Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory

Ming Zhang, Yu Hua, and Zhijun Yang, *Wuhan National Laboratory for Optoelectronics, School of Computer, Huazhong University of Science and Technology*

https://www.usenix.org/conference/osdi24/presentation/zhang-ming

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory

Ming Zhang, Yu Hua*, Zhijun Yang
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

In modern datacenters, memory disaggregation unpacks monolithic servers to build network-connected distributed compute and memory pools to improve resource utilization and deliver high performance. The compute pool leverages distributed transactions to access remote data in the memory pool to provide atomicity and strong consistency. Existing single-versioning designs have been constrained due to limited system concurrency and high logging overheads. Although the multi-versioning design in the conventional monolithic servers is promising to offer high concurrency and reduce logging overheads, which however fails to work in the disaggregated memory. In order to bridge the gap between the multi-versioning design and the disaggregated memory, we propose Motor that holistically redesigns the version structure and transaction protocol to enable **m**ulti-versi**o**ning for fast distributed **t**ransacti**o**n processing on the disagg**r**egated memory. To efficiently organize different versions of data in the memory pool, Motor leverages a new consecutive version tuple (CVT) structure to store the versions together in a continuous manner, which allows the compute pool to obtain the target version in a single network round trip. On top of CVT, Motor leverages a fully one-sided RDMA-based MVCC protocol to support fast distributed transactions with flexible isolation levels. Experimental results demonstrate that Motor improves the throughput by up to 98.1% and reduces the latency by up to 55.8% compared with state-of-the-art systems.

1 Introduction

Memory disaggregation in modern datacenters receives extensive attentions [2, 3, 35, 46, 53, 62]. Specifically, memory disaggregation decouples the compute and memory resources from traditional monolithic servers to build independent and scalable compute and memory pools. These pools are connected via fast network (e.g., RDMA [75] or CXL [7]). A compute pool contains many powerful compute units to run tasks and small DRAM-based memory to maintain metadata. Moreover, a memory pool consists of substantial memory modules to store application data and a small number of weak compute units only for memory allocations and network interconnections [84, 86]. With the aid of efficient resource pooling, memory disaggregation significantly improves the resource utilization, elasticity, and failure isolation [65, 72].

To provide atomicity and strong consistency guarantees for applications on the disaggregated memory, the compute pool leverages distributed transactions to access remote data in the memory pool. A recent design, i.e., FORD [84], is able to run distributed transactions on the disaggregated memory. To simplify the data store in the memory pool, FORD maintains one version of each data. However, this single-versioning design limits the concurrency since the reads need to wait for the writes to become visible during transaction commit. Moreover, to guarantee atomicity, FORD writes many undo logs to back up the old data, which consumes the network bandwidth and decreases throughput.

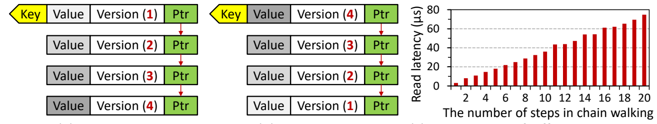
Enabling multi-versioning is expected to efficiently address the above limitations. By storing multiple versions of each data in the memory pool, the read requests are able to fetch existing versions of data rather than waiting for the writes to complete, thus improving the concurrency. Moreover, with multi-versioning, the old versions of data are retained to provide the atomicity, thus eliminating the need of writing undo logs. Prior multi-versioning based distributed transaction processing systems have been proposed in the traditional monolithic architecture [57, 64, 76]. Unfortunately, these systems are difficult to work on the new disaggregated memory architecture due to two challenges, as presented below.

1) Incompatible Transaction Protocol. Prior systems working on monolithic architecture assume that each server has strong CPUs to execute compute tasks in the transaction protocol, e.g., locking [64], validation [57], and timestamp calculation [76]. In general, a single task is not computationally expensive. However, when the number of requests increases, these tasks become substantial and frequent. The CPU in a memory pool is too weak to frequently poll massive tasks and execute them [45, 46, 66, 69, 75, 84, 86]. Therefore, legacy multi-versioning based transaction protocols are not compatible with the disaggregated memory pool.

2) Inefficient Version Structure. To store different versions of data, existing schemes leverage pointer-based structures to dynamically link the versions, called *linked chains* in this paper. In general, there are two types of the linked chains. (1) The old-to-new chain links the versions from the oldest to the newest version [10, 25, 38, 76], as shown in Fig. 1a. (2) The new-to-old chain links the versions from the newest to the oldest version [9, 32, 57, 64, 81], as shown in Fig. 1b. To read a specific version, CPU performs *chain walking* that leverages the pointers to fetch the versions one by one until the target version. In fact, the linked chains work well in monolithic servers, since each server contains enough CPUs to quickly perform chain walking in its local memory. However, the linked chains become inefficient in disaggregated memory, since all the application data are stored in the remote memory pool, which does not contain powerful CPU to execute the chain walking. As a result, the compute pool has to perform the chain walking by consuming multiple network round trips to fetch remote versions one after another until the target version, leading to high overheads. Fig. 1c shows that when increasing the number of steps in the chain walking from 1 to 20, the RDMA read latency significantly increases by 24.8× in our testbed (§ 7.1). Moreover, to prevent long chains, the garbage collection (GC) is required to delete the obsolete versions that are no longer used by any transaction [16]. However, when using linked chains, GC is difficult to carry out on disaggregated memory, since the compute pool needs to frequently track the oldest transaction and reclaim the unused versions. Such tracking consumes many round trips for synchronizations and wastes the compute power.

To address the above challenges, we propose Motor, which holistically redesigns the version structure and transaction protocol to enable **m**ulti-versi**o**ning for distributed **t**ransacti**o**n processing on the disagg**r**egated memory. Instead of using linked chains, Motor leverages a new *consecutive version tuple* (CVT) structure to efficiently organize multiple versions of one data in the memory pool. CVT consecutively stores several versions together to fill in continuous address space. In this way, the compute pool is able to fetch all the versions of the same data by reading a CVT in a single round trip, instead of fetching the remote versions one by one, thus reducing the networking overheads to achieve low latency. When the CVT is filled up, Motor leverages a lightweight coordinator-active garbage collection (GC) scheme that reclaims the old versions in a preemptive manner without tracing transaction states. In the presence of GC, Motor also enables the applications to easily identify the consistency between the data value and its version in CVT to guarantee the correctness.

On top of the CVT structure, Motor designs a fast multi-version concurrency control (MVCC) based transaction protocol. This protocol fully leverages one-sided RDMA to bypass the weak compute units in the memory pool. Our protocol allows the reads not to be blocked by writes, and avoids writing logs, thus improving the concurrency and saving network



(a) Old-to-new chain (b) New-to-old chain (c) Read latency of different walk steps

Figure 1: The linked chain based version structures (a, b), and the latency of using RDMA `READ` for chain walking (c).

bandwidth. Moreover, our protocol supports various isolation levels (e.g., serializability and snapshot isolation) to flexibly meet the requirements of different OLTP applications.

In summary, this paper makes the following contributions:

- We propose Motor that enables multi-versioning for distributed transactions on the disaggregated memory.
- Motor designs a new consecutive version tuple (CVT) structure to efficiently organize multiple versions of data in the memory pool. CVT enables the compute pool to obtain the target version in one round trip, and provides lightweight garbage collection without the overhead of tracking (§ 4).
- Motor leverages a fast MVCC transaction protocol that fully exploits one-sided RDMA and CVT to meet the CPU-less memory pool with various isolation-level supports (§ 5).
- We implement[1] Motor and compare it with two state-of-the-art systems [64, 84]. The experimental results demonstrate that Motor significantly improves the transaction throughput by up to 98.1% and reduces the latency by up to 55.8%.

2 Background and Motivation

2.1 Memory Disaggregation

Traditional datacenters consist of many monolithic servers, each of which contains a set of compute and memory units. However, this monolithic architecture suffers from low resource utilization and coarse failure domain [65, 72]. Specifically, even if a user only needs more *compute* power, we have to add more entire servers in which the *memory* modules are wasted. Moreover, if a CPU is broken, the whole server becomes unusable, which expands the failure domain.

To improve resource utilization and failure isolation, memory disaggregation [20, 35, 46, 50, 51] becomes a promising solution, which decouples the compute and memory resources from a monolithic server to build separate resource pools. These pools are connected via fast network, e.g., RDMA [29] or CXL [7]. A compute pool contains many strong CPUs to intensively execute computing tasks. There are small amounts of DRAM in the compute pool to cache some metadata. Moreover, a memory pool consists of substantial memory modules to store the large-volume application data. The memory pool does not contain strong compute capability [46, 65, 69, 72, 75], but have some low-power compute units only for memory allocation and network interconnection [84, 86]. By efficient resource pooling, datacenters are able to provide appropriate amounts of compute and memory units to meet the requirements of different applications in an on-demand manner, thus

[1] Source code is available at https://github.com/minghust/motor.

improving the resource utilization and reducing costs [48]. Moreover, even if a CPU fails in the compute pool, the decoupled memory modules in the memory pool are not affected due to the separate architecture, thus narrowing the failure domain. Therefore, memory disaggregation is a promising solution for modern datacenters and cloud providers. Without loss of generality, this paper considers that the compute pool leverages one-sided RDMA verbs (including `READ`, `WRITE`, and atomics such as `CAS` an `FAA`) to access the application data in the memory pool to bypass remote CPUs like existing studies [53, 66, 75, 84].

2.2 Transactions on Disaggregated Memory

System Model. To provide atomicity and strong consistency for applications on the disaggregated memory, the compute pool is required to employ distributed transactions to access remote data in the memory pool [84]. Specifically, the CPU threads in a compute pool run many ***coordinators***, which execute a transaction protocol to read data, handle conflicts, and commit updates. The compute pool does not store application data, but contains a small amount of DRAM to buffer some metadata (e.g., remote data addresses). The memory pool stores all the application data without running compute tasks. Each data is replicated into multiple replicas for high availability. In practice, the fail-stop failure [36] could occur at any time to cause the data in the memory pool inaccessible[2] [27]. To tolerate such failures, we adopt the $(f+1)$-way primary-backup replication [42] to generate 1 primary replica and f backup replicas for each data in the memory pool. Each replica can be accessed by multiple coordinators. During transaction processing, coordinators in compute pool read/write remote replicas via network at the byte granularity, and the compute units in memory pool are not involved. Since the coordinators and replicas are fully separated by network, all transactions become distributed in our system model.
Limitations of Single-Versioning. Recently, FORD [84] supports distributed transactions on the disaggregated memory and stores the latest version of each data in the memory pool. This *single-versioning* design simplifies the memory store but incurs two limitations. (1) *Low concurrency.* During transaction commit, the data being updated cannot be read. FORD makes these data invisible until completing the write, thus blocking the read operations; (2) *High logging overheads.* FORD writes the undo logs to all replicas to guarantee atomicity. These undo logs consume the network bandwidth, and the coordinator needs to wait for all ACKs of the logging requests before committing the updates to remote replicas.

2.3 Enabling Multi-Versioning

To address the limitations of single-versioning, we adopt a *multi-versioning* methodology to store multiple versions of each data in the memory pool. By doing so, the writes do not

[2] In line with existing studies [27, 38, 39, 64, 77, 84], we currently do not consider the byzantine failures [37].

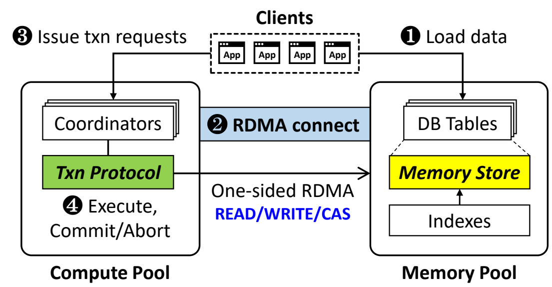


Figure 2: The system overview of Motor.

block reads, since the read request obtains an existing version of data, instead of waiting for the update operation, thus improving the concurrency. Moreover, the multi-versioning design does not need to additionally write logs to back up data in replicas, since the old versions naturally act as "undo logs" to guarantee atomicity. In this way, we eliminate the logging overheads to accelerate transaction commit.
Challenges. Existing studies have adopted multi-versioning in transaction processing [16, 43, 57, 64, 76]. However, as analyzed in § 1, these studies do not fit the new disaggregated memory architecture due to two reasons. (1) Their transaction protocols target on traditional monolithic servers, which requires powerful CPUs in each server to execute substantial compute tasks [57, 64, 76]. However, in the disaggregated memory architecture, the compute units in the memory pool are too weak to frequently handle compute tasks [75, 84, 86]. (2) The version structures of new-to-old and old-to-new linked chains incur substantial RDMA round trips for chain walking and high overheads for garbage collection.

To address the above challenges, we propose Motor to efficiently enable multi-versioning for fast distributed transaction processing on the disaggregated memory.

3 Motor Overview

Fig. 2 illustrates the system overview of Motor, which contains two parts working in harmony. First, the *Motor memory store* (§ 4) efficiently organizes multiple versions of data in the memory pool. Second, the *Motor transaction protocol* (§ 5) handles multi-versioning based distributed transactions in the compute pool.
Workflow. We outline the workflow of Motor. ❶ The client initially leverages the CPUs in the memory pool to allocate memory to load the application data into relational database (DB) tables. These tables are organized by our consecutive version tuple (CVT) structure, as described in § 4.1. The CVTs can be quickly accessed using indexes, e.g., hash table [86] or B+tree [75]. ❷ We establish RDMA connections between the compute and memory pools. Moreover, the memory pool sends some metadata (e.g., the address of the RDMA memory region and descriptions of indexes) to the compute pool. These metadata help coordinators locate the remote data at runtime. ❸ The clients issue transactions to the compute pool to be executed. ❹ The compute pool uses CPU threads to simultaneously run many coordinators, which leverage our

transaction protocol to process transactions. In general, the coordinators fetch and lock remote data, and then execute the transaction logic. After execution, the coordinators validate that the data versions are not changed. Finally, the coordinators commit the updates to remote memory pool and unlock data. Our protocol enables coordinators to fully use one-sided RDMA to bypass the weak CPUs in memory pool during transaction processing.

4 Motor Memory Store

4.1 Consecutive Version Tuple

Key Idea. Motor proposes a *consecutive version tuple* (CVT) structure to maintain different versions of data in the memory pool. Unlike the linked chains using pointers to link versions, CVT consecutively stores the versions together to fill in continuous address space. By using CVT, the coordinator is able to fetch multiple versions in a single RDMA `READ`, instead of performing the chain walking to read remote versions one by one until the target version. After fetching the CVT, the coordinator locally searches for the target version, which is fast due to not involving any network I/O.

Structure. Fig. 3 shows the structure of the memory store in the memory pool, which is organized by CVTs. All the CVTs form a CVT region. A CVT consists of a header and several version cells (Vcells). In a header, `TableID` indicates the DB table this record belongs to. A record is a row of user data, containing the key and value, in a DB table. Moreover, `Key` is the unique identifier of this record, and `Lock` is used for concurrency control in transaction processing (§ 5.1). The `AttrBarPtr` points to an attribute bar in the value region. An attribute bar stores the modified attributes of different versions of a record's value, as described in § 4.2. The `VpkgPtr` points to a value package (Vpkg) in value region. A Vpkg contains the actual data value, which is wrapped by a `VpkgSA` and a `VpkgEA` to indicate whether the value is completely written, as explained in § 4.4. Moreover, in a Vcell, the `VcellSA` and `VcellEA` work with the `VpkgSA` and `VpkgEA` to check the consistency between a version and its value (§ 4.4). The `Valid` indicates whether this version of value is available, and the `Version` represents a version number. In addition, the `Bitmap` indicates the modified attributes at the current version, and the `StartOffset` represents the offset of attributes stored in the attribute bar (more details are presented in § 4.2).

Number of Versions in CVT. Motor needs to configure the number of versions (VNum) to hold in CVT. Considering that the memory pool does not contain powerful CPU to dynamically adjust VNum in transaction processing, Motor sets VNum to be fixed, i.e., a record has a fixed maximum number of versions. In fact, it is challenging to determine an efficient VNum due to the tradeoff among read latency, memory footprint, and transaction abort rate. Specifically, if VNum is too small, the CVT size becomes small, which decreases the RDMA transmission payload to decrease the read latency,

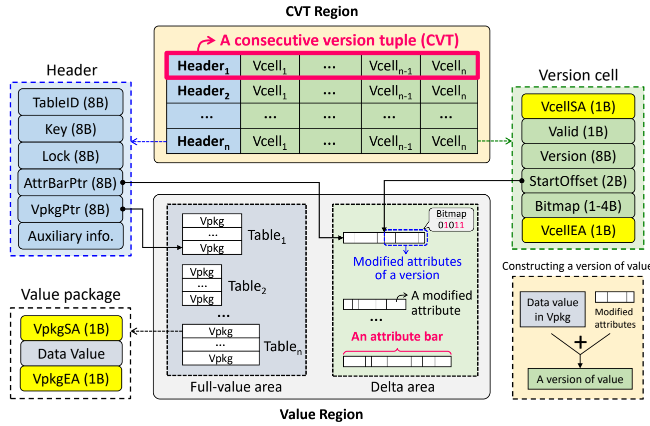


Figure 3: The structure of the Motor memory store, which is organized by CVTs in the disaggregated memory pool.

and also reduces the memory footprint in memory pool. However, due to limited available versions in CVT, the garbage collection (§ 4.3) can be frequently triggered, and this may increase transaction aborts to hamper the throughput when the contention is high. In contrast, if VNum is too large, it helps mitigate transaction aborts, but would waste memory in read-intensive workloads that do not require many versions of data. Moreover, since an entire CVT is read at a time, a large CVT increases the payload to lengthen the RDMA read latency. We explore such tradeoff in § 7.2 and § 7.6, and observe that a suitable VNum significantly depends on the characteristics of workloads (e.g., the access contention and the number of records to read in a transaction). In general, setting VNum to 2 is sufficient for low-contention workloads with short-running transactions (e.g., TATP [1]). For high-contention workloads with long-running transactions (e.g., TPCC [13]), a slightly larger VNum (e.g., 4) efficiently reduces transaction aborts without heavy memory footprint and high read latency.

Indexes Supports. Motor provides unified interfaces for coordinators to quickly access remote CVTs by leveraging indexes (e.g., hash table [86] and B+tree [75]). Motor stores CVTs within the index. For example, when using B+tree indexes, CVTs are stored in leaf nodes, and the internal pointer nodes are cached in compute pool to reduce remote tree traverses. When using hashing indexes, CVTs are stored in hash tables by hashing `Keys`. Therefore, writing CVTs simultaneously modifies the index. Without loss of generality, our paper considers to use the hash table as a case in point to present the details of indexing remote data like existing studies [26, 78, 84]. To address hash collisions, Motor reserves multiple slots in a hash bucket [86]. Each slot stores one CVT. Given a key (e.g., `K0`) of a record, the coordinator hashes `K0` to obtain the ID of hash bucket and calculate the remote address of this bucket. The coordinator then reads the bucket and locally traverses slots to search for the target CVT whose `Key` is equal to `K0`.

CVT Address Cache. In practice, it is expensive to fetch an entire hash bucket each time when reading a CVT. To address this issue, Motor enables each coordinator to leverage a small private *CVT address cache* in the compute pool to store the remote addresses of CVTs. When reading the same CVTs next time, the coordinator can quickly use the cached addresses to directly read the CVTs instead of hash buckets. However, if the `Key` of fetched CVT mismatches the queried key, the cached address becomes stale. The coordinator addresses this issue by re-reading the hash bucket to confirm the existence of the target CVT, and then updates its address cache. To store millions of addresses (each one is 8B), an address cache only consumes several MBs of DRAM space, which is acceptable for the compute pool [65, 84].

4.2 Separate Value Region

Some prior studies like FORD [84] and Silo [71] store the value together with its version, so that coordinators can fetch the value and version in one read. However, this design becomes inefficient in our context, because storing the value together with its version significantly increases the CVT size, leading to high read latency and network bandwidth waste (all values are transmitted but only one is needed). Such drawbacks become even worse when the value size gets larger.

To tackle the above challenge, Motor separates the CVTs from data values in memory pool. The coordinator first reads a CVT to determine the target version, and then reads the corresponding value. In this way, the CVT size is not affected by the value size to achieve stable low read latency, and only one data value is transmitted to mitigate bandwidth wastes.

Reducing Memory Overhead. In the value region, storing a full-sized data value for each version simplifies the data store but wastes memory space. To alleviate the memory overhead, we have two observations. (1) The records in a relational DB table follow the same schema, which defines the number of attributes of the value and the size of each attribute. (2) When updating a record, a transaction can modify only one or several attributes. For example, in TPCC, the value of a record in `DISTRICT` table contains 9 attributes (100B in total), but in `NEW_ORDER` transaction only one attribute is modified, i.e., `D_NEXT_O_ID` (4B). Based on these observations, Motor stores the variable-sized modified attributes, instead of full-sized values, to maintain different versions of values for any record, thus reducing the memory overhead. Fig. 3 shows that the value region contains a full-value area plus a delta area. The full-value area stores the newest version of full-sized values, and the delta area stores old attributes being modified by transactions (like "undo logs"). Therefore, an updated record has only one full value and different versions of variable-sized attributes that are actually modified. To construct an old-version value, we only need to apply the attributes at the old target version into the newest full value.

Attribute Bar. In the delta area, Motor leverages a new structure, called *attribute bar*, to consecutively and compactly store the modified attributes of a record across transactions, as illustrated in Fig. 3. Motor uses the following metadata in CVT to efficiently manage attributes bars.

1) `AttrBarPtr` in Header. When a record is updated for the first time, the coordinator allocates an attribute bar in the delta area, and keeps the remote address of the attribute bar (i.e., `AttrBarPtr`) in the CVT's header.

2) `Bitmap` in Vcell. The coordinator uses a bitmap in Vcell to represent the modified attributes at the current version. For example, if a value has 8 attributes and the 1st, 2nd, and 4th attributes are modified by a transaction, the coordinator writes a bitmap of "00001011" (the rightmost bit represents the first attribute, i.e., the little-endian style) into the Vcell. The length of bitmap depends on the number of attributes.

3) `StartOffset` in Vcell. This is used to represent the offset of a group of modified attributes at the current version inside the attribute bar. The initial `StartOffset` is 0. The coordinator calculates a new `StartOffset` by using the last-written Vcell's `StartOffset` and `Bitmap`. Specifically, according to the positions of "1" in the last-written bitmap, the coordinator accumulates the total size of attributes in the last write, and adds this total size with the last-written `StartOffset` to obtain a new `StartOffset`.

Attribute Bar Size. A coordinator needs to allocate a proper-sized attribute bar to hold modified attributes to alleviate memory wastes. By sampling transaction execution, we observe that for records in a DB table, the total sizes of attributes being updated per transaction (called TotAttrSizes) are different but occur at specific frequencies. For example, in TPCC's `CUSTOMER` table, the TotAttrSize can be 512B, 12B, and 4B, respectively occurring at frequencies of 10%, 88%, and 2% across transactions. This is because in OLTP scenarios, the transaction logic specifies the attributes to update, and different transactions follow the standard execution ratio in the transaction mix [1, 4, 13]. According to the frequencies of different TotAttrSizes, Motor reserves corresponding proportions of space in the attribute bar to hold these attributes of VNum versions (i.e., if some attributes are more frequently updated, Motor reserves more space for these attributes). Hence, Motor approximately estimates the attribute bar size (ABS) = $\sum_{i=1}^{n}(\max(VNum \times Frequency_i, 1) \times TotAttrSize_i)$, where n is the number of TotAttrSizes. For example, when VNum = 4, the ABS of records in `CUSTOMER` table is: 1×512B + 3×12B + 1×4B = 552B, which is sufficient to hold modified attributes of different versions without wasting memory. Note that even if all attributes of a value are modified at some versions (i.e., TotAttrSize = full-value size), the attribute bar can still store all these attributes, since in this case the calculated ABS is guaranteed to be larger than the full-value size.

Mitigating Contentions on Allocating Attribute Bars. When coordinators simultaneously allocate attribute bars, they will compete for the free space in delta area, leading to high contentions. To avoid this, Motor pre-assigns a small MB-scale delta space with proper size (based on ABS) in the delta area to each coordinator. In this way, the coordinator allocates attribute bars in its own delta space without competing with others. The `AttrBarPtr` is globally visible to all coordinators after completing the update operation, so that a coordinator is able to append attributes to the attribute bars created by other coordinators. In rare cases the delta space is exhausted, the coordinator informs remote CPU to allocate larger space.

One RTT for Reading/Writing Values. Though the full value and attributes are separated, Motor consumes only one round-trip time (RTT) to read/write a value at target version. (1) *Read*. A coordinator selects the target version (e.g., $V0$) in a CVT. The selection scheme is presented in § 5.1. If $V0$ is the newest version, the coordinator reads the full value using RDMA `READ` in one RTT. Otherwise, the coordinator calculates remote addresses of the required old attributes by using `AttrBarPtr` in CVT header and `StartOffset` as well as `Bitmap` in the Vcells whose `Version` is larger than $V0$. The coordinator then uses batched RDMA `READ`s to read the full value and old attributes together in one RTT and locally constructs an old version of value. (2) *Write*. The coordinator uses batched RDMA `WRITE`s to update the full value and appends old attributes to the attribute bar together in one RTT.

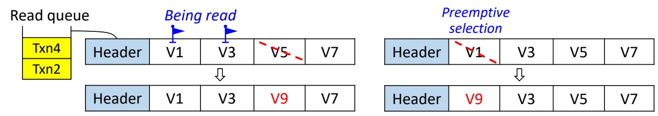
4.3 Coordinator-Active Garbage Collection

If there is no empty Vcell when updating data, we need a garbage collection (GC) mechanism to reclaim the obsolete versions. Legacy GC schemes track the oldest running transactions and delete the versions that are no longer used [16,64]. However, since the compute unit in the memory pool is not aware of transaction states, it is difficult to apply tracking in the memory pool. On the other hand, if the compute pool performs tracking, the coordinators need to confirm which versions are unused among all the in-flight transactions. This increases the network round trips for synchronizations and wastes the compute power.

In order to avoid the overhead of tracking, Motor proposes a *coordinator-active GC* scheme. The idea is that, if there is no empty Vcell, Motor allows the coordinator to actively select a victim version to be overwritten by the new version to complete GC. This scheme is lightweight due to eliminating the need of tracking the oldest running transaction.

To select the victim version, Fig. 4a shows a baseline scheme that skips the versions being read in a CVT, and selects the oldest version in the remaining versions. A read queue is reserved in each CVT to store the timestamps of transactions that are reading the CVT. Other coordinators check the read queue and skip the in-use versions. However, for read operations, since the coordinator does not know the current position of the queue's tail, it has to use RDMA `FetchAndAdd` to atomically move the tail, and then use RDMA `WRITE` to insert a timestamp to the read queue. Such extra RTTs in each read significantly increase the latency.

We observe that the oldest version in CVT has the smallest probability to be used, given that RDMA significantly accelerates transactions [26,78]. Hence, Motor enables coordinators to preemptively select the oldest version in CVT as the victim, as shown in Fig. 4b. This GC scheme avoids the RTT overhead in the baseline method. The tradeoff is that some long-running transactions would be aborted if their previously read data are quickly reclaimed. Nevertheless, the experimental results in § 7.2 show that reserving a proper number of versions in CVT efficiently mitigates such aborts. Overwriting old versions will make the versions in CVT unsorted, but the correctness is not affected, since the coordinator locally traverses all the versions in CVT to locate the target one.



(a) Skipping the versions being read (b) Overwriting the oldest version

Figure 4: Different garbage collection schemes for CVT.

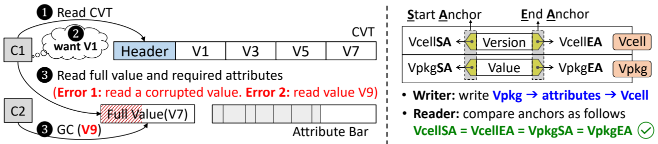
Note that if the attribute bar does not have enough space, the coordinator reclaims old attributes from the start of the attribute bar to write newly modified attributes. In this procedure, the coordinator checks which Vcells correspond to the reclaimed attributes, and sets the `Valid` in these Vcells to 0 to delete these versions. Since Motor appropriately configures the size of attribute bar to store attributes of multiple versions, reclaiming the old attributes does not invalidate many Vcells.

4.4 Anchor-Assisted Read

To obtain a data value, the coordinator reads a CVT to select the target version, and then reads the full value and necessary attributes. As shown in Fig. 5a, coordinator `C1` reads a CVT and needs the value at version $V1$ (Value$_{V1}$). `C1` reads the full value (Value$_{V7}$) and old attributes to reconstruct Value$_{V1}$. At this point, another coordinator `C2` is performing GC to reclaim version $V1$ and write Value$_{V9}$. As a result, there are two incorrect results for `C1`. (1) `C1` reads a corrupted full value due to being partially updated by `C2`. (2) `C1` reads Value$_{V9}$ but mistakenly regards it as Value$_{V7}$, thus reconstructing an incorrect Value$_{V1}$. The root cause of this issue is that the version and data value are separately stored, which prevents coordinators from "atomically" reading a value and its version.

To address the above challenge, Motor proposes an *anchor-assisted read* scheme to help coordinators identify the consistency between the version and value. As shown in Fig. 5b, this scheme uses two anchors at the start and end of a Vcell, called VcellSA (i.e., Vcell's **S**tart **A**nchor) and VcellEA (i.e., Vcell's **E**nd **A**nchor). Similarly, in a Vpkg, two anchors (VpkgSA and VpkgEA) are used to wrap the full value. An anchor is 1 byte. A pair of SA and EA and the content they wrap are implemented in a C++ `struct`, allowing a coordinator to access them together using a single RDMA `READ` or `WRITE`.

To make anchors efficiently work, coordinators follow two rules. (1) *Write*. A coordinator increases the anchor value by 1 for all the four anchors (i.e., VpkgSA, VpkgEA, VcellSA, and VcellEA) to make them **equal**. The coordinator writes the Vpkg first, then the modified attributes, and finally the Vcell. (2) *Read*. A coordinator reads a CVT and then fetches the Vpkg and necessary attributes. Since the full value region stores the newest value, the VpkgSA and VpkgEA are also



(a) Reading an incorrect value caused by concurrent GC (b) Using anchors to detect incorrectness

Figure 5: The anchor-assisted read scheme.

the newest. Hence, the coordinator checks whether the newest VcellSA and VcellEA in CVT are equal to VpkgSA and VpkgEA. If the four anchors are equal, the full value and attributes are not modified since the last read. The coordinator then safely reconstructs the target-version value by copying the fetched old attributes into the newest full value. However, if any of the two anchors are not equal, the coordinator aborts the transaction due to detecting partial updates or a conflicting in-flight GC procedure. In essence, the four anchors assist the coordinator to read a version and the corresponding value in an "atomic" manner. Unlike Silo [71] that reads the version twice to confirm consistency, our scheme only needs to read once and compares the four anchors to identify consistency.

Guaranteeing Write Order. The correctness of the anchor-assisted read scheme is based on that all the written data are installed into the memory pool in the correct order, which has two requirements. **[R1]** Vpkg → modified attributes → Vcell. **[R2]** Inside a Vpkg (or Vcell): start anchor → content → end anchor. In practice, the two requirements are satisfied in network and at remote RDMA NIC (RNIC), because (1) the *reliable connection* mode for one-sided RDMA guarantees that the transmitted messages are not lost or reordered [6], and (2) when the request reaches the remote RNIC, the RNIC ensures that the RDMA `WRITE`s are totally ordered with regard to each other [61], i.e., these write requests are sent to the on-chip integrated memory controller (iMC) in order. However, the two requirements can be then violated due to DDIO (i.e., Data Direct I/O [8]). If DDIO is enabled, iMC sends the written data to the L3 CPU cache. Due to unpredictable cache behavior, the data in L3 cache could be evicted to memory out of order to break **R1** and **R2**. In fact, DDIO aims to improve the cache locality, which benefits the CPU execution in traditional monolithic servers, but becomes useless in the disaggregated memory, since the weak CPU in memory pool is not involved during transaction processing. Hence, Motor disables DDIO in the memory pool, so that iMC directly sends writes from its internal first-come-first-serve write pending queue to the main memory. In this way, the writes are installed into remote memory in the correct order to satisfy **R1** and **R2**.

5 Motor Transaction Protocol

We present the Motor transaction protocol. Our protocol works in a widely-recognized transaction processing framework, which includes reading data, handling conflicts, and writing data back. The main difference from existing studies [27, 39, 64, 77, 78, 84] is that our protocol fully exploits the CVT structure and pure one-sided RDMA to support MVCC based distributed transactions on the disaggregated memory.

Timestamp Generation. Motor leverages sequential numbers as transaction timestamps (i.e., 1, 2, 3 ...), which are also adopted as data versions. In fact, the timestamp generation is orthogonal to our designs. Existing studies propose scalable timestamp generation schemes [24, 38, 64, 76], which can be applied to the compute pool as the timestamp service to assign strictly and monotonically increasing timestamps. Our paper does not focus on optimizing the timestamp generation, and we assume that a scalable timestamp service is efficiently leveraged in the compute pool to serve for all coordinators.

Overview. In the memory pool, each table is replicated to 1 primary and f backups, and the weak CPUs are not involved during transaction processing. In the compute pool, the coordinators leverage our protocol to execute transactions and access remote data through one-sided RDMA.

5.1 Processing Phases

Fig. 6 shows the procedure of handling a read-write transaction (e.g., `T0`) with serializability guarantee. All requests in the same RTT are issued in parallel. The read-write set is {`A`, `B`} and the read-only set is {`C`}. In Motor, the write set is included in the read set, since (1) for *Updates* and *Deletions*, the coordinator reads remote CVTs before writing data back, and (2) for *Insertions*, the coordinator reads remote buckets to obtain empty CVTs before inserting data. The detailed processing phases are presented below.

Phase 1. Execution. The coordinator obtains a start timestamp (T_{start}) from the timestamp service. For each read-only (RO) or read-write (RW) data, the coordinator looks up its local CVT address cache. (1) If the address has been cached (e.g., `A` and `C`), for the RO data (e.g., `C`), the coordinator uses RDMA `READ` to fetch their CVTs from the primaries; for the RW data (e.g., `A`), the coordinator uses doorbell-batched RDMA `CAS+READ` to respectively lock and read the CVTs from the primaries. The locking request prevents other conflicting transactions from modifying the same CVT at the same time. If the locking request fails, the coordinator aborts the transaction, instead of waiting, to avoid deadlocks. (2) If the address is not cached (e.g., `B`), the coordinator uses RDMA `READ` to fetch a hash bucket and then locally search for a `Key`-matched CVT. After obtaining the CVT, the coordinator selects a target version $V0$, which is the largest version among all the versions that are smaller than T_{start}.

Early Abort. If the coordinator observes a version (e.g., $V1$) larger than T_{start} in the CVT, it means that another transaction `T1`, has committed after `T0`'s T_{start}. In this case, the coordinator can *early abort* `T0` to guarantee serializability. The reason is that, even if using T_{start} to select $V0$ for execution, `T0` will be aborted in the next Validation phase, in which `T0` will obtain a larger commit timestamp than `T1`. That is, `T0` with a larger commit timestamp should have used `T1`'s update, i.e., $V1$, for execution, but `T0` used $V0$. Hence, the coordinator early aborts `T0`. Note that the early abort is unnecessary in the snapshot isolation, since it is sufficient for `T0` to read a snapshot at T_{start}, even if the snapshot becomes slightly stale [76].

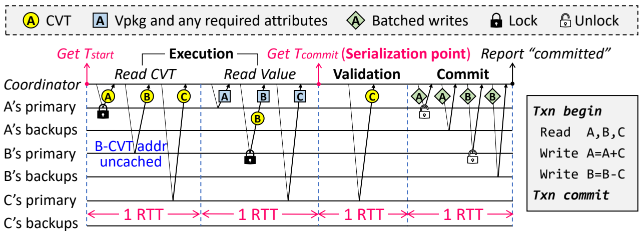


Figure 6: The distributed transaction protocol of Motor.

After version selection, the coordinator uses batched RDMA `READ`s to read the Vpkgs and any required old attributes to construct the target-version value (§ 4.2). Note that for RW data that have not been locked (e.g., `B`), the coordinator additionally batches RDMA `CAS` with `READ`s to lock and re-read their CVTs when reading Vpkgs. After fetching all the data, the coordinator performs three checks for correctness: (1) if any locking fails, `T0` is aborted; (2) if a newer version larger than $V0$ occurs in the re-read CVT, `T0` is aborted, since another transaction has updated this data; (3) if the four anchors are not equal, `T0` is aborted, because the version and value are inconsistent. If passing all checks, the coordinator safely uses the data value inside the Vpkg to execute the transaction logic. Though Motor uses two RTTs to read the CVT and data value, the network payload is significantly reduced due to not transmitting unnecessary data values.

Phase 2. Validation. After all the remote CVTs of the RW data are successfully locked, the coordinator obtains a commit timestamp (T_{commit}) from the timestamp service. Note that if the read-write transaction does not contain any RO data, the following operations can be skipped to reduce latency, since all the RW data have been already locked. However, if the transaction contains RO data, the coordinator needs to validate that the versions of RO data are not changed from T_{start} to T_{commit} to provide serializability. To this end, the coordinator re-reads the CVT of each RO data from remote primaries and uses T_{commit} to select a version V', which is the largest version among all the versions that are smaller than T_{commit}. The coordinator checks whether any of the two cases occur: (1) the CVT is locked by another coordinator, or (2) $V' \neq V0$. In the first case, it is possible that another transaction with a lower T_{commit} is committing a new version. The second case means that another transaction with a lower T_{commit} has committed a new version. If either case occurs, the validation fails, because `T0` with a higher T_{commit} should read the new version but fails to do so in the Execution phase. As a result, `T0` is aborted to ensure serializability. In short, the validation succeeds only if the CVT is not locked and $V' = V0$.

Phase 3. Commit. When the validation succeeds, a coordinator commits the updates to all remote replicas together in a single RTT. The coordinator locally prepares the data to be written, which can be interpreted in three scenarios. **(1) Update.** If the record is updated for the first time, the coordinator allocates an attribute bar in its own pre-assigned delta space. The coordinator then finds an empty Vcell (i.e., `Valid` is 0) in the fetched CVT, sets the `Valid` to 1, fills the `Version` using T_{commit}, sets the `Bitmap` of the updated attributes, calculates the `StartOffset` inside the attribute bar, and configures both of VcellSA and VcellEA to be equal to a new number. If there is no empty Vcell or the `StartOffset` exceeds the length of attribute bar, the coordinator actively performs GC to reclaim old versions. Moreover, the coordinator collects the modified attributes that will be written to the attribute bar. The coordinator then prepares a new Vpkg by filling the new data value, and setting both of VpkgSA and VpkgEA to be equal to VcellSA. **(2) Insert.** Apart from preparing the Vpkg and Vcell like the Update operation, the coordinator prepares a new header and fills the `TableID`, `Key`, and `VpkgPtr`. The `TableID` and `Key` come from applications. The coordinator allocates the `VpkgPtr` in its delta space, i.e., Motor allows the newly inserted data to share the delta area with attribute bars to improve the space efficiency. **(3) Delete.** The coordinator sets the `Valid` of $V0$ to 0, so that subsequent transactions with larger timestamps cannot use the deleted version. The delete operation needs to set the full value in remote memory pool to an old-version value. To this end, the coordinator copies the old attributes fetched in Execution phase into the full value.

After these local preparations, the coordinator leverages doorbell-batched RDMA `WRITE`s to write the prepared data to all replicas and unlocks primaries in one RTT. When receiving *all* ACKs from *all* replicas, the coordinator reports "committed" to the application.

Processing Read-Only Transactions. A coordinator obtains a read timestamp (T_{start}) and reads the required CVTs from the primaries. The coordinator uses T_{start} to determine the target version, and then fetches the Vpkgs and any required old attributes from primaries to construct the value at the target version. If the four anchors are equal, the transaction commits, and otherwise aborts. Note that in single-versioning designs, the read-only transactions require validation [27, 39, 77, 84]. However, with multi-versioning, the read-only transactions do not require validation [57] due to obtaining a stable version snapshot at T_{start} (more details are discussed in § 5.2).

5.2 Flexible Support of Isolation Levels

By using our protocol, Motor supports two widely-used isolation levels, i.e., serializability (SR) [11] and snapshot isolation (SI) [12], to flexibly meet the requirements of different OLTP applications. With SR, the concurrent transactions appear to be executed one by one. Moreover, with SI, the transaction reads data from a snapshot at a time, which does not reflect changes made by other in-flight transactions.

Supporting SR. (1) For ***read-write*** transactions, they are serializable at the point of T_{commit} if guaranteeing that all the target versions selected at T_{start} are equal to those at T_{commit}. This property allows the transactions to be considered as executing at their T_{commit} one after another. Motor ensures this property by using locks and validations. i) If a transaction obtains all the locks of CVTs at T_{start}, the versions of read-write data cannot be changed by other transactions until T_{commit}. Hence, the versions of read-write data at T_{start} are equal to those at T_{commit}. ii) During validation, if a transaction detects that the remote CVT is locked or a new version appears at T_{commit}, the validation fails and the transaction aborts, since the previously fetched versions of read-only data become stale. If the validation succeeds, the versions of read-only data at T_{start} are equal to those at T_{commit}. (2) For ***read-only*** transactions, they do not have a commit timestamp due to not making data changes. In the multi-versioning design, since read-only transactions only observe a snapshot, the start time of read-only transactions can be considered to be "movable" in order to find a serializable execution order [57], i.e., the read-only transactions can be placed among other read-write transactions to make all the transactions appear to execute one by one. In summary, the write-write and read-write conflicts between transactions are respectively addressed by using locks and validations, which ensure that the precedence graphs [5] of all the transaction schedules do not contain cycles, thus guaranteeing serializability [68].

Supporting SI. To support SI, Motor disables the version validation for the read-only data in read-write transactions, i.e., these transactions are allowed to use a stale snapshot by using T_{start}. Note that the locking is still required to resolve the write-write conflicts. SI is weaker than SR, but achieves higher performance (as demonstrated in § 7.7) and has been adopted by multiple popular systems, e.g., MySQL [56], PostgreSQL [60], Oracle [59], and SQL Server [63].

ACID Guarantee. Motor guarantees ACID for transactions. (1) **A**tomicity. Motor maintains multiple versions of data, and the old versions act as "undo logs" to preserve the atomicity. (2) **C**onsistency. The data versions in memory pool are in a consistent state before a transaction starts and after it commits. (3) **I**solation. Motor supports serializability and snapshot isolation. (4) **D**urability. Motor stores $f+1$ replicas of each data against data loss, and can employ UPS-backed DRAM [27] or persistent memory [84] in the memory pool to durably store the committed updates even if a power failure occurs.

5.3 Fault Tolerance

Replica Failures in Memory Pool. By enabling data replication, Motor is able to tolerate replica failures in the memory pool. The replica failures can be quickly detected using RDMA [27]. If any replica fails before commit, the coordinator discards all the fetched data, unlocks remote locks, and aborts the transactions. If a primary fails during commit, Motor promotes a backup as the new primary to retain the committed updates, because the backups have the same updates as primary. The new primary is not visible to coordinators until the updates are installed into alive replicas. When the new primary becomes visible and subsequent coordinators can grab locks on the new primary, the updates of previous transactions have been already committed, thus guaranteeing serializability. Moreover, if a backup fails during commit, the coordinator selects another memory node to add a new backup. Adding a backup requires data migration, in which Motor enables memory nodes to use RDMA `WRITE` to quickly transmit application data. Subsequent transactions involving failed replicas hang up until the replicas are recovered. The $(f+1)$-way replication tolerates at most f replica failures.

Coordinator Failures in Compute Pool. In line with existing studies [27, 78], Motor supports to use leases [31] to detect coordinator failures. Motor enables the coordinators to write small-sized operation logs in local memory to record the operations (e.g., the keys that will be locked or committed) during execution. The operation logs are stored in UPS-backed memory and are not lost [27]. If a coordinator fails, Motor employs a new one to use the operation logs to resume the in-flight commit and unlock keys for recovery. For example, the new coordinator uses RDMA `CAS` to unlock the recorded keys, i.e., if the `CAS` succeeds, the previous lock is released to avoid starvation, and otherwise the key is actually not locked.

Network Failures. A network failure causes the network partition. In practice, it is hard to distinguish network failure from server failure. Like uKharon [34], we assume that the network partitions are discovered and resolved by datacenter administrators. If a network partition occurs, either availability or consistency cannot be fully guaranteed according to the CAP theorem [18, 30]. In the context of OLTP applications, offering consistency is more important to satisfy the ACID requirements. Hence, Motor weakens the availability by only allowing the major partition [17] to serve requests.

6 Implementations

We present some important implementation details including the transaction interfaces and execution framework.

Easy-to-Use Transaction Interfaces. Motor provides the following interfaces for applications to easily run MVCC based distributed transactions on the disaggregated memory.

- `TxnBegin()`: Start a transaction and record its ID.
- `GetTS()`: Get a timestamp from the timestamp service.
- `AddObject()`: Add a read-only (or read-write) object to the read-only (or read-write) set.
- `FetchAll()`: Obtain remote CVTs and target-version data values. The remote CVTs are simultaneously locked.
- `Validate()`: Validate the versions of read-only data.
- `TxnCommit()`: Commit the transaction by writing the updates back to remote replicas and unlocking the primaries.

Execution Framework. In the compute pool, Motor uses the CPU cores to spawn massive threads to execute transactions in parallel. However, if using a thread as a coordinator, the CPU core will become idle when waiting for RDMA ACKs, which decreases the throughput. To saturate the compute power of a CPU core, Motor generates multiple coroutines in a CPU thread to execute in a pipeline manner [39, 77, 84]. In a thread, one coroutine polls the RDMA ACKs, and each of the other coroutines acts as a transaction coordinator. Therefore, Motor enables substantial coordinators to concurrently execute transactions in the compute pool.

7 Performance Evaluation

7.1 Experimental Setup

Testbed. We configure four servers connected through a Mellanox SB7890 100Gbps InfiniBand (IB) Switch. Each server contains a 100Gbps Mellanox ConnectX-5 IB RNIC. One server containing Intel Xeon Gold 6330 CPUs is configured as the compute pool to run coordinators. Other three servers form the memory pool, and each server contains 192GB DRAM.

Benchmarks. We leverage a key-value store (`KVS`) as a microbenchmark. `KVS` stores 10M key-value pairs in one database (DB) table. The key is 8B and the value is 40B [39, 84]. In `KVS`, each transaction performs a read or an update operation to a 48B KV pair with skewed accesses following the Zipfian distribution [23]. We enable the skewness and the ratio of read-write transactions in the transaction mix of `KVS` to be configurable to facilitate comprehensive evaluation. Furthermore, we leverage three widely-used OLTP benchmarks, i.e., `TATP` [1], `SmallBank` [4], and `TPCC` [13], to evaluate the end-to-end transaction throughput and latency. Specifically, `TATP` shows a telecom application, which includes 4 DB tables and 80% of the transactions are read-only. `TATP` contains 2M subscribers and the record size is up to 48B. `SmallBank` models a banking application, which contains 2 DB tables and 85% of transactions are read-write. `SmallBank` has 10M accounts and the record size is 16B. `TPCC` models a complex ordering system, which contains 9 DB tables and 92% of transactions are read-write. `TPCC` contains 24 warehouses and the record size is up to 672B. Moreover, for all benchmarks, each DB table is replicated to three memory nodes to maintain a 3-way replication, i.e., 1 primary and 2 backups.

Comparisons. We compare our Motor with two state-of-the-art systems, i.e., FaRMv2 [64] and FORD [84]. FaRMv2 supports multi-versioning for transactions on monolithic servers, and uses the new-to-old chains to link versions [64]. To make FaRMv2 compatible with disaggregated memory (DM), we use one-sided RDMA to implement its transaction protocol, which is referred to as FaRMv2-DM in the rest of this paper. Moreover, FORD supports single-versioning for transactions on the disaggregated memory, and we run its open-source code. Though FORD leverages persistent memory, its one-sided RDMA designs on transaction protocol are also compatible with DRAM. Note that Motor targets on the disaggregated architecture, which is not comparable with the systems running on the monolithic architecture [39, 57, 76].

Performance Metrics. We report the transaction throughput by counting the number of committed transactions per second. Moreover, we report the 50th and 99th percentile latencies of committed transactions as the transaction latency.

7.2 Number of Versions in CVT

We explore how the number of versions (VNum) in CVT affects the performance of Motor. For each benchmark, we vary VNum from 2 to 15. The ratio of read-write transactions in `KVS` is 80%. Fig. 8 and 9 show that as VNum increases,

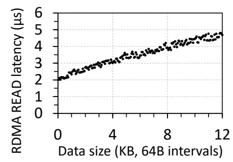


Figure 7: The latency of reading different sizes of data.

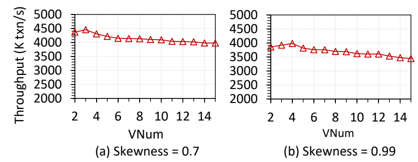


Figure 8: The transaction throughput on `KVS` benchmark when varying VNum with skewness 0.7 and 0.99.

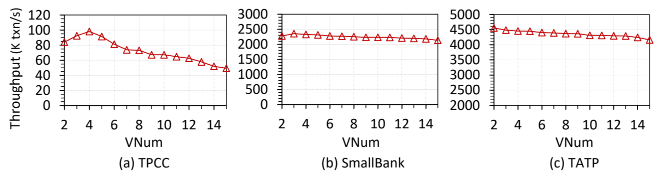


Figure 9: The transaction throughput on `TPCC`, `SmallBank`, and `TATP` benchmarks when varying VNum.

the transaction throughput generally first increases and then decreases. The reason is that, when VNum gets larger, the abort rate of read-only transactions is reduced to increase the throughput. For example, in `TPCC`, the abort rate of a long-running read-only transaction `STOCK_LEVEL` decreases from 32.1% (VNum = 2) to 3.8% (VNum = 4). However, after reaching the peak transaction throughput, increasing VNum no longer significantly reduces aborts, but the CVT size continues to increase, which enlarges the payload size to increase RDMA read latency, as shown in Fig. 7. The increased read overhead overwhelms the benefit of reducing aborts, thus decreasing the performance. Besides, large VNums also consume more memory space, as presented in § 7.6. Fig. 8 shows that at skewness 0.7, `KVS` reaches the peak throughput earlier than 0.99, since a larger skewness incurs higher access contention and requires more versions to reduce aborts.

We observe that, as VNum increases after the peak throughput, the throughput degradation of `TPCC` (up to 49.6%) is heavier than other workloads. This is because one transaction in `TPCC` can access hundreds of records, which is much larger than other benchmarks, e.g., one transaction in `SmallBank` (or `TATP`) only accesses 1–3 (or 1–4) records. Therefore, the overall read overhead (considered as CVT size $\times$ number of records) of `TPCC` transactions is more sensitive to VNum, leading to sharper performance decrease. `SmallBank` is write-intensive, but its transactions are short, and maintaining 3 versions reaches the peak performance. `TATP` only requires 2 versions for a record to achieve the peak throughput, since 80% of transactions in `TATP` are read-only and short-running with low contentions. As VNum grows, the high read overhead leads to continuous throughput degradation in `TATP`.

In summary, determining a suitable VNum significantly depends on the characteristics of workloads, including the access contention and the number of accessed records in a transaction. When the contention is low (e.g., `TATP`), setting a small VNum is enough. If the contention is high, more versions are needed to allow higher concurrency, especially for the long-running transactions. We also need to consider

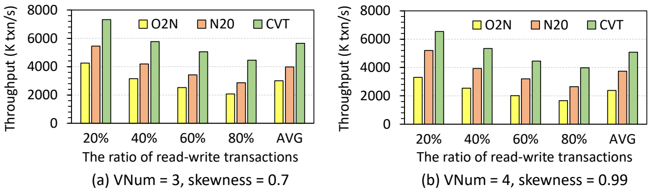


Figure 10: The transaction throughput of different version structures on `KVS` benchmark.

the number of records accessed per transaction to avoid large CVTs incurring high overall read overhead. According to these results, we respectively set the suitable VNum in `TPCC`, `TATP`, `SmallBank`, and `KVS` to 4, 2, 3, and 4.

7.3 Performance of Version Structures

We compare the performance of our CVT and traditional linked-chain version structures, i.e., old-to-new (O2N) and new-to-old (N2O), upon the `KVS` benchmark. We configure the access skewness as 0.7 and 0.99, and vary the ratio of read-write transactions (RW-ratio) from 20% to 80% in the transaction mix of `KVS`. Based on the results in Fig. 8, we change the maximum number of versions to hold for all structures to 3 for skewness 0.7, and 4 for skewness 0.99.

Fig. 10 shows that CVT respectively improves the throughput by 1.7–2.4× and 1.3–1.6× compared with O2N and N2O. The reason is that, CVT enables the transaction to fetch the target version in a single round trip, while O2N and N2O require multiple round trips for chain walking. When increasing the RW-ratio, the throughputs of three structures decrease, since the write conflicts increase and read-write transactions require more round trips to commit. When the skewness is high (e.g., 0.99) and RW-ratio is low (e.g., 20%), the throughput gap between N2O and CVT becomes small, because the access is more concentrated and many read-only transactions quickly obtain new values from the chain head of N2O. However, such performance gap between O2N and CVT becomes larger at high skewness since the new versions in O2N are placed in the chain tail, which increases the read overhead. Moreover, CVT respectively reduces the 50th (and 99th) percentile latencies by 59.8%/30.8% (and 67.9%/47.7%) on average compared with O2N/N2O at skewness 0.99 due to the same reasons above. We have also examined that when further increasing the maximum number of versions to hold, CVT can deliver more performance benefits over O2N and N2O.

7.4 End-to-End Performance

We leverage `TATP`, `TPCC`, and `SmallBank` to evaluate the end-to-end performance of Motor, FORD, and FaRMv2-DM. All systems guarantee serializability. We configure the maximum number of versions in FaRMv2-DM's version chain to be the same as our CVT for fair comparisons. Fig. 11 illustrates the transaction throughput and latency. To plot a throughput-latency curve, we increase the request load by running 10–40 threads and 2–8 coroutines per thread, i.e., 10–280 concurrent coordinators. Each thread executes 1M transactions following the standard transaction mix of each benchmark [1, 4, 13].

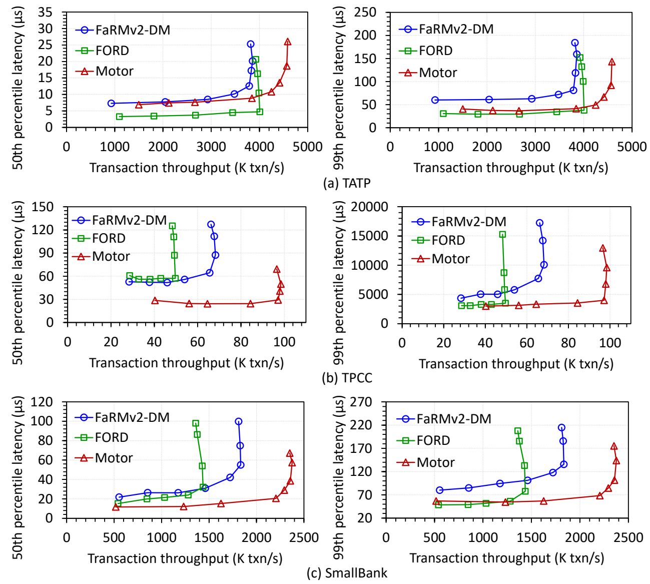


Figure 11: The transaction throughput and latency of all the systems on `TATP`, `TPCC`, and `SmallBank` benchmarks.

Compared with FORD, Motor respectively improves the transaction throughput by 14.4% on `TATP`, 98.1% on `TPCC`, and 65.4% on `SmallBank`. FORD adopts the single-versioning design, which limits the throughput, since reads are blocked by writes during commit, and the undo logs consume network bandwidth. Unlike FORD, Motor allows to read existing versions in CVTs, and does not need to write undo logs to remote replicas by maintaining old versions of values. Hence, Motor improves the throughput over FORD. The improvements are higher in `TPCC` and `SmallBank`, because (1) they are write-intensive workloads in which Motor avoids many undo logs, and (2) Motor reserves multiple versions to reduces aborts for read-only transactions, especially long-running ones, e.g., `STOCK_LEVEL` in `TPCC`. FORD delivers the lowest 50th percentile latency in `TATP`, since the two transactions, i.e., `GET_SUBSCRIBER_DATA` and `GET_ACCESS_DATA`, occupy 70% of the transaction mix, and both of them only read one object. In this case, FORD only uses one RTT to read data, while Motor requires two RTTs to separately read the CVT and data value. However, the 99th percentile latency of Motor on `TATP` is close to FORD when the transaction becomes complex. Furthermore, Motor reduces the 50th percentile latency by 55.8%/26.2% on `TPCC`/`SmallBank` compared with FORD.

Compared with FaRMv2-DM, Motor respectively improves the transaction throughput by 18.9%/44.3%/29.5%, and reduces the 50th (99th) percentile latencies by 8.6% (39.1%) / 52.1% (35.6%) / 43.6% (34.5%), on `TATP`/`TPCC`/`SmallBank`. Motor achieves these improvements due to three reasons. (1) FaRMv2 uses the linked chain to store different versions, which increases network round trips to perform chain walking to obtain the target version. Unlike FaRMv2, Motor uses CVT to fetch the versions together in one round trip. Motor shows the highest improvement over FaRMv2-DM in `TPCC`, since `TPCC` requires more versions and the transactions read many

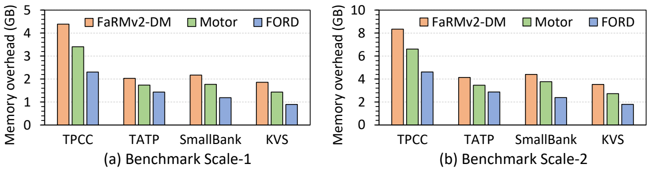


Figure 12: The space consumption in memory pool of all systems at two scales of benchmarks.

records, which exacerbates the chain walking in FaRMv2-DM to cause high overheads. (2) The design of FaRMv2 consumes a dedicated RTT to lock the read-write data, but Motor enables to batch the locking and CVT/value read requests to save RTTs. (3) The design of FaRMv2 uses two RTTs to commit the backups and primaries, while Motor updates all replicas together in one RTT. Moreover, FORD can also achieve lower latency than FaRMv2-DM by alleviating the read overhead, but FaRMv2-DM allows more concurrency in multi-versioning to improve the throughput.

7.5 Memory Overhead

We present the memory overheads of all systems in the memory pool using two different scales of benchmarks. Scale-1 (or Scale-2): `TPCC` contains 24 (or 48) warehouses; `TATP` has 2M (or 4M) subscribers; `SmallBank` has 10M (or 20M) accounts; `KVS` stores 10M (or 20M) KV pairs with skewness 0.99 and RW-ratio 80%. Scale-1 is the default configuration in § 7.1.

As shown in Fig. 12, FORD exhibits the lowest memory overhead by storing only one version of data. Due to supporting multi-versioning, Motor and FaRMv2-DM consume larger memory space than FORD. Nevertheless, Motor saves memory space in three aspects: (1) maintaining the actually modified attributes rather than full values for different versions; (2) appropriately estimating the size of attribute bar without wasting space; and (3) configuring suitable VNums for different workloads without storing unnecessary versions. For example, Motor supports 4 versions of data in `TPCC`, but only consumes 1.45×, instead of 4×, of memory space over FORD. Such memory saving is also shown in other benchmarks. In `TATP`, Motor only incurs 17.3% higher memory overhead than FORD, since only 16% of transactions perform updates and the modified attributes are small. In `SmallBank` and `KVS`, Motor respectively consumes 32.7% and 37.7% higher memory space than FORD, since `SmallBank` and `KVS` are write-intensive and require more versions than `TATP`. FaRMv2-DM suffers from 14.6%-22.8% higher memory overhead than Motor due to two reasons. First, FaRMv2 stores a full-sized value for each version, while Motor only stores the modified attributes of values. Second, FaRMv2 requires pointers to link old versions in its version chain, while Motor does not need such pointers since our CVT structure consecutively stores all the versions. Moreover, Fig. 12b shows that when the benchmark scale increases, the gap of space consumption between Motor and FORD generally keeps stable in all benchmarks. This demonstrates that our reduction of memory overhead still works even if the workload scale becomes larger. In summary, Motor trades some extra memory space to achieve better performance than the single-versioning design, while also reducing the memory overhead as much as possible.

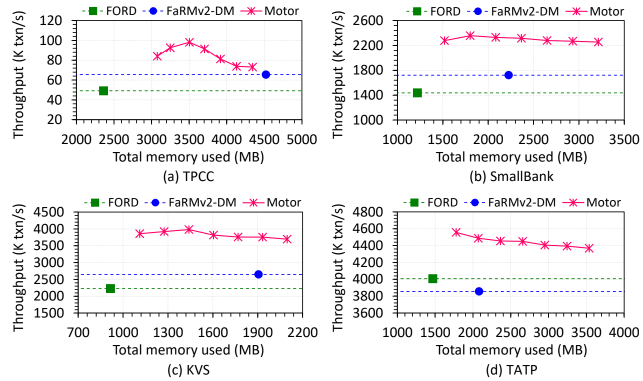


Figure 13: The comparisons of transaction throughput when varying Motor memory footprint by changing VNum.

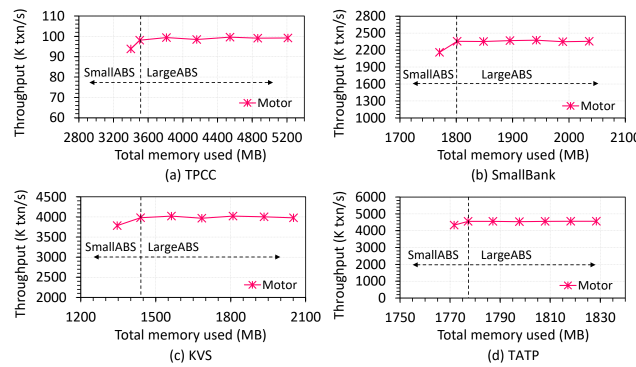


Figure 14: The transaction throughput of Motor when varying the memory footprint by changing ABS.

7.6 Varying Motor Memory Footprint

We study how Motor performs when varying the memory footprint based on the benchmark Scale-1 (§ 7.5). In the memory pool, since the full values always exist to provide complete user data, we vary Motor memory footprint by changing the number of versions (VNum) and the attribute bar size (ABS). As Motor has significantly reduced the memory overhead, the room to further decrease memory footprint is limited. For example, Motor only reserves 2 versions of data in `TATP`. This is the minimal number of versions for multi-versioning. Hence, in `TATP`, we increase VNum up to 8 to increase memory footprints. For other benchmarks, since their suitable VNums are larger than 2, we decrease (and increase) VNum from the suitable VNum to 2 (and 8) to vary memory footprints. When changing VNum (2–8), the corresponding ABS is estimated using the formula in § 4.2. Moreover, to vary ABS, we fix VNum to the suitable VNum in each benchmark, and (1) increase ABS to 2–6× of the estimated ABS using the suitable VNum, and (2) decrease ABS to 1× of the sum of different TotAttrSizes per transaction. Fig. 13–16 show the transaction throughput and latency of Motor when varying memory footprints. We also report the performance and memory footprints of FORD and FaRMv2-DM for comparisons.

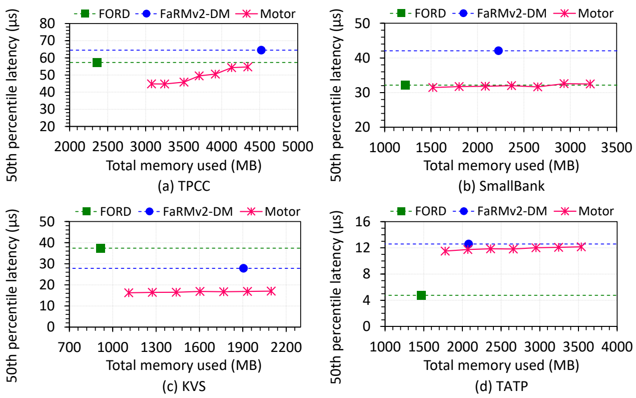


Figure 15: The comparisons of the 50th percentile latency when varying Motor memory footprint by changing VNum.

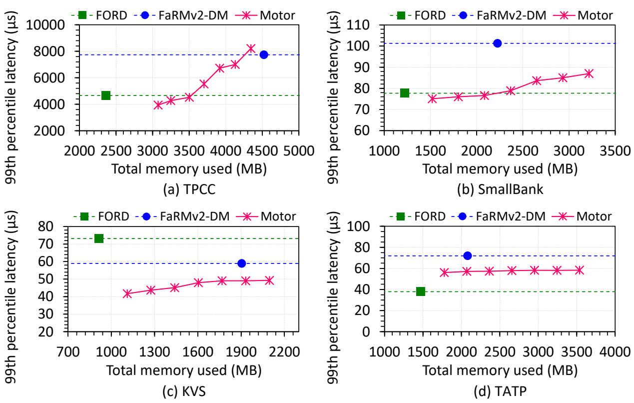


Figure 16: The comparisons of the 99th percentile latency when varying Motor memory footprint by changing VNum.

As shown in Fig. 13, when decreasing VNum from the suitable value, the memory footprints of Motor are reduced by up to 22.8% and are close to FORD on many workloads. Through reducing the memory footprint to contain less versions, Motor still achieves higher throughput than FORD and FaRMv2-DM. The reason is that compared with FORD, (1) Motor reserves more than one version to avoid blocking reads and reduce transaction aborts; (2) Motor does not need to additionally write undo logs and the read-only transactions do not need to validate versions with multi-versioning. Moreover, compared with FaRMv2-DM, (1) our CVT structure avoids chain walking to reduce latency; (2) our MVCC protocol saves RTTs via efficient request batching (§ 7.4). When slightly increasing VNum (e.g., from 4 to 6 in `KVS`), Motor still consumes less memory than FaRMv2-DM thanks to only storing necessary modifications in the delta area. Hence, compared with FaRMv2-DM, Motor can store more versions using a smaller amount of memory. In fact, when VNum increases from 2 to 8 (4×), the Motor memory footprint only increases by 1.4×/2.1×/2×/1.9× on `TPCC`/`SmallBank`/`TATP`/`KVS`. Fig. 14 shows that when fixing VNum and reducing ABS from the suitable ABS, the throughput decreases, since a small-sized attribute bar would result in more than one Vcells being invalidated in garbage collection to increase aborts. However, when increasing ABS from the suitable ABS, the throughput generally keeps stable, since the transaction aborts are hardly reduced. This demonstrates the efficiency of our estimation on ABS, i.e., reserving an exact and sufficient size for the attribute bar without wasting memory. Fig. 15 and 16 show that the latency of Motor grows when increasing VNum to enlarge the memory footprint, since large-sized CVTs increase the transmission latency. Nevertheless, Motor still exhibits lower latency than FaRMv2-DM by using the CVT to obtain all versions in a single read. In `TATP`, FORD achieves the lowest latency due to consuming less RTTs to fetch data, as analyzed in § 7.4. But in other benchmarks, Motor shows lower latency than FORD at suitable VNums due to eliminating the overheads of writing logs for read-write transactions and validating versions for read-only transactions. In summary, these results demonstrate the benefits of Motor over state-of-the-art systems when varying Motor memory footprint.

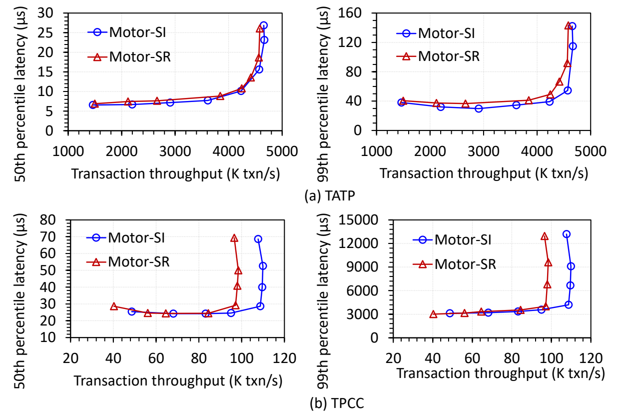


Figure 17: The transaction throughput and latency on `TATP` and `TPCC` benchmarks when using different isolation levels.

7.7 Performance of Different Isolation Levels

Motor supports two isolation levels, i.e., serializability (SR) and snapshot isolation (SI). Fig. 17 show that Motor-SI generally achieves lower latency and higher throughput than Motor-SR on both read-intensive (`TATP`) and write-intensive (`TPCC`) workloads by eliminating the validation phase for read-write transactions. Compared with `TATP`, Motor-SI shows higher throughput improvement in `TPCC`, since `TPCC` accesses more read-only data per transaction and features higher read-write contentions, thus allowing more throughput improvement when relaxing the isolation requirement.

7.8 Using PM in Memory Pool

Both DRAM and persistent memory (PM) can be used in a memory pool [69, 86]. We leverage six 128GB Intel Optane PM modules in each memory node to evaluate the performance of Motor on `TPCC`. We use RDMA `READ`-after-`WRITE` to flush the written data from remote RNIC to PM for remote data persistency [84]. Fig. 18 shows that the throughput only decreases by 13.1% on PM due to the limited PM bandwidth [80, 84]. The results demonstrate that Motor efficiently works on both DRAM and PM, thus offering good portability for applications to run on different types of memory devices.

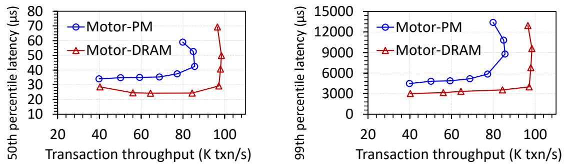


Figure 18: The transaction throughput and latency on TPCC benchmark when using DRAM and PM in the memory pool.

7.9 Fault Tolerance

We leverage TPCC to show the resilience of Motor under coordinator failures in compute pool and replica failures in memory pool. We report the instantaneous transaction throughput in 1 ms interval over time (the crash occurs at time 0).

Fig. 19a shows the throughput timeline of recovering coordinators. We run 84 coordinators and 60 of them fail at the same time. Motor then generates 60 new coordinators and establishes network connections, which consumes about 170 ms. Afterwards, the new coordinators take over the remaining tasks. In Motor, each coordinator writes local operation logs to record the operations during execution. These operation logs consume very small space (up to 556B per transaction) and the log space can be reused across transactions. The new coordinators use the operation logs of failed ones to resume in-flight commits and unlock CVTs to avoid starvation. After recovery, Motor regains peak throughput.

Fig. 19b shows the results of recovering replicas. Considering that the CUSTOMER table is frequently used, we respectively allow the primary and one backup of CUSTOMER to fail, i.e., cannot be accessed. A small portion of transactions that do not access the failed replicas are normally executed, and hence the throughput does not become 0. Motor handles the primary failure by promoting a backup as the new primary and adding a backup. Motor tolerates the backup failure by adding a backup. Recovering the primary consumes more time, since Motor needs to change the view of primaries for coordinators, and the new primary is not visible until the updates are committed into alive replicas. Adding a backup requires data migration, during which Motor allows a memory node to use RDMA WRITE to transmit DB tables, CVTs, and attribute bars to another memory node. Write requests to the replicas involved in migration are blocked to guarantee the data consistency among replicas. Since the CUSTOMER table is large, the migration consumes nearly 200 ms. We also examine that if a small DISTRICT table fails, the migration consumes only 1.1 ms. Further optimization on migration is out of our scope. In practice, our ms-scale recovery is acceptable given that prior systems [27, 64, 66] also provide ms-scale recovery.

8 Related Work

Fast Distributed Transactions. Fast distributed transaction processing is a key pillar in distributed systems. Many systems use RDMA to process transactions [22, 26, 27, 39, 41, 58, 64, 77, 78]. Some studies transform a distributed transaction to a local one to reduce the communication overheads [19, 40, 52]. Some protocols on concurrency control [55, 74, 79, 82] and

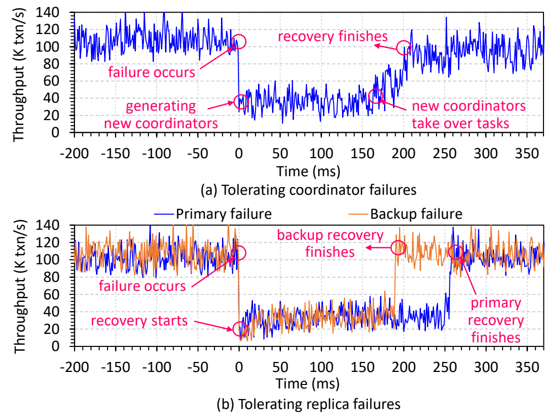


Figure 19: The Motor's transaction throughput on TPCC over time under (a) coordinator failures and (b) replica failures.

data replication [83] are proposed to improve the performance. The above systems work on the monolithic architecture, while our Motor targets on the disaggregated architecture.

Memory Disaggregation. Memory disaggregation improves the resource utilization. Existing studies explore memory disaggregation in many areas, such as hardware designs [35, 50, 51], operating systems [65], indexes [53, 75, 86], key-value stores [45, 49, 66, 69], networking [29, 67], erasure coding [47, 85], swapping [15, 20, 33, 62], and memory managements [14, 46, 48, 54, 70, 72, 73]. In fact, Motor focuses on transaction processing, which is orthogonal to the above systems. Though FORD [84] supports transactions on disaggregated memory, it adopts single-versioning, which limits the concurrency and incurs high logging overheads. Unlike FORD, Motor enables multi-versioning to address these limitations.

Multi-Versioning Schemes. Multi-versioning schemes have been adopted to support distributed transactions. They focus on high-performance MVCC protocols [28, 43, 57, 64], timestamp generations [38, 76, 81], garbage collections [16, 44], and verifications [21]. These systems are designed for traditional monolithic servers, which do not fit the disaggregated memory. Unlike these studies, our CVT structure and distributed transaction protocol efficiently support multi-versioning on the disaggregated memory.

9 Conclusion

This paper proposes Motor, an efficient distributed transaction processing system for multi-versioning in the context of disaggregated memory. Motor proposes a new consecutive version tuple structure to efficiently organize multiple versions of data in memory pool. On top of this, Motor designs a fully one-sided RDMA-oriented MVCC protocol to accelerate transactions. Extensive experimental results demonstrate that Motor significantly improves the transaction throughput and reduces the latency with moderate memory overhead.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to anonymous reviewers for their constructive suggestions and feedback.

References

[1] Telecom application transaction processing benchmark. http://tatpbenchmark.sourceforge.net, 2011.

[2] Intel® rack scale design architecture. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf, 2018.

[3] Vmware Research: Remote memory. https://research.vmware.com/projects/remote-memory, 2021.

[4] Smallbank benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank, 2022.

[5] Precedence graph. https://en.wikipedia.org/wiki/Precedence_graph, 2023.

[6] Rdma aware networks programming user manual v1.7. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/transport+modes, 2023.

[7] Compute express link®. https://www.computeexpresslink.org, 2024.

[8] Intel® Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2024.

[9] MySQL: The world's most popular open source database. https://www.mysql.com, 2024.

[10] PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org, 2024.

[11] Serializability. https://en.wikipedia.org/wiki/Database_transaction_schedule#Serializable, 2024.

[12] Snapshot isolation. https://en.wikipedia.org/wiki/Snapshot_isolation, 2024.

[13] Tpc-c benchmark. http://www.tpc.org/tpcc, 2024.

[14] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 775–787. USENIX Association, 2018.

[15] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[16] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory MVCC systems. *Proc. VLDB Endow.*, 13(2):128–141, 2019.

[17] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[18] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[19] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, 2018.

[20] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 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*, pages 79–92. ACM, 2021.

[21] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying vmvcc, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 871–886. USENIX Association, 2023.

[22] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 26:1–26:17. ACM, 2016.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 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*, pages 143–154. ACM, 2010.

[24] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254. ACM, 2013.

[26] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[27] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[28] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and general datacenter transactions for on-disk databases. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 343–360. USENIX Association, 2023.

[29] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[30] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[31] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[32] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, 2010.

[33] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[34] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. ukharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 101–120. USENIX Association, 2022.

[35] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 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*, pages 417–433. ACM, 2022.

[36] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, 2015.

[37] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 175–188. USENIX Association, 2008.

[38] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. Aurogon: Taming aborts in all phases for distributed In-Memory transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 217–232, Santa Clara, CA, February 2022. USENIX Association.

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA,*

USA, November 2-4, 2016, pages 185–201. USENIX Association, 2016.

[40] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 145–161. ACM, 2021.

[41] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 297–312. ACM, 2018.

[42] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 312–313. ACM, 2009.

[43] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.

[44] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1307–1318. ACM, 2016.

[45] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023–4037, 2022.

[46] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 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*, pages 488–504. ACM, 2021.

[47] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 181–198. USENIX Association, 2022.

[48] 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. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 574–587. ACM, 2023.

[49] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 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*, pages 99–114. USENIX Association, 2023.

[50] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 267–278. ACM, 2009.

[51] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[52] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[53] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou and. 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*, pages 553–566. USENIX Association, 2023.

[54] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 757–773. ACM, 2020.

[55] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 479–494. USENIX Association, 2014.

[56] MySQL. Transaction isolation levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html, 2024.

[57] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689. ACM, 2015.

[58] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 97–108. ACM, 2019.

[59] Oracle. Transaction isolation levels. https://www.oreilly.com/library/view/java-programming-with/0596000871/0596000871_orasqlj-CHP-9-SECT-2.html, 2024.

[60] PostgreSQL. Transaction isolation. https://www.postgresql.org/docs/current/transaction-iso.html, 2024.

[61] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. RDMA is turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 71–85. USENIX Association, 2022.

[62] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 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*, pages 315–332. USENIX Association, 2020.

[63] SQL Server. Set transaction isolation level. https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver16, 2023.

[64] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[65] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 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*, pages 69–87. USENIX Association, 2018.

[66] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 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*, pages 81–98. USENIX Association, 2023.

[67] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki-Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 255–270. USENIX Association, 2019.

[68] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 7th Edition*. McGraw-Hill Education, 2019.

[69] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[70] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 306–324. ACM, 2017.

[71] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13,*

Farmington, PA, USA, November 3-6, 2013, pages 18–32. ACM, 2013.

[72] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, November 2020.

[73] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 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*, pages 35–53. USENIX Association, 2022.

[74] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 198–216. USENIX Association, 2021.

[75] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1033–1048, 2022.

[76] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 357–372. USENIX Association, 2021.

[77] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 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*, pages 233–251. USENIX Association, 2018.

[78] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 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*, pages 87–104. ACM, 2015.

[79] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 279–294. ACM, 2015.

[80] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[81] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[82] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 511–526. ACM, 2020.

[83] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM, 2015.

[84] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 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*, pages 51–68. USENIX Association, 2022.

[85] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 55–71. USENIX Association, 2022.

[86] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.