 6.31× the throughput, respectively under write-only workload. GAM exhibits limited thread scalability due to its serialized queue for all the read/write operations and ScaleStore's performance is limited by the insufficient remote computing power.

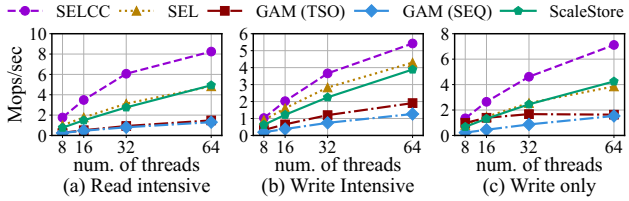


Figure 10: Performance evaluation with access locality.

7.1.3 ***Evaluating the Workloads with Access Skewness (zipf distribution).*** Workloads with access skewness represent additional scenarios where cache can achieve significant benefits. To illustrate the performance benefits of SELCC under a workload with access skewness, we run the benchmark with a Zipfian distribution, 100% sharing ratio. The skewness parameter, θ, is set to 0.99, without applying the access locality. Other parameters are configured in the same way as those in the previous subsection. For read-intensive workloads, SELCC exhibits significant performance gains, achieving throughput 3.09× over that of SEL at 64 threads. These gains result from the high cache hit ratios (60.7%) of skewed workloads (Figure 11). For write-intensive and write-only workloads, SEL shows better performance than SELCC when the thread count is low, as SELCC suffers from a large number of invalidation messages triggered by the data hotspot. As the thread count increases, SEL experiences significant performance degradation , due to the high contention in RDMA atomic operations over the data hotspot. In contrast, SELCC exhibits better thread scalability as it prioritizes local concurrency control (Section 5.1.1), shifting the bottleneck from RDMA conflict to local conflict. Finally, SELCC outperforms GAM(TSO), GAM(SEQ), and ScaleStore across most workloads, highlighting its effectiveness as a native cache coherence protocol for disaggregated memory.

To show the effectiveness of our approaches in improving fairness, we employ two experimental setups. First, we run a write-only workload with varying handover thresholds γ as introduced in Section 5.3.1. As shown in Figure 12 a, prioritizing local ownership transfer ($\gamma = Inf$) improves performance while prioritizing global ownership transfer ($\gamma = 0$) guarantees access fairness. Selecting an intermediate threshold ($\gamma = 256$) effectively balances performance and fairness. Second, we configure a single-writer, multiple-reader setup, in which one compute node executes 100% writes while the others execute 100% reads. We enable spin-waiting and priority-matching mechanisms (Section 5.3.2) one at a time to evaluate their effectiveness. We show the cumulative executed operations over time for one reader node and one writer node in Figure 12 b. When no optimization is applied, the writer experiences starvation. After enabling spin-waiting, the writer is no longer blocked by the readers, but its throughput remains substantially lower. Once we activate the priority-matching mechanism, the writer's performance becomes comparable to that of the readers, demonstrating significantly enhanced access fairness.

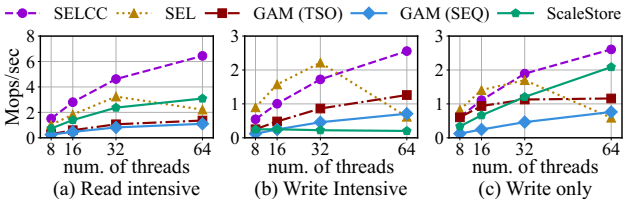


Figure 11: Performance evaluation with access skewness.

(a) Performance per node under write-only workload.

(b) Cumulative operations over time for single-writer and multiple-readers setup.

Figure 12: Performance evaluation on access fairness.

7.1.4 ***Evaluating the Workloads with Varying Remote Computing Power.*** In this subsection, we aim to demonstrate the CPU-agnostic nature of SELCC. We evaluate SELCC and ScaleStore under three different configurations: (1) Stranded remote memory, where we exhaust the computing power on memory nodes when running the memory-server processes. (2) 1 remote core, the default configuration; and (3) 8 remote cores, where there is no CPU limitation on memory nodes when running the memory-server processes. As in Figure 13, SELCC achieves a performance comparable to ScaleStore when there is no limitation on remote CPU. SELCC slightly outperforms ScaleStore with 8 remote cores in write-intensive workloads

due to its dirty data flushing optimization (Section 5.2.2). The performance gap in read-intensive workloads is due to RDMA read and atomic operations reaching bandwidth limitations faster than RDMA send, receive, or write operations, as RDMA atomic and RDMA read require more processing in the RDMA NIC[2]. Finally, the impact of remote computing power on the performance of RPC-based protocols is significant, especially under stranded memory conditions, where the throughput of ScaleStore is less than 0.3 Mops/s across all workloads.

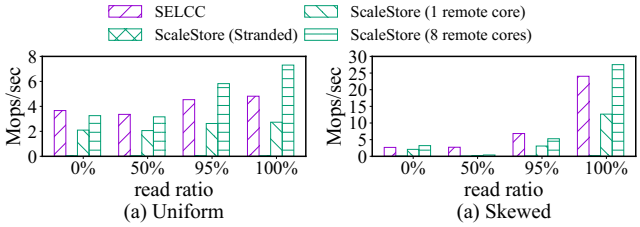


Figure 13: Performance with varied remote computing power.

7.2 Evaluating Index Support over SELCC

While the previous experiments highlight the performance advantages of SELCC as a cache coherence protocol, its efficiency in supporting applications remains uncertain. In this subsection, we construct an index following the methodology outlined in Section 6, and evaluate its performance using YCSB [10].

Baselines. Four B-tree baselines are evaluated in this experiment. The first baseline is Sherman [39], an optimized index over disaggregated shared memory. The second baseline is the B-tree over ScaleStore [50], configured the same as the one in the micro-benchmark. The third baseline is DEX [27], a sharding-based B-tree over disaggregated memory. Unlike the other shared-memory baselines, DEX employs a sharding mechanism to bypass the cache coherence problem. The final baseline is the B-tree over SEL.

Benchmarks & Configuration. We benchmark the indexes using YCSB, following methodologies established in the existing literature [27, 39, 42]. Each index is loaded with 2 billion key-value records (around 40GB) and tested under varying read ratios and data skewness ($\theta = 0.99$). The experiments are conducted over 8 compute nodes, with 8 threads per node.

7.2.1 ***Results of Uniform Workloads.*** Compared to Sherman, the B-tree over SELCC outperforms Sherman by factors up to 1.76×, because Sherman's remote synchronization requires one more RDMA round trip compared with SELCC and Sherman cannot cache leaf nodes locally. SELCC outperforms scalestore by factors up to 1.87×, showing better performance as an abstraction layer over compute-limited disaggregated memory. Finally, the B-tree over SELCC slightly loses to DEX under uniform workloads. This result is expected, as the sharding mechanism in DEX fully bypasses the cache coherence problem and includes many index-specific optimizations. Moreover, DEX has limitations when serving as an index component in a full-fledged multi-primary database due to the overhead of cross-shard transactions (See Section 7.3).

[2]This gap could be mitigated with more advanced RDMA NICs, such as ConnectX-5.

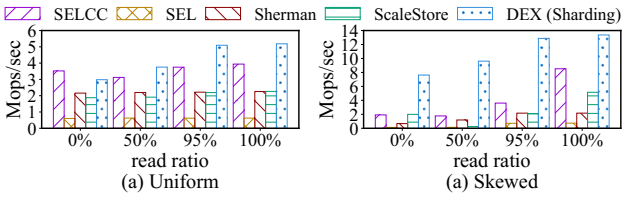


Figure 14: B-tree performance over disaggregated memory.

7.2.2 ***Results of Skewed workloads.*** The B-tree over SELCC outperforms Sherman by factors up to 4.00×, because the local cache in SELCC can hold most of the hot data. The B-tree over SEL has very limited performance under skewed workloads due to the excessive RDMA round trips required for traversing the tree. Sherman exhibits weaker performance than the B-tree over SELCC, because its leaf nodes cannot be cached locally, resulting in high RDMA atomic traffic contention over the hot spots. In contrast, SELCC can mitigate this traffic by pre-resolving conflicts in the local cache. SELCC outperform ScaleStore greatly, aligning with our micro-benchmark results in Section 7.1. DEX demonstrates extremely fast performance as it completely avoids concurrency control and caches data locally.

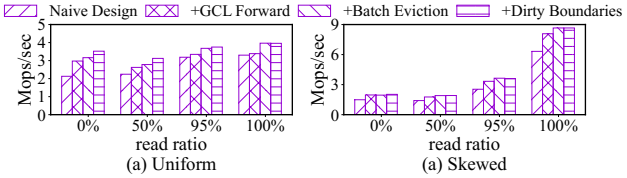


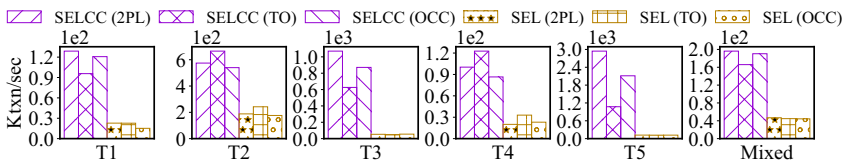
Figure 15: Ablation study

7.2.3 ***Ablation Study.*** We evaluate the impact of several optimizations from Section 4 and 5 using the YCSB benchmark with a uniform workload and varied read ratios. We start with 3 key optimizations disabled (GCL Forwarding in Section 4.4.1, Batched Eviction in Section 5.2.1, and dynamic dirty boundaries in Section 5.2.2) and then re-enable them one at a time. First, enabling GCL Forwarding results in huge improvements (40% and 17%) for write-only and write-intensive workloads. Next, we activate the Batched Eviction, which yields performance improvements across all workloads. Finally, we enable the dynamic dirty boundaries optimization for GCL flushing, which significantly boosts performance for pure-write and write-intensive workloads by 11% and 12%.

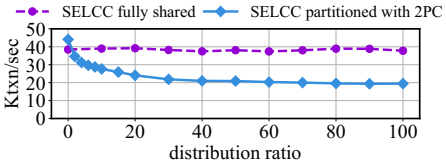
7.3 Evaluating Transaction Support over SELCC

In this subsection, we evaluate the performance of transaction engines over SELCC using the TPC-C benchmark.

Baselines. We build transactional engines using various representative concurrency control algorithms: two-phase locking (2PL) with no-wait strategy, timestamp ordering (TO), and optimistic concurrency control (OCC), following the methodology in Section 6. Also, we build transaction engines over SEL's abstraction layer to explore the benefits of cache under OLTP workloads. Additionally,



(a) Evaluation of different concurrency control algorithms over SELCC and SEL.



(b) Fully shared vs. partitioned with 2PC.

Figure 16: TPC-C benchmark results

we built a 2-Phase Commit (2PC) engine over partitioned SELCC. By comparing the performance of fully-shared SELCC against partitioned SELCC, we aim to demonstrate the advantages of fully-shared SELCC for bypassing the two-phase commit (2PC) protocol.

Benchmark & Configuration. A database is loaded with 256 warehouses, occupying approximately 64GB of disaggregated memory. The benchmark suite includes five transactions: three of them (T1, T2 and T4) contain insertions and updates[3] . The experiment is conducted in two parts. First, we evaluate SELCC against SEL using three concurrency control algorithms, with all data fully-shared. The B-tree over SELCC serves as the index for this benchmark. Write-ahead logging is disabled to clearly highlight performance discrepancies. In the second part, we compare fully-shared SELCC against partitioned SELCC using the same database setup. The transaction concurrency control algorithm is set to 2PL, and write-ahead logging is enabled to fully demonstrate the overhead of 2PC.

7.3.1 ***Results for SELCC vs. SEL.*** As in Figure 16a, concurrency control algorithms over SELCC offer significant performance benefits compared to those over SEL when handling workloads generated by TPC-C. SELCC achieves up to 24.8× throughput with read-only transactions, 7.96× with update transactions, and 4.29× in mixed scenarios. SELCC has a considerable advantage over SEL even for update transactions as there are still numerous reads on immutable data (e.g., index traversal and reading immutable tables). Also, the performance of concurrency control algorithms varies when dealing with different transactions. Algorithm TO over SELCC exhibits poor performance in read-only transactions (T3 and T5) because even read operations require updating the read timestamp, resulting in cache invalidation. However, TO outperforms the 2PL algorithm for update transactions due to its lower abort rate. Generally, OCC has slower performance than 2PL as it requires acquiring the SELCC latch for the GCL twice per tuple; once during the read phase and again during the validating phase that results in a higher volume of cache invalidation messages.

7.3.2 ***Fully-Shared SELCC vs. Partitioned SELCC.*** For partitioned SELCC, we partition the data according to warehouse IDs. T1 (New Order) is evaluated with varying distribution ratios, representing the percentage of cross-shard transactions. As in Figure 16b, partitioned SELCC outperforms fully-shared SELCC when the distribution ratio is 0. The gap between fully-shared and partitioned SELCC is not apparent due to slow log writing onto SSD, shifting the bottleneck from RDMA access to SSD writes. As the number of cross-shard transactions increases, the performance of partitioned SELCC decreases significantly. This decline is attributed mainly to communication overhead and the high cost of `fsync` during both the prepare and commit stages, despite the use of group commit to reduce overhead. In contrast, the fully-shared SELCC that bypasses 2PC, remains unaffected by the distribution ratio.

[3]T1: NewOrder, T2: Payment, T3: OrderStatus, T4: Delivery, T5: StockLevel [30].

8 RELATED WORK

As cache coherence has been covered in Section 2, this section presents additional related work.

Database systems over disaggregated memory. Approaches to database research over disaggregated memory differ significantly between academia and industry. Academic research focuses on redesigning specific database components, e.g., indexes [27, 28, 39, 42, 52, 53] and transaction concurrency control algorithms [37, 43, 45] over the disaggregated memory. SELCC converges the individual database research by providing a layer of abstraction. In contrast, industry, e.g., Alibaba PolarDB and Huawei GaussDB, conducts research in full-fledged system support over disaggregated memory [8, 23, 32, 44, 49]. They migrate the buffer pool onto disaggregated memory, achieving a higher cache hit ratio [8, 49], instant failure recovery [8, 23], elasticity resource provisioning [8], and multiple primary nodes [24, 44]. Existing multi-primary databases use RPC-based protocols to maintain cache coherence over disaggregated memory, but their performance could be constrained by the limited remote computing power.

CXL-based disaggregated memory CXL is an emerging technology for disaggregation [16, 25]. In CXL 3.0 specification [2], cache coherence is expected to be guaranteed at the hardware level. However, that coherence is maintained between the CPU caches and remote memory. This work focuses on cache coherence between the local memory in compute nodes and remote memory. SELCC will remain valuable even with CXL 3.0, as there is still a need to cache data in the local memory of compute nodes, which introduces the software-level cache coherence problem.

9 CONCLUSION

This paper addresses a key challenge for database systems over disaggregated memory: Maintaining cache coherence over disaggregated memory via one-sided RDMA. SELCC provides a disaggregated memory abstraction that facilitates further research, e.g., in indexing and transaction management. SELCC can be utilized by cloud-native databases to enable scalable multi-primary designs.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under Grant Number 2337806.

REFERENCES

[1] [n.d.]. Advancing Cloud with Memory Disaggregation, https://phys.org/news/2018-01-advancing-cloud-memory-disaggregation.html. Accessed: 2025-06-07.

[2] [n.d.]. CXL 3.1 Specification. https://computeexpresslink.org/cxl-specification/. Accessed: 2025-06-07.

[3] [n.d.]. Intel RSD. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf Accessed: 2025-06-07.

[4] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. Treadmarks: Shared Memory Computing on Networks of Workstations. *Computer* 29, 2 (1996), 18–28.

[5] Hagit Attiya and Jennifer L Welch. 1994. Sequential Consistency Versus Linearizability. *ACM Transactions on Computer Systems (TOCS)* 12, 2 (1994), 91–122.

[6] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://panthema.net/tlx, Accesed: 2025-06-07.

[7] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment (PVLDB)* 11, 11 (2018), 1604–1617.

[8] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2477–2489.

[9] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and Performance of Munin. In *Proceedings of ACM Symposium on Operating System Principles, SOSP*. 152–164.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of ACM Symposium on Cloud Computing, (SoCC)*. 143–154.

[11] David Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel Computer Architecture: a Hardware/software Approach*. Gulf Professional Publishing.

[12] Alex Depoutovitch, Chong Chen, Per-Åke Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: bringing multi-master to the cloud. *Proceedings of the VLDB Endowment (PVLDB)* 16, 12 (2023), 3488–3500.

[13] 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 *USENIX Annual Technical Conference (ATC)*. 1–14.

[14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource Packing for Cluster Schedulers. In *Proceedings of the ACM conference on SIGCOMM*. 455–466.

[15] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 649–667.

[16] Yunyan Guo and Guoliang Li. [n.d.]. A CXL-Powered Database System: Opportunities and Challenges. ([n. d.]).

[17] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009. Improving OLTP Scalability using Speculative Lock Inheritance. *International Conference on Very Large Data Bases (VLDB)* 2, 1 (2009), 479–489.

[18] Ashok M. Joshi. 1991. Adaptive Locking Strategies in a Multi-node Data Sharing Environment. In *International Conference on Very Large Data Bases (VLDB)*. 181–191.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *USENIX Annual Technical Conference (ATC)*. 437–450.

[20] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Conference on Innovative Data Systems Research (CIDR)*.

[21] Leslie Lamport. 1979. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

[22] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 302–313.

[23] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 355–370.

[24] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 3786–3798.

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

[26] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)* 7, 4 (1989), 321–359.

[27] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment (PVLDB)* 17, 10 (2024), 2603–2616.

[28] 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 *USENIX Symposium on Operating Systems Design and Implementation OSDI*. 553–571.

[29] William W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.

[30] Francois Raab. 1993. TPC-C - The Standard Benchmark for Online Transaction Processing (OLTP). In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann.

[31] Pramod Subba Rao and George Porter. 2016. Is Memory Disaggregation Feasible?: A Case Study with Spark SQL. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems, ANCS*. ACM, 75–80.

[32] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent Memory Disaggregation for Cloud-Native Relational Databases. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, Vol. 3. 498–512.

[33] Daniel Sorin, Mark Hill, and David Wood. 2022. *A primer on Memory Consistency and Cache Coherence*. Springer Nature.

[34] Robert Stets, Sandhya Dwarkadas, Nikos Hardavellas, Galen C. Hunt, Leonidas I. Kontothanassis, Srinivasan Parthasarathy, and Michael L. Scott. 1997. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the ACM Symposium on Operating System Principles, SOSP*. ACM, 170–183.

[35] Andrew S. Tanenbaum. 2009. *Modern Operating Systems, 3rd Edition*.

[36] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 18:1–18:17.

[37] Chao Wang and Xuehai Qian. 2021. RDMA-enabled Concurrency Control Protocols for Transactions in the Cloud Era. *IEEE Transactions on Cloud Computing* (2021).

[38] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 37–44.

[39] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *ACM International Conference on Management of Data (SIGMOD)*. 1033–1048.

[40] Ruihong Wang, Chuqing Gao, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2024. Optimizing LSM-based indexes for disaggregated memory. *The VLDB Journal* 33, 6 (2024), 1813–1836.

[41] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2023. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)* 16, 1 (2023), 15–22.

[42] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *International Conference on Data Engineering (ICDE)*. 2835–2849.

[43] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 233–251.

[44] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the International Conference on Management of Data*. 295–308.

[45] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The End of a Myth: Distributed Transactions can Scale. *Proceedings of the VLDB Endowment (PVLDB)* 10, 6 (2016), 685 – 696.

[46] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2022. Redy: Remote Dynamic Memory Cache. *Proceedings of the VLDB Endowment (PVLDB)* 15, 4 (2022), 766 – 779.

[47] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *Conference on Innovative Data Systems Research (CIDR)*.

[48] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020), 1568–1581.

[49] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)* 14, 10 (2021), 1900–1912.

[50] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 685–699.

[51] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 2 (2023), 131:1–131:26.

[52] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 741–758.

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