CREST: High-Performance Contention Resolution for Disaggregated Transactions

Qihan Kang
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences
Beijing, China

Mi Zhang
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong
Hong Kong, China

Yongkang Hu
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences
Beijing, China

Abstract

Distributed transaction systems can leverage memory disaggregation for efficient resource scaling, yet they experience significant performance degradation under high-contention workloads. We present CREST, a disaggregated transaction system that efficiently manages high-contention transaction workloads in disaggregated memory architectures via three key techniques: (i) cell-level concurrency control, which achieves more fine-grained transaction concurrency than existing record-level approaches and reduces remote access latencies using a metadata-aggregated record structure; (ii) localized execution, which allows compute nodes to operate on local uncommitted results to reduce blocking time; and (iii) parallel commits, which parallelize commit operations under transaction dependencies. Evaluation shows that CREST achieves a throughput gain of up to 1.92× over state-of-the-art systems under high-contention workloads.

CCS Concepts: • **Computer systems organization** → **Architectures**.

Keywords: Distributed transactions, disaggregated memory, RDMA

ACM Reference Format:
Qihan Kang, Mi Zhang, Patrick P. C. Lee, and Yongkang Hu. 2026. CREST: High-Performance Contention Resolution for Disaggregated Transactions. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790148

1 Introduction

Distributed transaction systems [16] ensure ACID properties in relational data processing across multiple servers, and are critical for applications requiring strong consistency and fault tolerance (e.g., e-commerce and financial services [27, 54, 64]). However, traditional distributed transaction systems build on monolithic servers, where compute resources (e.g., CPU) and memory resources (e.g., DRAM and persistent memory) are tightly coupled. Such rigid coupling limits resource scalability and flexibility [29, 40, 46].

Memory disaggregation [19, 48, 52, 55] enables efficient resource scaling by decoupling compute and memory resources and interconnecting them with high-speed networks, such as Remote Direct Memory Access (RDMA). It avoids resource over-provisioning, reduces expensive data migration [24], and alleviates single-node bottlenecks [60], thereby improving resource utilization and performance [47]. Recent studies [32, 33, 67, 68] demonstrate significant performance gains when applying memory disaggregation to distributed transaction processing under typical workloads.

However, existing studies do not specifically address high-contention workloads, which are actually prevalent in real-world applications. Contention occurs when concurrent transactions access the same data item in conflicting modes (e.g., read-write or write-write conflicts). Frequently accessed data items further exacerbate such contention. Our analysis (§2.3) reveals that state-of-the-art disaggregated memory systems, FORD [68] and Motor [67], adopt record-level concurrency control and experience frequent transaction aborts in high-contention workloads, even though the concurrent transactions may access different column fields of the same record. Also, both FORD and Motor acquire locks for the whole transaction to ensure serializability, yet this increases the blocking time of other concurrent transactions.

We present CREST, a disaggregated transaction system designed for high-performance contention resolution of transactions in disaggregated memory architectures. CREST builds on three key design techniques:

- *Cell-level concurrency control:* CREST applies fine-grained concurrency control at the granularity of *cells* (i.e., individual column fields of a record). This design allows transactions to simultaneously access different fields of the same record, thereby improving concurrency. Similar approaches can be applied to storage systems that already adopt sub-record units for concurrency control (e.g., BigTable [5]). However, cell-level operations aggravate metadata access overhead, so CREST introduces a specialized record structure that aggregates metadata to reduce the number of RDMA calls.

- *Localized execution:* CREST allows transactions to operate on local uncommitted execution results in the same compute node, so as to reduce the blocking time of concurrent transactions. It employs *pipeline execution* to parallelize transaction executions for high performance, while ensuring serializability.
- *Parallel commits:* CREST parallelizes transaction commits to remote memory using *dependency-tracking redo-logging* and *last-writer-wins* mechanisms, so as to ensure serializability under transaction dependency constraints.

We show that CREST maintains ACID guarantees and incurs limited space and RDMA communication overhead. We implement CREST and evaluate it against FORD [68] and Motor [67] under various transaction workloads. CREST achieves a throughput gain of up to 1.92× under high-contention workloads. Our CREST prototype is open-sourced at: **https://github.com/adslabcuhk/crest**.

2 Background and Motivation

2.1 Disaggregated Memory Architectures

Disaggregated memory architectures decouple compute and memory resources into two types of independent entities: (i) *compute nodes*, which have powerful CPUs for executing application logic but limited local memory for metadata storage or caching, and (ii) *memory nodes*, which have abundant memory space but limited computing power only enough for basic tasks (e.g., network connectivity and memory management) [38, 46, 48, 57, 70]. We refer to the collection of compute nodes as a *compute pool* and that of memory nodes as a *memory pool*. In this work, we assume that both pools are interconnected via high-speed network fabrics based on RDMA, which offers low-latency remote access (e.g., 3-5 μs) and high bandwidth (e.g., about 100 Gbps [14]).

We note that *one-sided RDMA primitives* (e.g., READ, WRITE, CAS (compare-and-swap)) allow compute nodes to directly access data in memory nodes without involving the remote CPU processing of memory nodes. This property is particularly well-suited for disaggregated memory architectures, where memory nodes often have limited processing capabilities. One-sided RDMA primitives have been extensively studied in academic research [32, 33, 67, 68] (see §9 for details) and are reportedly supported for transactions in production disaggregated memory architectures (e.g., ByteDance's veDB [49], Alibaba's PolarDB-MP [63], and Huawei's GaussDB

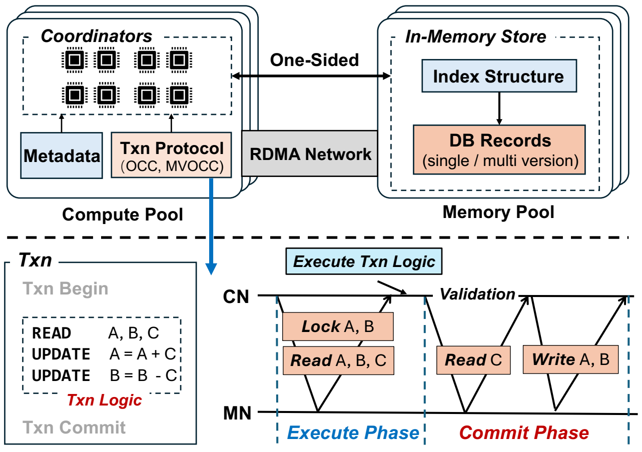


Figure 1. Example of transaction processing under disaggregated memory architectures (CN: compute node; MN: memory node).

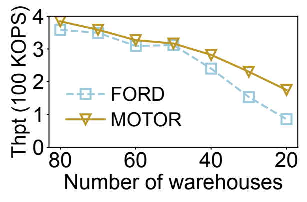
[28]). Motivated by the potential of one-sided RDMA primitives, this work examines the design and feasibility of achieving high-performance contention resolution for disaggregated transactions based on one-sided RDMA primitives.

2.2 Disaggregated Transaction Processing

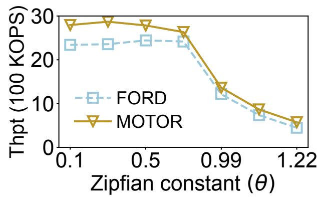
Figure 1 depicts the architecture of a disaggregated transaction system based on memory disaggregation. In the memory pool, database records are stored in either single-version [33, 68] or multi-version [67] formats and indexed using optimized data structures [57, 70]. To mitigate the network overhead due to RDMA's one-sided operations, state-of-the-art systems adopt specific storage layouts. For example, Motor [67] employs a consecutive version table to eliminate chain traversal when accessing multi-versioned records [45]. In addition, to ensure fault tolerance, state-of-the-art systems (e.g., FORD [68] and Motor [67]) adopt $(f + 1)$-primary-backup replication, where each record is replicated across f backup memory nodes and updated synchronously upon transaction commits.

Each compute node deploys multiple *coordinators* to process transaction requests from clients. Each request comprises a *read-write set*, which contains records to be modified, and a *read-only set*, which contains records for retrieval. To enforce serializability, the coordinators employ concurrency control in transaction processing. For example, FORD [68] and Motor [67] use a variant of Optimistic Concurrency Control [25, 53, 65] for disaggregated memory architectures. Processing a transaction is done in two phases (Figure 1):

- *Execution phase*: A coordinator first acquires locks (using CAS) for records in the read-write set (e.g., A and B) and reads all relevant records (e.g., A, B, and C) from the memory pool; locking is needed to prevent different transactions from simultaneously modifying the same record. It then executes the transaction, modifies records locally, and stores the results in the memory of compute nodes.

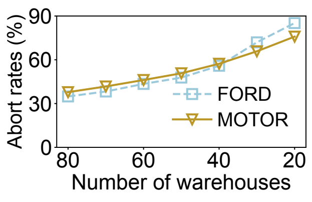


(a) TPC-C

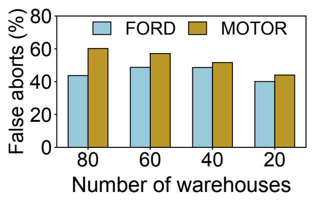


(b) SmallBank

Figure 2. Throughput (in thousand operations per second (KOPS)) of FORD and Motor under different contention levels.



(a) Abort rates



(b) False abort rates

Figure 3. Transaction aborts of FORD and Motor under TPC-C.

- *Commit phase*: The coordinator performs *validation* on the transaction to check for potential conflicts with other transactions during the execution phase. It checks the version numbers and lock states of records in the read-only set to ensure no modification from other transactions. If validation succeeds, the coordinator writes logs to the memory pool, updates all modified records (including replicas) in the memory pool, and releases the locks, so as to make the changes visible; otherwise, the coordinator aborts the transaction.

2.3 Motivating Experiments

Transaction contention is prevalent in real-world applications (e.g., due to hotspots), thereby significantly reducing throughput and increasing response latencies in transaction systems [1, 43, 59, 66]. We show via experiments the performance degradation of two state-of-the-art disaggregated transaction systems, FORD [68] and Motor [67], under varying contention levels.

We consider two representative transaction benchmarks, TPC-C [51] and SmallBank [3]. TPC-C is an e-commerce benchmark that exhibits high contention in the `warehouse` table, which is accessed by 92% of transactions. We vary the contention level by issuing transactions to a varying number of warehouses from 80 (i.e., low contention) to 20 (i.e., high contention). SmallBank simulates banking operations (e.g., transfers and deposits). We configure the access to accounts with varying skewness, following a Zipf distribution with a Zipfian constant θ varied from 0.1 to 1.22 as observed in production workloads [6]. We use two compute nodes, each running 60 coordinators, and two memory nodes.

Figure 2 plots the throughput of FORD and Motor under TPC-C and SmallBank workloads. In TPC-C, both FORD and Motor achieve high throughput at around 400 KOPS under low contention (80 warehouses). However, as the contention increases (20 warehouses), FORD's throughput drops by 71.2% to 105 KOPS only, while Motor, albeit benefiting from its multi-version design, still has a throughput drop of 57.3%. In SmallBank, the throughput trends are similar, where FORD and Motor have a throughput drop by 81.7% and 80.1% under high contention ($\theta = 1.22$), respectively.

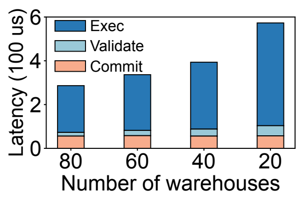
We argue that there are two root causes of our observed performance degradation.

(i) Frequent false conflicts in record-level concurrency control. Existing record-level concurrency control is a coarse-grained approach and can introduce frequent *false conflicts*, in which concurrent transactions access different column fields of the same record, but are treated as conflicts under record-level concurrency control and unnecessarily abort transactions; accordingly, we define *true conflicts* as the transactions that access the same column field of the same record. We measure the *abort rate*, defined as the ratio of the number of all aborted executions (due to both false and true conflicts) to the total number of transactions, under TPC-C across different contention levels. Figure 3(a) shows that the abort rates for FORD and Motor increase with contention, and reach 75.9% and 85.2% at 20 warehouses, respectively. A significant portion of these aborts are false conflicts. Figure 3(b) shows that the *false abort rates* (i.e., the fraction of aborts caused by false conflicts) under TPC-C reach 44.1% for Motor and 40.7% for FORD. In modern OLTP workloads with multi-attribute tables, transactions often access only specific columns instead of whole records [2, 17, 41], implying that false conflicts are common under record-level concurrency control. For example, in TPC-C's `warehouse` table, `NewOrder` transactions only read the columns of identification information (e.g., the `name` or `location` columns), while `Payment` transactions update the `balance` column.

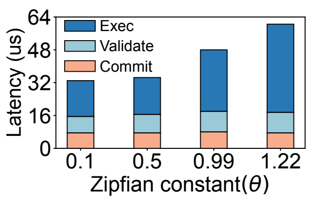
Note that SmallBank exhibits a zero false abort rate, as all its transactions operate on the same column. Nevertheless, given the complexity of modern OLTP schemas, mitigating false conflicts remains necessary under high contention.

(ii) Long blocking time increases transaction latency. True conflicts degrade performance as they prolong blocking time. Current approaches enforce *strict locking*, which requires transactions to hold locks on their read-write sets until commits to prevent concurrent transactions from accessing uncommitted data. In disaggregated memory architectures, committing a transaction involves multiple network round-trips for undo-logging, record updates, and lock releases, thereby amplifying blocking time under contention.

To demonstrate, we measure the latencies of committed transactions under TPC-C and SmallBank by decomposing a transaction into execution, validation, and commit operations, where validation and commit operations together form the commit phase (§2.2). For brevity, we focus on Motor, while FORD shows similar trends. Figure 4 shows that



(a) TPC-C



(b) SmallBank

Figure 4. Latency breakdown of Motor.

the execution latency dominates the total latency in Motor and increases significantly under high contention, as more transactions need to wait for committed results and cannot be concurrently executed. In TPC-C, the total latency doubles as the number of warehouses decreases from 80 to 20, with the execution latency increasing by 119.8%. In SmallBank, the execution latency increases by 152.9% as θ increases from 0.1 to 1.22.

3 Design Overview

Main idea. CREST aims to mitigate false conflicts due to record-level concurrency control and prolonged blocking from strict commit procedures. It builds on three key techniques: (i) *cell-level concurrency control* (§4), which eliminates false conflicts by performing fine-grained concurrency control at the granularity of a *cell*, which represents a column field within a record and the smallest accessible unit in transactions; (ii) *localized execution* (§5), which exposes uncommitted results within a compute node and allows transactions to continue execution without waiting for commits to the memory pool; and (iii) *parallel commits* (§6), which parallelizes commits for high performance under transaction dependency constraints.

The above techniques, however, pose new challenges to RDMA communication and transaction correctness:

- *RDMA communication:* Cell-level concurrency control introduces metadata to track each cell's status, thereby aggravating metadata access via RDMA. In particular, locking multiple cells within a record incurs separate `CAS` calls for each cell's lock, while validating multiple cells needs to fetch multiple cell versions via multiple `READ` calls. Thus, cell-level operations can amplify RDMA communication overhead, leading to degraded system performance.
- *Transaction correctness:* With localized execution and parallel commits, CREST needs to resolve local (within the same compute node) and global (across compute nodes) conflicts to maintain correct execution ordering. During commits, the memory pool should maintain consistent states with local execution results, especially when multiple transactions in the same compute node update the same record, and only the latest valid version in the memory pool should be updated.

Architecture. Figure 5 depicts the architecture of CREST.

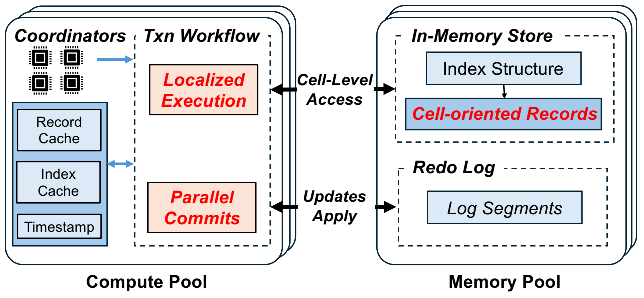


Figure 5. Overall architecture of CREST.

CREST stores database records in the memory pool with a specialized record structure, which mitigates RDMA overhead during cell-level operations (§4). In each compute node, CREST deploys multiple coordinators to process transactions in two phases: localized execution and parallel commits. During localized execution, the coordinators within the same compute node execute transactions locally and write uncommitted results to a *record cache*, so that other transactions can access the uncommitted results early without being blocked from fetching data from the memory pool. During parallel commits, CREST applies dependency-tracking redo-logging to preserve transaction dependencies for correctness.

Design assumptions. CREST supports transactions invoked via stored procedures (i.e., pre-compiled database commands) with user-configurable input parameters, a common feature in transaction systems [22, 35, 36, 50]. This offers two benefits. First, CREST can leverage stored procedures to readily identify accessed columns in each table for cell-level operations. Second, by analyzing input parameters, CREST can determine if a record is read-only or will be updated, so as to manage data correctly in localized execution.

4 Cell-level Concurrency Control

4.1 Record Structure

CREST allows concurrent transactions to operate on distinct cells within the same record. It maintains a version for each cell, co-located with the cell value within a cacheline. This ensures atomic access via a single RDMA call from the compute pool [10]. Each cell version comprises a 2-byte *epoch number* (`EN`) and a 6-byte *commit timestamp* (`TScommit`). A coordinator increments the epoch number with each cell update to track modifications. The epoch number is used for validation during the commit phase, while the commit timestamp forms the global order of committed transactions.

To mitigate metadata access overhead in cell-level concurrency control, CREST adopts a specialized record structure. Specifically, each record contains a *record header* storing metadata and an array of cells, while multiple records are stored continuously in the memory pool and accessed via a hash index [8, 67, 68]. Figure 6 shows the record structure in CREST. The record header comprises multiple fields:

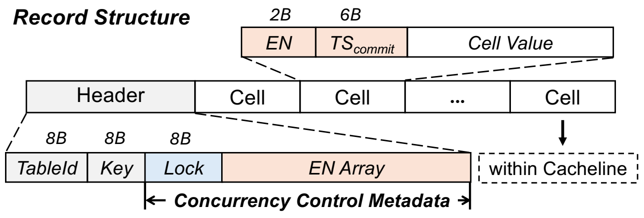


Figure 6. Record structure in CREST.

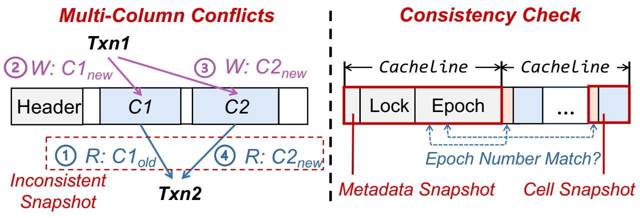


Figure 7. Example of resolving read-write conflicts in CREST.

TableID, Key, Lock, and an epoch number array (EN array). The TableID and Key fields uniquely identify a record, while the Lock and EN array fields resolve write-write and read-write conflicts, respectively. In particular, the EN array consolidates the epoch numbers of all cells in the same record, so as to allow efficient multi-cell validation (§4.2).

4.2 Concurrent Access

Each coordinator accesses and updates cells via one-sided RDMA operations, including locking and fetching relevant cells in the execution phase, and validating and updating the cells in the commit phase. To reduce RDMA communication overhead due to per-cell processing, CREST proposes *multi-cell locking* and *multi-cell validation.*

Multi-cell locking. To mitigate locking overhead, CREST allocates one bit per cell to represent lock status and aggregates the bits of all cells in a record into the 8-byte Lock field in the record header. This enables a coordinator to atomically modify multiple cell locks using a single masked CAS (masked-CAS) primitive provided by RDMA NIC hardware. The masked-CAS primitive allows bit-level comparisons and modifications within an 8-byte value. For example, to lock the second and fourth cells, a coordinator issues a masked-CAS call, which sets the CAS masks to 0101...000. To release cell locks, the coordinator clears the corresponding bits from 1 to 0 using another masked-CAS call.

Multi-cell validation. To reduce validation overhead, CREST consolidates epoch numbers of all cells into a contiguous EN array in the record header, such that each epoch number is consistently updated with the corresponding cell version upon each transaction commit. Such consolidation allows efficient validation of multiple read-only cells via a single RDMA READ to the record header. However, the 2-byte epoch number EN may lead to overflow (after $2^{16} = 65{,}536$ updates) and incorrect validation. Thus, CREST implements a *time threshold* mechanism, inspired by Sherman [57]. Specifically, if the duration between the first read in the execution phase and the validation exceeds a threshold, the coordinator reads the entire record and uses the commit timestamp for validation. CREST currently sets the threshold to 65,536μs, assuming that each transaction lasts no more than 1 μs (note that the RDMA communication latency is typically around 2 μs). Such a conservative threshold ensures validation correctness, as a rollover of the 2-byte EN field requires at least 65,536 transactions. As most transactions are short-lived and completed within the threshold, it is unlikely for them to read the entire record for validation (note that our evaluation shows no rollover of epoch numbers (§8)).

After validation, the coordinator consistently updates cell values and corresponding metadata in the record header during the commit phase. For each record, the coordinator first updates all modified cells, including both values and cell versions (i.e., EN and TScommit), then increments the corresponding epoch numbers in the EN array, and finally releases the locks. By leveraging RDMA's delivery order guarantee [57, 68], the above steps can be executed with a batch of ordered RDMA WRITE and masked-CAS calls in a single round-trip [67].

4.3 Resolving Cell Conflicts

Ensuring correctness under cell-level concurrency control is more complex than the record-level one. For example, in the left part of Figure 7, a transaction Txn1 updates cells C1 and C2, while another transaction Txn2 reads C1 and C2 concurrently. Under serializability, Txn2 should read either all old or all new values of both C1 and C2, but Txn2 may read C1's old value and C2's new value. Traditional approaches, such as cacheline versioning [8, 10, 37, 45] or wrapped versioning [57, 67], detect read-write conflicts for single objects only, but fail across multiple cells. Thus, CREST adopts *snapshot-based verification* to resolve intra-cell and inter-cell conflicts.

Intra-cell conflicts. CREST places each cell (with both cell version and value) within a cacheline to allow atomic RDMA READ and WRITE [10, 37]. This ensures that a transaction reads either the old or new version of a cell. For cells exceeding the cacheline size, CREST applies cacheline versioning [8, 10] to split a large cell into multiple cachelines.

Inter-cell conflicts. To detect read-write conflicts across cells, CREST uses the Lock and EN array fields in the record header, both within one cacheline, to generate a snapshot with an RDMA READ. As shown in the right part of Figure 7, CREST determines that all read cells are consistent if: (i) the locks of read cells are not acquired by other transactions and (ii) the epoch numbers in the record header snapshot match the epoch numbers in the cell snapshots. If both conditions hold, CREST obtains a consistent snapshot, as the locks of all updated cells can be released only after all cells are updated.

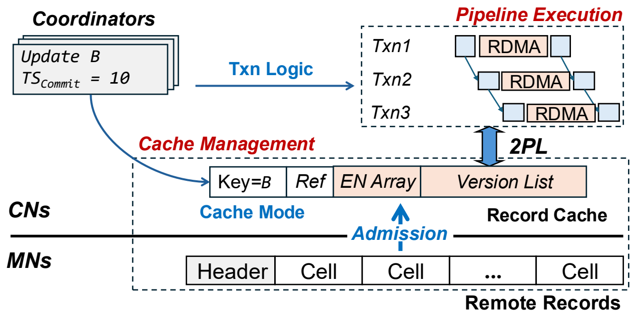


Figure 8. Localized execution in CREST.

4.4 Discussion

While CREST shares conceptual similarities with column-level concurrency control for web applications [69] (e.g., both employ fine-grained locking), CREST does not maintain metadata for an entire column as in column-level concurrency control. Instead, CREST associates metadata with each cell within a record, and we call this cell-level concurrency control.

To reduce metadata overhead caused by cell-level concurrency control, one solution is to apply cell-level concurrency control only to the cells with high contention. CREST inserts or deletes entire rows by acquiring all cell locks via an RDMA CAS. To support efficient deletion, CREST uses one spare bit in the record header's Lock field to mark whether the record has been logically deleted, and later frees the deleted records on memory nodes.

CREST aggregates cell metadata into a single cacheline (64 bytes), so the maximum number of supported cells is bounded. The 8-byte Lock field supports at most 64 cells, while the `EN array` supports up to 20 cells. This suffices for common OLTP workloads (§8.2). For tables exceeding 20 columns, CREST consolidates all cells beyond the 20th into a single large cell, but may reintroduce false conflicts and limit concurrency for wide tables. One potential improvement is to consolidate cells based on transactions' access patterns (e.g., grouping read-intensive cells) to mitigate conflicts.

5 Localized Execution

CREST uses localized execution to accelerate the processing of transactions running in the same compute node. The key idea is to allow a coordinator to locally operate on uncommitted record versions within the same compute node, instead of waiting for the commits of prior transactions and fetching the latest versions from the memory pool. Figure 8 shows the workflow of localized execution.

5.1 Local Data Management

Each compute node maintains fetched records and uncommitted record versions in a *record cache*, accessed by multiple coordinators. The record cache keeps multiple *local objects*, each corresponding to a record in the memory pool. A local object includes three fields to support localized execution across coordinators: (i) *reference counter*, which tracks the number of local transactions accessing the record; (ii) *epoch array*, which specifies the array of epoch numbers of all cells (i.e., `EN array` in the record header), and (iii) *version list*, which stores uncommitted versions of the record. All records are cached in either *read-write* or *read-only* mode. If a record is to be updated, it is cached in read-write mode, which involves acquiring cell locks (§4.2); otherwise, it is cached in read-only mode, which only requires fetching cells of the record. CREST tracks the numbers of local transactions accessing the record in read-only and read-write modes with `readers` and `writers` variables in the reference counter, respectively.

The record cache dynamically manages local objects to control memory usage. A local object is either created when its corresponding record is accessed for the first time, or destroyed if no transaction in this compute node accesses this record. A coordinator checks the record cache for the required records before executing a transaction. For any uncached record, the coordinator initiates cache admission by fetching the record from the memory pool. To prevent redundant I/O and local conflicts, CREST ensures that only one coordinator performs cache admission for each record.

Since local transactions may access uncommitted versions, CREST tracks dependencies between transactions to ensure correctness in the commit phase. For example, if transaction Txn2 reads or updates a version created by transaction Txn1, Txn2 commits only if Txn1 commits, or aborts if Txn1 aborts. CREST assigns each transaction a unique transaction ID before execution and embeds it in all created local versions. When a transaction accesses a local version, it adds the embedded ID to its own dependency list, which is checked during the commit phase to enforce consistency.

5.2 Pipelined Execution

A coordinator executes a transaction once all the required records are locally cached. CREST adopts two-phase locking (2PL) to ensure serializability across all local transactions, as 2PL excels in high-contention workloads [56]. However, applying 2PL directly to the whole transaction can extend lock holding time when a transaction involves *key dependencies* [66], where the next record's primary key depends on the current record's value. CREST proposes *pipelined execution*, which divides a transaction workflow into *execution blocks* and pipelines the processing of execution blocks. At a high level, CREST combines 2PL with *timestamp ordering* during localized execution, where it applies 2PL within an execution block and uses execution timestamps across different execution blocks to maintain consistent ordering. It groups operations with key dependencies within the same execution block and releases local locks at the end of an execution block, instead of holding all local locks for the entire transaction. Note that the execution blocks are pre-

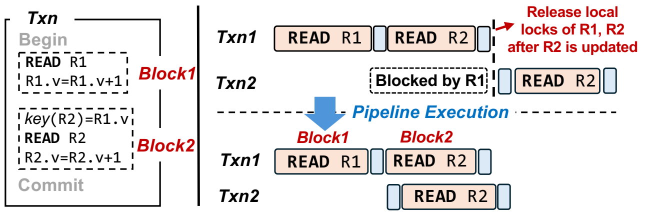


Figure 9. Example of pipelined execution. The assignment `Key(R2)=R1.v` sets R2's primary key as R1's updated value and introduces a key dependency. Thus, R2 can be fetched only after R1's value is updated.

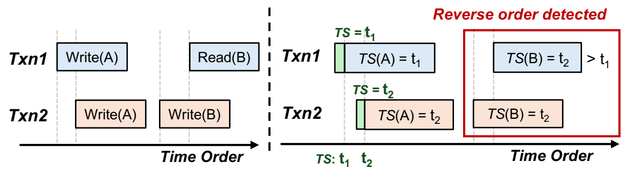


Figure 10. Example of block ordering coordination. The left part shows a reverse order across blocks in Txn1 and Txn2, while the right part shows that Txn1 detects a reverse order, where `TSexec` of B's latest version (t_2) is greater than its own `TSexec` (t_1).

defined by programmers based on the transaction logic and are generated during runtime.

Figure 9 shows an example of pipelined execution. CREST coordinates *inner-block execution* and *block ordering* to ensure serializability.

Inner-block execution. Within each block, CREST applies 2PL using local locks maintained by each local object. Before execution, a coordinator acquires all necessary local locks in ascending order of the tuple (`TableID`, `Key`) to prevent deadlocks [12]. For writes, the coordinator creates and appends a new local version to the version list. As a transaction may abort during the commit phase, CREST keeps all intermediate versions in the compute node without updating the record cache in-place. For reads, the coordinator accesses the latest local version (the tail of the version list). Once the coordinator detects key dependencies, the current block execution is completed. Then, the coordinator releases all locks acquired in the current block and starts fetching records for the next block.

Block ordering coordination. While 2PL ensures serializability within an execution block, the execution order of transactions must remain consistent across different blocks. As shown in Figure 10, both transactions Txn1 and Txn2 try to access records A and B: Txn1 updates A and reads B in its first and second blocks, respectively, while Txn2 updates A and B in its first and second blocks, respectively. Suppose that Txn2 depends on Txn1 (i.e., Txn1 should be executed before Txn2). However, the actual execution may lead to a conflicting dependency that violates serializability: Txn2 updates A based on Txn1's version in the first block, while Txn1 reads B based on Txn2's version in the second block.

To coordinate block ordering, CREST tags timestamps to locally written versions to detect conflicting dependencies across blocks during local execution. Each transaction is assigned an *execution timestamp* `TSexec`, which is generated from a monotonically increasing counter in the compute node and assigned after the coordinator acquires all local locks in the transaction's first block. The execution timing ensures a key ordering property: if Txn2 depends on Txn1 in the first block, then Txn1's `TSexec` must be less than Txn2's `TSexec`, as Txn1 acquires `TSexec` before releasing its local locks. The coordinator tags all local versions with the corresponding `TSexec`. For example, in Figure 10, the coordinator tags any locally written versions of Txn1 (i.e., A) with t_1 and those of Txn2 (i.e., A and B) with t_2. During block execution, if a transaction accesses a local version with a tagged timestamp greater than its own `TSexec` (e.g., when Txn1 reads B, B's tagged timestamp is t_2, which is greater than Txn1's own `TSexec` t_1), it implies that a later transaction has already updated the record and violates the order enforced by `TSexec`, so the coordinator aborts the transaction. Note that CREST only detects, instead of repairing, any reverse ordering, yet in practice, such reverse ordering rarely happens and local execution still improves overall performance by pipelining execution blocks (§8.4).

6 Parallel Commits

After processing all records in localized execution, a transaction enters the commit phase, which comprises validation, redo-logging, and applying updates to the memory pool. To reduce commit latency while ensuring correctness (i.e., the order of applying updates to the memory pool must be consistent with that of local execution), CREST parallelizes the commits of different transactions using *dependency-tracking redo-logging* and *last-writer-wins* mechanisms.

Validation. CREST validates a transaction by checking the epoch numbers of all involved records and any dependent transaction. Checking the epoch numbers ensures that the transaction's records have not been updated by other transactions in different compute nodes (note that all transactions in the same compute node are already serialized by local concurrency control). Specifically, for each record in a transaction's read-only set, the coordinator compares the epoch numbers of all cells accessed in the record cache during the execution phase with the latest epoch numbers of the record in the memory pool. If the two epoch numbers differ, the coordinator concludes that the read-only set has been modified by another transaction and the transaction aborts. Otherwise, the coordinator continues to check whether any dependent transaction aborts; if so, the current transaction also aborts. If the transaction passes the above checks, the coordinator considers the transaction *committable* (i.e., ready to apply updates to the memory pool), assigns a commit timestamp, and updates the `TScommit` field of all local versions.

Redo-logging. CREST uses dependency-tracking redo-logging to ensure atomicity during the commit phase, so that it can always roll forward committed transactions even though some dependent transactions crash. Specifically, for each transaction, the coordinator generates a *log entry* that contains the transaction ID, modification logs, and a list of dependent transaction IDs. It writes the log entry to the memory pool using an RDMA `WRITE`. In the memory pool, CREST allocates fixed-size *log segments* to each coordinator. Each log segment serves as a queue of log entries for transactions processed by a specific coordinator, such that the coordinator can append a new log entry to the log segment's tail. A transaction is committed if the transaction is committable during validation and its log entry persists in the memory pool. During crash recovery, CREST restores the memory pool to a consistent snapshot using the log entries from redo-logging.

Applying updates. CREST adopts a *last-writer-wins* mechanism to ensure serializability when applying updates to the memory pool. CREST uses the `writers` variable in the reference counter (§5.1) to determine the last writer. After confirming that a transaction is committable, the coordinator decrements `writers` to indicate the write completion. If `writers` reaches zero, the coordinator is identified as the last writer, and its local version is deemed the latest committable version. The last writer then updates the records in the memory pool and releases the locks.

7 Correctness and Overhead Analysis

In this section, we show that CREST provides ACID guarantees and incurs limited space and communication overhead.

7.1 ACID Guarantees

Atomicity. CREST ensures atomicity by applying all operations within a transaction or none at all via dependency-tracking redo-logging and last-writer-wins during parallel commits (§6). A transaction is committed only if: (i) it passes validation and is committable, and (ii) all cell updates are persistently logged. For concurrent updates to the same cell, the writer with the highest `TScommit` prevails. This ensures atomic visibility; that is, either all updates from a transaction are applied via redo-logging, or none are applied if the transaction is aborted.

Consistency. CREST maintains consistency with cell-level conflict detection (§4) and localized execution (§5). It resolves intra-cell conflicts using atomic cacheline updates, and detects inter-cell conflicts using the `Lock` and `EN array` fields in the record header. Before executing a transaction locally, CREST validates cached records to ensure that data reads reflect a consistent snapshot.

Isolation. CREST ensures isolation across concurrent transactions using timestamp ordering (§5.2 and §6) and concurrency control (§4). Each transaction is assigned an execution timestamp `TSexec` to determine the local execution order among different blocks and a commit timestamp `TScommit` to form the global order of committed transactions. CREST adopts 2PL to ensure serializability among all local transactions and uses epoch numbers (i.e., `EN`) to serialize transactions across different compute nodes. Note that exposing uncommitted execution results does not compromise serializability, as local executions within a compute node are correctly coordinated.

Durability. CREST ensures durability by writing a log entry containing all updates to the memory pool via RDMA `WRITE` before marking a transaction as committed and applying the updates (§6). All committed transactions are recoverable with redo-logging. CREST achieves fault tolerance via primary-backup replication, where each record has multiple replicas and updates are applied to all replicas.

Table 1. Space overhead in memory nodes.

Workload	FORD	Motor	CREST
(a) Metadata-only, without cacheline padding			
TPC-C	7.49%	58.75%	20.65%
SmallBank	39.84%	318.75%	42.19%
YCSB	9.66%	63.64%	27.27%
(b) With cacheline padding			
TPC-C	19.61%	63.92%	41.30%
SmallBank	100.00%	400.00%	100.00%
YCSB	45.45%	81.82%	45.45%

7.2 Space Overhead

CREST introduces extra metadata. Here, we analyze its space overhead in memory nodes and compute nodes.

Space overhead in memory nodes. Table 1 compares the space overhead of FORD, Motor, and CREST in memory nodes for the TPC-C [51], SmallBank [3], and YCSB [9] workloads (§8.2). For each workload (with multiple tables), we measure the space overhead based on the number of records, number of cells per record, and cell sizes in each table. We first examine the metadata overhead without cacheline padding (Table 1(a)). As all workloads have relatively large cells (see detailed schemas in §8.2), the overall metadata overhead of CREST remains moderate. CREST shows higher space overhead than FORD, especially for the TPC-C and YCSB workloads with a high number of cells per record, as CREST adds an epoch number and a commit timestamp to each cell. Motor has the highest space overhead due to extensive metadata for MVCC support. CREST trades reasonable space overhead for increased transaction processing throughput.

We also consider the scenario where cacheline padding

Table 2. RDMA operations issued by a transaction.

Systems	Execution	Validation	Commit
FORD / Motor	`READ` (read-only) `CAS+READ` (read-write)	`READ`	`WRITE+CAS`
CREST	`READ` (read-only) `masked-CAS+` `READ` (read-write)	`READ`	`WRITE+` `masked-CAS`

is enabled, where padding is added to align with 64-byte cachelines (Table 1(b)). All systems incur increased space overhead due to cacheline padding, and the overall space overhead of CREST still lies between FORD's and Motor's.

Space overhead in compute nodes. CREST caches each transaction's working-set records in the compute node's memory, which is freed when the reference counter reaches zero (i.e., no active transaction). Thus, records reside in the cache only during transaction execution, with memory usage proportional to the transaction's working-set size. Note that a transaction cannot be executed if its working set exceeds the memory capacity of a compute node. Nevertheless, for common OLTP workloads, where transactions are short-lived and access small data sets, the memory pressure is limited. Compared to FORD and Motor, CREST only adds a reference counter (2 bytes) and an epoch number (2 bytes). For example, under TPC-C (which contains larger records than SmallBank and YCSB), FORD, Motor, and CREST incur an average memory usage of 1.96 KiB, 3.99 KiB, and 2.48 KiB, respectively.

7.3 RDMA Communication Overhead

CREST introduces no additional RDMA communication overhead compared to record-level transaction systems. Table 2 compares the RDMA operations issued in each transaction for FORD, Motor, and CREST. CREST uses one `READ` for read-only data or one `masked-CAS` plus one `READ` for read-write data during execution, one `READ` during validation, and one `WRITE` plus one `masked-CAS` during commit. While CREST reads more data due cell-level concurrency control, the performance impact is minimal albeit the small size differences, as the number of RDMA operations (or round-trip times (RTTs)) is the major factor. Localized execution further reduces RTTs by skipping some `READ` operations. On the other hand, Motor can incur additional RTTs due to MVCC support, such as separately reading consecutive version tuples and values during execution.

8 Evaluation

8.1 Implementation

We implement the prototype of CREST in C++ from scratch, with 14 K LOC. It has the following key components.

Local version management. In each compute node, CREST uses multiple concurrent hash maps to index local objects. Each hash map is assigned to a database table and each record key is mapped to its corresponding local object. For local versions, CREST allocates memory from pre-registered memory region to ensure that a local version does not consume additional dynamic memory resources. All local versions are released once the record is committed to the memory pool.

RDMA request handling. CREST uses coroutines to improve CPU utilization [61, 68]. Each thread executes multiple coroutines, each acting as a coordinator. After issuing RDMA requests, a coordinator yields CPU control to the next coroutine. The waiting coroutine resumes execution upon completion of its RDMA requests. To mitigate CPU overhead of posting RDMA requests and polling for completion, CREST uses doorbell batching and selective signaling [20].

8.2 Methodology

Testbed. We conduct all experiments on a local cluster comprising five machines, three designated as compute nodes and two as memory nodes. Each machine is equipped with a 100 Gbps Mellanox ConnectX-5 Infiniband NIC, and all machines are interconnected via a 100 Gbps Mellanox Infiniband switch. Each compute node has an Intel(R) Xeon(R) Platinum 8260 CPU with 48 cores and 8 GiB of DRAM to host coordinators for transactions. Each memory node contains 192 GiB of DRAM to store database records and indexes. All machines run Ubuntu 20.04 LTS with Linux kernel v5.4.0.

Benchmarks and configurations. We evaluate CREST using the following transaction workloads:

- *TPC-C* [51]: It is a widely adopted benchmark for OLTP systems. By default, it includes 40 warehouses (the number is configurable), each including nine tables and five transaction types, with 92% being read-write transactions (e.g., creating orders or processing payments). Each record has an average of 6.6 cells, with an average cell size 36.1 bytes.
- *SmallBank* [3]: It simulates banking services, where a small fraction of accounts ("hot accounts") issue most transactions. We configure 100 K accounts, and use a Zipf distribution with a Zipfian constant $\theta = 0.99$ to model hot account activities. Each record has only one cell with an average cell size 26.7 bytes.
- *YCSB* [9]: It is a widely used benchmark for key-value stores. We adapt it for transaction processing. It creates a single table with 1 M records with four 40-byte cells each and supports read and write transactions. Each read or write transaction randomly selects N out of 1 M records (by default, $N = 4$), where read transactions access all cells of records, while write transactions randomly update one cell of a record. We generate hot records using a Zipf distribution with $\theta = 0.99$.

Setup. Before each experiment, we pre-load database records and indexes into memory nodes. Each compute node initializes coordinators and connects to the memory nodes for transaction execution. By default, each compute node runs

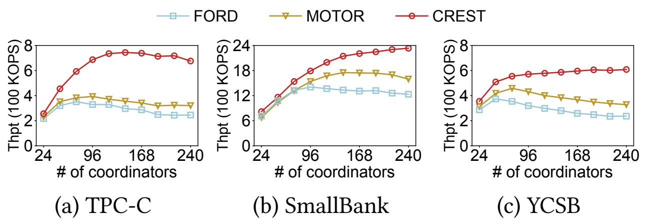


Figure 11. Exp#1: Throughput.

40 threads, each with two coroutines (i.e., 240 coordinators across three compute nodes in total). For each workload, we generate 10 M transactions per compute node.

We compare CREST with two state-of-the-art disaggregated transaction processing systems: FORD [68] and Motor [67], using their open-source prototypes for evaluation.

8.3 Overall Performance

We measure the throughput and latencies of CREST, FORD, and Motor by varying the number of coordinators. Each thread runs two coordinators, and we vary the number of threads per compute node from 1 to 40, (i.e., the total number of coordinators across three compute nodes increases from 24 to 240). We plot aggregated throughput (summed across compute nodes), average latencies, and tail latencies.

Exp#1 (Throughput). Figure 11 shows the throughput of CREST, FORD, and Motor. CREST consistently achieves the highest throughput under all workloads. At 240 coordinators, CREST achieves a throughput gain of 1.92×, 1.46×, and 1.85× over Motor, and 2.76×, 1.89×, and 2.58× over FORD, for TPC-C, SmallBank, and YCSB, respectively.

CREST also shows better scalability. For TPC-C, FORD and Motor reach the peak throughput at 72 and 96 coordinators, respectively, while CREST's throughput increases until 144 coordinators and reaches 743.7 KOPS, 72.4% higher than Motor's and 112.6% higher than FORD's. The reason is that CREST allows multiple coordinators to operate on different fields of the same record simultaneously. Similar trends are observed in other workloads.

Exp#2 (Average and median latencies). Figure 12 shows the average and median latencies of all systems. The latencies increase with the number of coordinators due to more severe contention and blocking. CREST consistently achieves the lowest average latency across a different number of coordinators. At 240 coordinators, CREST reduces the average latency by 41.1-62.6% compared to FORD and 17.7-44.4% compared to Motor. The latency reduction is attributed to CREST's localized execution, which reduces blocking time for conflicting transactions on the same compute node.

Similar to the average latencies, the median latencies increase with more coordinators. Under TPC-C, CREST achieves the lowest median latency with up to 192 coordinators. At 216 and 240 coordinators, Motor has a lower median latency, yet it also has a higher abort rate and consequently more transaction retries, which lead to a higher tail latency (Exp#3) and, overall, a higher average latency. Under SmallBank, CREST and Motor have comparable median latencies, both lower than FORD's. CREST does not reduce the median latency over Motor for SmallBank because more than half of the transactions are uncontended. Under YCSB, at 240 coordinators, CREST reduces the median latency by 45.8% and 47.2% compared to FORD and Motor, respectively.

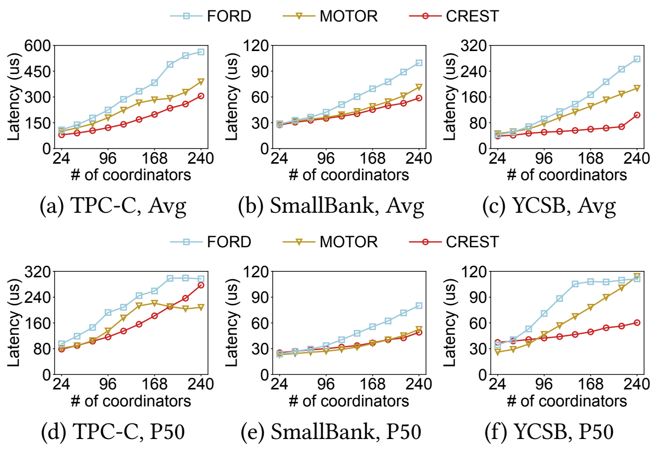


Figure 12. Exp#2: Average (figures (a)-(c)) and median (figures (d)-(f)) latencies.

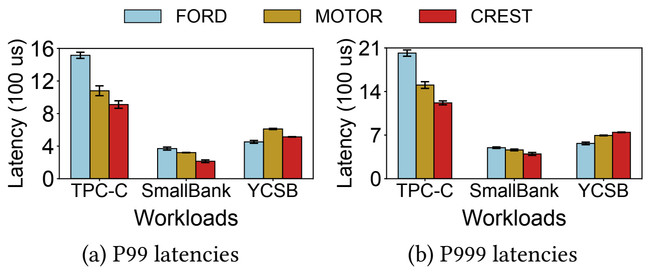


Figure 13. Exp#3: Tail latencies under different workloads.

Exp#3 (Tail latency). Figure 13 shows the 99th (P99) and 99.9th (P999) percentile latencies with 240 coordinators. CREST achieves the lowest tail latencies under TPC-C and SmallBank. Under TPC-C, the P99 latency of CREST is 15.2% lower than Motor's and 33.7% lower than FORD's. Under SmallBank, it is 36.7% lower than Motor's and 42.1% lower than FORD's. The reduction is due to CREST's reduced contention, which avoids unnecessary blocking and transaction aborts. Under YCSB, CREST has similar P99 latencies to Motor and FORD as high contention persists, in which coordinators across different compute nodes may still block each other, leading in high tail latencies.

8.4 System Analysis

We analyze the performance gains of CREST by considering both high and low skewness patterns. For TPC-C, we set the number of warehouses as 40 for high skewness and 100 for low skewness. For Smallbank and YCSB, we vary the Zipfian

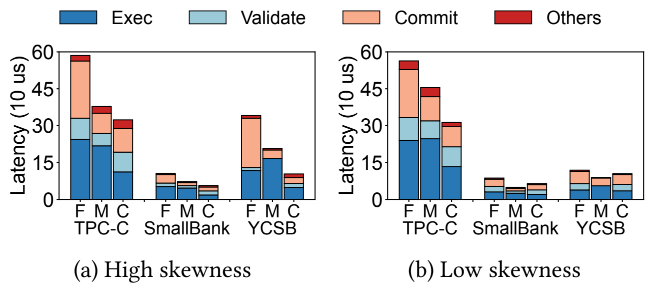


Figure 14. Exp#4: Average latency breakdown for CREST (C), FORD (F), and Motor (M).

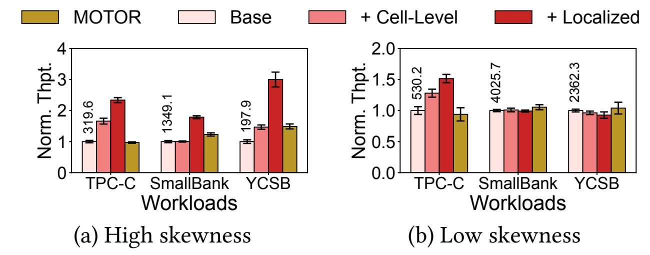


Figure 15. Exp#5: Factor analysis. We normalize results to Base, and the number above each bar represents the absolute throughput of Base in KOPS.

constant θ as 0.99 for high skewness and θ as 0.1 for low skewness.

Exp#4 (Average latency breakdown). Figure 14 shows the average latency breakdown for CREST, FORD, and Motor. CREST's latency reduction mainly comes from localized execution during the execution phase. Under $\theta = 0.99$, CREST reduces the execution latency by 54.3%, 65.9%, and 58.4% compared to FORD, and by 48.7%, 61.3%, and 70.7% compared to Motor, for TPC-C, SmallBank, and YCSB, respectively. Under $\theta = 0.1$, CREST reduces the execution latency by 44.6%, 18.3%, and 27.7% compared to FORD, and by 46.1%, 32.1%, and 15.4% compared to Motor, under TPC-C, SmallBank, and YCSB, respectively.

Exp#5 (Factor analysis). We examine the contribution of CREST's proposed techniques to performance gains. We start with a baseline system without the proposed techniques, and incrementally add cell-level concurrency control and localized execution (with parallel commits); note that for transaction correctness, both local execution and parallel commits must be performed together and cannot be isolated. Here, we omit FORD, as Motor consistently outperforms FORD.

Figure 15 shows the normalized throughput results with respect to the baseline system. Under high skewness, adding cell-level concurrency control increases throughput by 65.9% for TPC-C and 46.6% for YCSB due to improved concurrency. However, we do not observe improvements for SmallBank, as all its transactions access the same table column. Adding localized execution (with parallel commits) further increases

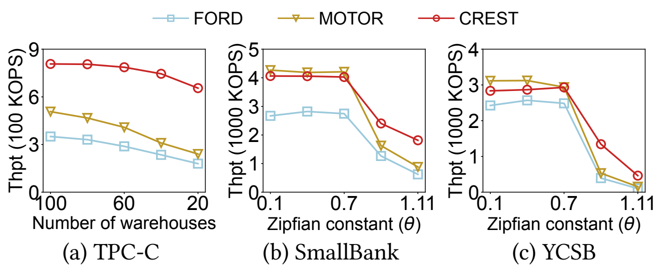


Figure 16. Exp#6: Throughput versus skewness.

throughput by 48.9%, 78.1%, and 104.6% under TPC-C, SmallBank, and YCSB, respectively, due to direct local cache access and reduced blocking time. Under low skewness, adding localized execution incurs extra cache management overhead, reducing throughput by 3.6% compared to non-localized configuration under YCSB. We argue that the cost is acceptable, given the significant throughput gains in skewed conditions that are common in practice. Note that cell-level concurrency control still improves throughput over the baseline system for TPC-C under low skewness, as false conflicts still exist.

8.5 Workload Sensitivity

Exp#6 (Impact of skewness). We evaluate the performance of CREST, FORD, and Motor under different skewness settings using TPC-C, SmallBank, and YCSB. For TPC-C, we increase skewness by reducing the number of warehouses from 100 to 20. For SmallBank and YCSB, we increase the Zipfian constant (θ) from 0.1 to 1.11 to simulate workloads with higher contention.

Figure 16 shows the throughput results. As skewness increases, all systems exhibit throughput degradation. Under TPC-C, reducing warehouses from 100 to 20 (i.e., increased skewness) results in a throughput drop of 18.8% for CREST, which outperforms both FORD (48.7% drop) and Motor (52.6% drop). Under SmallBank and YCSB with $\theta = 1.11$, CREST's throughput drops by 55.3% and 83.6%, respectively, compared to uniform cases, due to the increased conflicts across compute nodes and more aborts and blocking. Note that FORD and Motor experience higher severe throughput drops, with 76.3% and 79.3% under SmallBank, and 95.8% and 95.1% under YCSB, respectively.

CREST consistently achieves the highest throughput under skewed distributions. At maximum skewness, CREST's throughput is 3.64×, 2.91×, and 4.54× over FORD's, and 2.72×, 2.05×, and 3.02× over Motor, for TPC-C, SmallBank, and YCSB, respectively. The throughput gain of CREST grows with skewness. For example, for TPC-C with 60 warehouses (moderate skewness), CREST achieves 172.4% and 92.6% higher throughput than FORD and Motor, while its throughput gain increases to 264.4% and 172.6% at 20 warehouses (high skewness), respectively. The reason is that CREST's localized execution mitigates contention among transactions on the same compute node, especially under

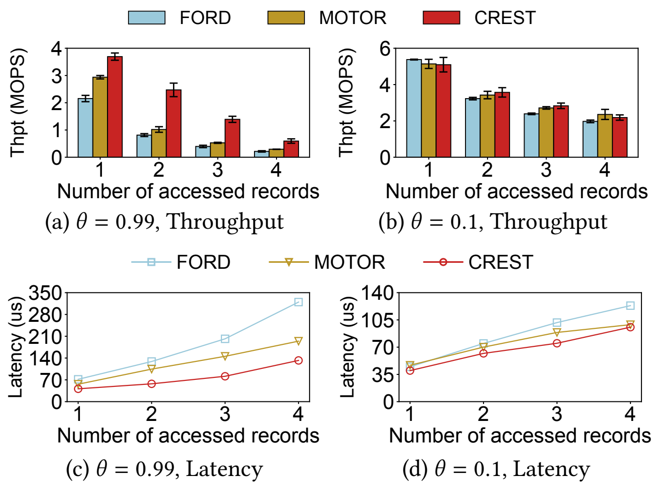


Figure 17. Exp#7: Throughput and average latency under YCSB when accessing a different number of records.

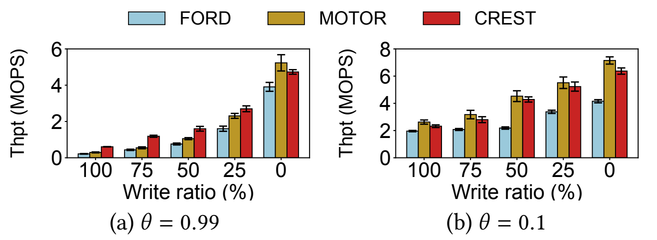


Figure 18. Exp#8: Throughput under YCSB versus write ratio.

more skewed conditions.

Exp#7 (Impact of number of accessed records). We examine the impact of the number of accessed records (N). We focus on YCSB, and vary N from 1 to 4 under $\theta = 0.99$ (high skewness) and $\theta = 0.1$ (low skewness).

Figure 17 shows the throughput results. All systems have throughput drops as N increases due to increased transaction conflicts. Under $\theta = 0.99$, CREST achieves the highest throughput. At $N = 1$, CREST achieves 71.4% and 25.7% higher throughput than FORD and Motor, respectively; at $N = 4$, the throughput gain of CREST increases to 177.4% and 102.3%, respectively. A larger N amplifies conflicts, so the benefits of CREST's contention resolution also increase.

Under $\theta = 0.1$, all systems have throughput drops as N increases, but exhibit similar performance. With limited contention, the throughput depends on commit round-trips (three per transaction for all systems in YCSB). CREST matches state-of-the-art performance under less skewness.

Exp#8 (Impact of write ratio). We study the impact of write ratio. We focus on YCSB and vary the write ratio from 100% (write-only) to 0% (read-only) under $\theta = 0.99$ (high skewness) and $\theta = 0.1$ (low skewness).

Figure 18 shows the results. Under $\theta = 0.99$, CREST excels in write-intensive workloads (write ratio $\geq$ 50%), with 180.9% and 105.1% higher throughput than FORD and Motor, respectively, when the write ratio is 100%. As the write ratio decreases, CREST's gain diminishes; at a 25% write ratio, its throughput is 16.9% higher than Motor's. For the read-only case, Motor outperforms CREST by 12.8% for two reasons: (i) Motor eliminates validation for read-only transactions using MVCC [67]; and (ii) CREST incurs additional overhead for local cache management. Under $\theta = 0.1$, CREST shows less throughput than Motor, by up to 10.1%.

9 Related Work

Accelerating distributed transactions. Extensive research has focused on enhancing the performance of distributed transactions. Some studies propose application-level optimizations, such as partitioning transaction workloads [23, 31, 66] and adopting deterministic execution models [13, 34, 50] to reduce transaction contention. Some studies propose new concurrency control and commit protocols, such as transaction reordering [42], runtime pipelining [62], and batch commits [35]. Some studies exploit RDMA to reduce network latencies by leveraging RDMA's one-sided operations [8, 10, 11, 45] or specialized RPC frameworks [21, 61]. All the above approaches target monolithic architectures, while our work focuses on disaggregated memory architectures, which cannot directly apply optimizations for monolithic architectures. In disaggregated memory architectures, transaction executions on compute nodes are separated from the data stored on memory nodes, so all conflict resolutions involve RDMA communications between compute nodes and memory nodes, and require specialized mechanisms to mitigate RDMA communication overhead while providing transaction correctness guarantees (§3).

Memory disaggregation. Some studies enable transparent disaggregated memory via specialized hardware [4, 15, 19, 29], OS kernels [18, 44, 46], or language runtimes [7, 39, 55]. Others optimize specific applications for disaggregated memory, such as indexing structures [30, 38, 57, 70], key-value stores [26, 48, 52, 58], caching systems [47], transaction systems [32, 33, 67, 68], and databases [28].

Among the studies, transaction and database systems designed for disaggregated memory [28, 33, 67, 68] are the most relevant to our work. FORD [68] is the first transaction system designed for disaggregated memory. It batches lock and read operations to reduce the number of round-trips required for transaction processing. It also maintains local versions for validation during the commit phase, so as to avoid additional round-trips for re-reading data. However, this approach is limited to a single compute node. In the scenario with multiple compute nodes, remote validation remains necessary due to potential staleness of local versions, thereby negating reduction in round-trips and prolonging blocking time. In contrast, CREST uses localized execution to expose uncommitted results early to effectively reduce blocking time between conflicting transactions.

Motor [67] uses multi-versioning to mitigate read-write conflicts and logging overhead, so as to achieve higher throughput. However, both FORD and Motor execute concurrency control at the record level, leading to false conflicts under high-contention workloads. CREST addresses this issue via fine-grained concurrency control.

Scythe [33] uses a hybrid concurrency control protocol that delegates reads and writes of frequently accessed records to memory nodes. However, it assumes sufficient computational capability in memory nodes to manage concurrent transaction reads and writes. CREST follows the common assumption that memory nodes have limited compute power.

HDTX [32] improves transaction performance via a fast commit protocol, RDMA-enabled offloading, and decentralized priority-based locking. Like FORD and Motor, HDTX still uses record-level concurrency control and faces similar performance issues under high-contention workloads.

GaussDB [28] is a distributed database that disaggregates compute, memory, and storage. It leverages a page-level ownership model, in which compute nodes temporarily own pages and use record-level locks with two-phase locking to process transactions. While this shares similarities to CREST's localized execution, CREST differs in two aspects. First, CREST adopts cell-level concurrency control instead of page-level ownership, thereby allowing concurrent operations on different cells within the same record across compute nodes and avoiding false conflicts inherent in GaussDB's page-level approach. Second, CREST exposes uncommitted results early to reduce blocking time, while GaussDB adheres to traditional commit procedures, which write undo logs before exposing results, and prolongs blocking periods. Note that GaussDB is close-sourced and we cannot directly compare it with CREST in evaluation.

Memory disaggregation for transactional support is reportedly deployed in production, such as ByteDance's veDB [49], Alibaba's PolarDB-MP [63], and Huawei's GaussDB [28]. These production systems support one-sided RDMA operations for fast transaction processing in disaggregated memory architectures. However, they do not specifically address the high-contention transaction workloads, as explored in CREST.

10 Conclusion

We present CREST, a disaggregated transaction system designed to efficiently handle high-contention workloads. CREST adopts cell-level concurrency control to eliminate false aborts, and localized execution to reduce the blocking time of conflicting transactions. To ensure that modifications are correctly and efficiently applied to memory pool, CREST introduces parallel commits together with consistent crash recovery based on redo-logging. Experimental results show that CREST outperforms FORD and Motor in different workloads.

Acknowledgments

We thank our shepherd, Shuai Mu, and the anonymous reviewers for their comments. This work was supported in part by the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92270202) and Research Grants Council of Hong Kong (GRF 14214622). The corresponding author is Mi Zhang.

References

[1] Raja Appuswamy, Angelos C Anadiotis, Danica Porobic, Mustafa K Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. In *Proc. of VLDB Endowment.*

[2] Martin Boissier, Carsten Alexander Meyer, Timo Djürken, Jan Lindemann, Kathrin Mao, Pascal Reinhardt, Tim Specht, Tim Zimmermann, and Matthias Uflacker. 2016. Analyzing Data Relevance and Access Patterns of Live Production Database Systems. In *Proc. of ACM CIKM.*

[3] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.

[4] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proc. of ACM ASPLOS.*

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[6] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *Proc. of USENIX FAST.*

[7] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *Proc. of USENIX OSDI.*

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions using RDMA and HTM. In *Proc. of ACM EuroSys.*

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of ACM SoCC.*

[10] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proc. of USENIX NSDI.*

[11] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proc. of ACM SOSP.*

[12] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.

[13] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High Performance Transactions via Early Write Visibility. In *Proc. of VLDB Endowment.*

[14] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proc. of USENIX OSDI.*

[15] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proc. of USENIX ATC.*

[16] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts*

and Techniques. Elsevier.

[17] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. Hyrise: A Main Memory Hybrid Storage Engine. In *Proc. of VLDB Endowment.*

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proc. of USENIX NSDI.*

[19] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proc. of ACM ASPLOS.*

[20] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proc. of USENIX ATC.*

[21] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proc. of USENIX OSDI.*

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. In *Proc. of VLDB Endowment.*

[23] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: Locality-Aware Distributed Transactions. In *Proc. of ACM EuroSys.*

[24] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2017. Rocksteady: Fast Migration for Low-Latency In-Memory Storage. In *Proc. of ACM SOSP.*

[25] Hsiang-Tsung Kung and John T Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[26] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. In *Proc. of VLDB Endowment.*

[27] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. (2019).

[28] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. In *Proc. of VLDB Endowment.*

[29] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based Memory Pooling Systems for Cloud Platforms. In *Proc. of ACM ASPLOS.*

[30] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. A High-performance RDMA-oriented Learned Key-value Store for Disaggregated Memory Systems. *ACM Transaction on Storage (TOS)* 19, 4 (2023), 1–30.

[31] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proc. of ACM SIGMOD.*

[32] Haodi Lu, Haikun Liu, Yujian Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2025. Fast Distributed Transactions for RDMA-based Disaggregated Memory. In *Proc. of USENIX ATC.*

[33] Kai Lu, Siqi Zhao, Haikang Shan, Qiang Wei, Guokuan Li, Jiguang Wan, Ting Yao, Huatao Wu, and Daohui Wang. 2024. Scythe: A Low-latency RDMA-enabled Distributed Transaction System for Disaggregated Memory. *ACM Transactions on Architecture and Code Optimization (TACO)* (2024).

[34] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. In *Proc. of VLDB Endowment.*

[35] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-Based Commit and Replication in Distributed OLTP Databases. In *Proc. of VLDB Endowment.*

[36] Yi Lu, Xiangyao Yu, and Samuel Madden. 2018. Star: Scaling Transactions through Asymmetric Replication. (2018).

[37] Xuchuan Luo. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proc. of ACM SOSP.*

[38] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *Proc. of USENIX OSDI.*

[39] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D Bond, Stephen M Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: A Low-Pause, High-Throughput Evacuating Collector for Memory-Disaggregated Datacenters. In *Proc. of ACM PLDI.*

[40] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proc. of ACM ASPLOS.*

[41] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. In *Proc. of ACM SIGMOD.*

[42] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *Proc. of USENIX OSDI.*

[43] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads using Fast Dynamic Partitioning. In *Proc. of ACM SIGMOD.*

[44] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *Proc. of USENIX NSDI.*

[45] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proc. of ACM SIGMOD.*

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proc. of USENIX OSDI.*

[47] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proc. of ACM SOSP.*

[48] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. 2023. FUSEE: A fully Memory-Disaggregated Key-Value Store. In *Proc. of USENIX FAST.*

[49] Jason Sun, Haoxiang Ma, Li Zhang, Huicong Liu, Haiyang Shi, Shangyu Luo, Kai Wu, Kevin Bruhwiler, Cheng Zhu, Yuanyuan Nie, et al. 2023. Accelerating Cloud-Native Databases with Distributed PMem Stores. In *Proc. of IEEE ICDE.*

[50] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. of ACM SIGMOD.*

[51] TPC-C Benchmark. Accessed in 2025. https://www.tpc.org/tpcc/.

[52] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proc. of USENIX ATC.*

[53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proc. of ACM SOSP.*

[54] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proc. of ACM SIGMOD.*

[55] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *Proc. of USENIX OSDI*.

[56] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *Proc. of USENIX OSDI*.

[57] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proc. of ACM SIGMOD*.

[58] Ruihong Wang, Jianguo Wang, Prishita Kadam, M Tamer Özsu, and Walid G Aref. 2023. dLSM: An LSM-based Index for Memory Disaggregation. In *Proc. of IEEE ICDE*.

[59] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. In *Proc. of VLDB Endowment*.

[60] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *Proc. of USENIX OSDI*.

[61] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better!. In *Proc. of USENIX OSDI*.

[62] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-Performance ACID via Modular Concurrency Control. In *Proc. of USENIX OSDI*.

[63] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Proc. of ACM SIGMOD*.

[64] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. In *Proc. of VLDB Endowment*.

[65] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proc. of ACM SIGMOD*.

[66] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-Centric Transaction Execution and Data Partitioning for Modern Networks. In *Proc. of ACM SIGMOD*.

[67] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *Proc. of USENIX OSDI*.

[68] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2023. Localized Validation Accelerates Distributed Transactions on Disaggregated Persistent Memory. *ACM Transactions on Storage* 19, 3 (2023), 1–35.

[69] Xiaodong Zhang and Jing Zhou. 2022. High-Performance Transaction Processing for Web Applications Using Column-Level Locking. In *Proc. of Springer WISE*.

[70] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *Proc. of USENIX ATC*.

A Artifact Appendix

A.1 Abstract

The artifact contains the CREST prototype, runner scripts for reproducing the experiments, and implementations of related workloads. The CREST prototype is a disaggregated transaction system featuring cell-level concurrency control, localized execution, and parallel commits. It is intended to validate the results reported in the paper and support further research on disaggregated transaction systems.

A.2 Artifact check-list (meta-information)

- **Program:** C++ programs, python scripts, and shell scripts.
- **Compilation:** CMake (version >= 3.5), g++ (version >= 11), Make.
- **Data set:** TPC-C, SmallBank, and YCSB.
- **Run-time environment:** Linux (Ubuntu 20.04 LTS or newer is recommended)
- **Hardware:** See §A.3.2 for details.
- **Metrics:** Throughput (transactions per second) and latencies (average, P50, P99, and P999 latency in microseconds).
- **Output:** Text log files with processed statistical results.
- **Experiments:** Overall performance, average latency breakdown, factor analysis, workload sensitivity.
- **How much disk space required (approximately)?:** At least 10 GiB for storing log files.
- **How much time is needed to prepare workflow (approximately)?:** Less than 30 minutes for initial setup, including configuring the setting, compiling code.
- **How much time is needed to complete experiments (approximately)?:** Our scalability evaluation takes over 14 hours, while other experiments take around 6 hours in total.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0 license.
- **Archived (provide DOI)?:** It is archived on Zenodo with DOI: `10.5281/zenodo.18227294`. It can be accessed at https://zenodo.org/records/18227294.

A.3 Description

A.3.1 How to access.

The artifact is available on Github at https://github.com/adslabcuhk/crest.

The artifact contains a README file, source code of CREST, scripts, and workload implementations. We explain these contents below, while more detailed information to run the artifact is provided in the README file.

The `src` directory contains the source code of CREST. The implementation of localized execution is in the `transaction` sub-directory, while the implementation of cell-level concurrency control is in the `transaction` and db sub-directories.

The `scripts` directory contains runner scripts for running experiments and collecting the results.

The `benchmark` directory contains the code to run and manage benchmark workloads, including TPC-C, SmallBank, and YCSB. `BenchRunner.cc` is the entry point of clients and servers with proper parameters.

A.3.2 Hardware dependencies.

To run all experiments in our artifact, we recommend a cluster of at least five Linux servers: two servers are used as memory nodes, and the other three servers are used as compute nodes. Each memory node should have at least 128 GiB of DRAM, and each compute node should have at least 16 GiB of DRAM. All servers should use Mellanox ConnectX-5 RNICs and be connected via an Infiniband switch.

A.3.3 Software dependencies.

The software dependencies include (see installation instructions in the README file):

- *RDMA driver* version `MLNX_OFED_LINUX-4.9-7.1.0.0` to enable using of RDMA experimental verbs.
- *C++ third-party dependencies* including TBB, Memcached, gflags, boost, and abseil.
- *python and python libraries* including python 3.8+, numpy, scipy, matplotlib, scp, and paramiko.

A.3.4 Data sets.

We provide the implementations of TPC-C, SmallBank, and YCSB workloads in the `benchmark` directory. The data sets are generated at runtime based on the configuration files in the `config` subdirectory.

A.4 Installation

The installation process includes the following steps. We use Ubuntu 20.04 LTS as an example, but the steps are similar for other Linux distributions.

1. **Install the RDMA driver.** Each machine needs to install the driver with version `MLNX_OFED_LINUX-4.9-7.1.0.0` for using RDMA experimental verbs. The driver can be downloaded here in the "Archived Versions". Extract the package and run the installation script `mlnxofedinstall` with root privilege.
```
tar -xzvf MLNX_OFED_LINUX-4.9*.tgz
cd MLNX_OFED_LINUX-4.9-7.1.0.0-ubuntu20.04-x86_64
sudo ./mlnxofedinstall
```
2. **Install build tools.** CREST requires gcc and g++ (version >= 11). Install the compilers. In Ubuntu, the compilers may be added via ppa.
```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-11 g++-11
# Use gcc-11 and g++-11 by default
sudo update-alternatives -install /usr/bin/gcc gcc
/usr/bin/gcc-11 110
sudo update-alternatives -install /usr/bin/g++ g++
/usr/bin/g++-11 110
```
We use `cmake` as our building tool. Install `cmake` using the following commands:
```
sudo apt-get install -y cmake
```
3. **Install C++ third-party dependencies.** The followings are the instructions to install the dependencies.
```
sudo apt-get install -y libgflags-dev libtbb-dev
libmemcached-dev memcached libmemcached-tools
```

```
wget https://archives.boost.io/release/1.83.0/
source/boost_1_83_0.tar.gz
tar -zxf boost_1_83_0.tar.gz
cd boost_1_83_0
./bootstrap.sh -prefix=path/to/installation/prefix
sudo ./b2 install
cd ../
git clone git@github.com:abseil/abseil-cpp.git
cd abseil-cpp
cmake -B build
cd build
sudo make install
cd ../../
```

4. **Instal Python script dependencies.** Before running these Python scripts, run the following command to install the dependencies:
```
cd scripts
pip install -r requirements
```
5. **Build CREST.** After installing the above dependencies, build CREST using the following commands:
```
cd CREST
make release
```
After successfully building the entire codebase, the binary file `build/benchmark/bench_runner` is created and will be used for benchmarking.

A.5 Experiment workflow

A.5.1 Setting up the configuration file. In CREST, each configuration file (`xxx.json` under `config` directory) contains all necessary information of this workload. It contains three parts: (i) `mn`, the configuration of each server comprising memory pool; (ii) `cn`, the configuration of each server comprising compute pool; and (iii) workload-specific configurations related to a specific workload. For each node, configure the following fields:

- The unique ID and IP address of this node
- devname, ibport and gid that are used to initialize the RDMA device
- mr_size is the size of memory region used for storing data. Create a large memory region for memory node and a small memory region for compute node.

A.5.2 Running nodes. CREST can manually start memory nodes and compute nodes using the binary file `bench_runner` generated from compilation.

```
# Memory node startup:
./bench_runner --type=mn --id=<id>
--config=<path_to_config> --workload=<workload>
--threads=<threads> --coro=<coros>
# Compute node startup:
./bench_runner --type=cn --id=<id>
--config=<path_to_config> --workload=<workload>
--threads=<threads> --coro=<coros>
--txn_num=<txn_num> --output=<path_to_output>
```

The `bench_runner` file takes multiple parameters, which need to be carefully set:

- `type`: set it to be `mn` or `cn`.
- `id`: the unique identifier of the node. When starting up the node, make sure to use the correct ID for this node.
- `config`: set it to be the path to the configuration file.
- `workload`: the workload to execute. It needs to be consistent with the `config` path.

After successfully booting the memory node, the output would be "`MNx waits for incoming connection`". You can use the compute node to run the workloads.

A.5.3 Data collection. After running a workload, each compute node will generate log files and results in the folder `bench_result`. We also provide scripts for reproducing the results presented in the paper. The scripts will modify the configuration files, run nodes, and merge result files.

- Exp#1 - Exp#3:
```
cd scripts
python3 run_scalability_bench.py crest tpcc
python3 run_scalability_bench.py crest smallbank
python3 run_scalability_bench.py crest ycsb
```
- Exp#4 - Exp#5:
```
cd scripts
python3 run_breakdown.py crest tpcc
python3 run_breakdown.py crest smallbank
python3 run_breakdown.py crest ycsb
```
- Exp#6:
```
cd scripts
python3 run_contentionlevel_bench.py crest ycsb
```
- Exp#7:
```
cd scripts
python3 run_sensitivity.py crest ycsb
```
- Exp#8:
```
cd scripts
python3 run_write_ratio.py crest ycsb
```

A.6 Evaluation and expected results

To reproduce the results presented in the paper, follow the instructions in the README file and §A.5.

Overall performance. This produces results in Exp#1 - Exp#3, which show CREST's overall throughput, average latency, P50 latency, and tail latency (P99 and P999) under different workloads and scales.

Average latency breakdown. This produces results in Exp#4, which shows the latency reduction from localized execution during the execution phase.

Factor analysis. This produces results in Exp#5, which shows the contribution of CREST's proposed techniques to performance gains.

Workload sensitivity. This produces results in Exp#6 - Exp#8, which show CREST's throughput under varying conditions of data skewness, a varying number of accessed records, and a varying write ratio.