OOCC: One-round Optimistic Concurrency Control for Read-Only Disaggregated Transactions

Hao Wu, Mingxing Zhang, Kang Chen, Xia Liao, Yingdi Shan, Yongwei Wu
Tsinghua University,Beijing China
hao-wu19@mails.tsinghua.edu.cn,zhang_mingxing@mail.tsinghua.edu.cn,chenkang@tsinghua.edu.cn
liaoxia5018@163.com,yingdi.shan@gmail.com,wuyw@tsinghua.edu.cn

***Abstract*—Read-only transactions predominate in many critical real-world scenarios. Yet, the presence of even a small proportion of read-write transactions poses challenges for existing Two-Phase Locking (2PL) and Optimistic Concurrency Control (OCC) based disaggregated transaction solutions. These approaches require either atomic operations or double reads to maintain consistent data for serializability, leading to suboptimal performance.**

This paper introduces OOCC, a novel One-round Optimistic Concurrency Control method tailored for disaggregated transactions. We propose that by intentionally postponing updates in write transactions for a moderate duration (a lease), it's possible to skip the validation phase in most OCC cases. This method enables read-only transactions to be completed within a single Round Trip Time (RTT) without involving any atomic operations. Additionally, we introduce several enhancements to boost OOCC's effectiveness in high-contention and write-intensive scenarios by reducing lock durations to just 1 RTT.

Our experimental results demonstrate that OOCC significantly boosts transaction throughput in read-heavy environments, showing improvements ranging from 1.2 to 4 times. OOCC consistently achieves the lowest average latency (40%-45% lower than the best counterpart) in both read- and write-heavy workloads.

***Index Terms*—disaggregated system, distribute transaction**

I. Introduction

A. Motivation

The rapid evolution of disaggregated architectures, powered by the efficiency of one-sided Remote Direct Memory Access (RDMA) operations, has attracted many research interests in the development of disaggregated transaction processing systems [1]–[3]. However, despite the advancements in this domain, the performance of distributed transactions, particularly those that are read-only, remains suboptimal. This fundamental issue stems from the necessity to maintain a consistent view of the data to ensure serializability.

Specifically, Figure 1 demonstrate the basic procedure of Two-Phase Locking (2PL) [4] and Optimistic Concurrency Control (OCC) [5], two fundamental frameworks for managing concurrency in transaction processing. As we can see from the figure and the comparison in Table I, OCC-based read-only transactions require a minimum of two communication rounds: one for reading and another for validation. This process, known as "double read", is even more taxing in disaggregated systems than in single-node systems. In a single-node environment, the latency for the second communication round is often mitigated by cache hierarchies. If there are no other threads concurrently updating the same data (i.e., no data conflict), the read data is likely residing in nearby caches and hence allowing for swift validation. Conversely, in a disaggregated environment, each round of RDMA-based communication incurs a significant cost.

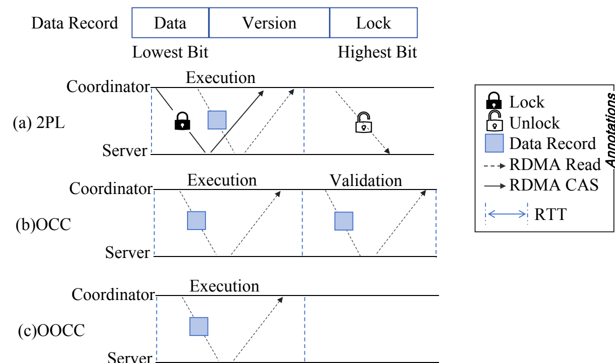


Fig. 1: Different protocols of handling read operation. The data/version/lock fields of each data record are co-located so that they can be read in a single RDMA operation. The locking and data fetching phase in 2PL can be overlapped in 1 RTT.

	OCC	2PL	OOCC (ours)
Round Trip	2	1	1
Primitive	RDMA Read only	Require RDMA CAS	RDMA Read only
Operations	2 Reads	2 CAS + 1 Read	1 Read

TABLE I: Comparisons of the costs of each protocol.

Alternatively, 2PL can process read-only transactions in just one round if the network layer ensures a specific message order (RDMA standard specifies that RDMA Read and CAS are totally ordered with all prior operations at the responder) [6]. The release-locking operations can be done in the background which only impacts the throughput. But this method requires the acquiring of read locks for each data partition, which involve heavy operations that involve remote atomic primitives. This requirement often cancels out the time saved from fewer round trips, especially when the conflict ratio is not high.

As demonstrated in Figure 2, the maximum Operations Per Second (OPS) for remote Compare-And-Swap (CAS) operations executed via RDMA is considerably lower than a standard RDMA read, attributable to additional synchroniza-

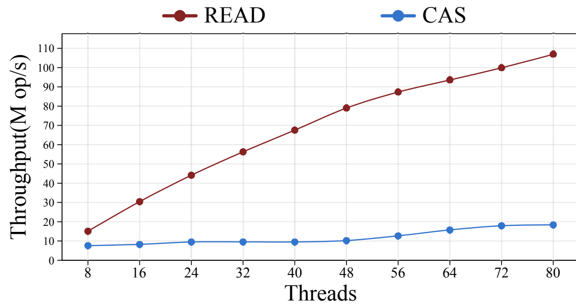


Fig. 2: OPS of different ops with different threads.

tion costs needed for atomicity. This observation aligns with other research in the field [7].

It is easy to imagine that, in scenarios involving only read-only transactions, a single RDMA read round trip should be sufficient. However, even in workloads dominated by reads, where only a few transactions involve updates, read-only transaction performance is significantly lowered. This is due to either the OCC's double read requirement or the overhead from 2PL's atomic operations. Yet, optimizing read-only transactions is essential as they make up a large portion of real-world workloads. For instance, Facebook's TAO reports 99.8% read operations [8], and Google's F1 on Spanner has three orders of magnitude more reads than write transactions [9].

Therefore, we believe improving read-only transaction performance is crucial in many important scenarios, even at the cost of a moderate additional latency for read-write transactions, to enhance overall system responsiveness.

B. Our Contributions

To address the above challenge, our work presents One-round Optimistic Concurrency Control (OOCC), a novel approach that is a combination of OCC and lease. OOCC leverages independent leases to establish a global lease period. The records remain unchanged during the lease, so read-only transactions can be quickly committed in the first communication round if they are within the lease. If they are, they can skip the second validation round, hence speeding up the transaction process. If not, the process reverts to standard OCC, avoiding any extra overhead in worst-case scenarios.

While our strategy prioritizes read operations, it does not unduly penalize read-write transactions. The additional latency delay for write transactions is consistent, resembling a lease duration. Additionally, by overlapping the lease period with the inherent latency in logging operations, we can further reduce the perceived latency increases. Thereby our strategy maintains a balanced approach that favors read-heavy workloads without severely impacting writes.

Moreover, similar to OCC, high contention scenarios present a unique set of challenges to our system. Even in read-dominate workloads, it is possible that only a small number of read-write transactions can lead to high contentions. Our analysis of OCC indicates that increasing skewness to 0.99 results in a significant decrease in throughput, primarily due to increased transaction failures and retries. To strengthen our system in such environments, we have introduced two additional features named "delayed write lock" (§IV) and "write unlock" (§V). These innovations are designed to shorten the lock duration for read-write transactions, thereby reducing the transaction aborts and retries. Moreover, our system also integrates a Two-Phase Commit(2PC) like protocol, which correctly coordinates the lease across coordinators (§VII).

To prove the correctness of OOCC, we present a detailed proof in §VI demonstrating that OOCC guarantees strict serializability. Additionally, a TLA+-based automatic proof of OOCC's correctness is included in the supplementary material.

To demonstrate the efficacy of OOCC, we conducted comparative analyses with both OCC-based and 2PL-based state-of-the-art solutions [1], [10], [11]. The experimental outcomes reveal that in read-dominant settings, for which OOCC is tailored, our system enhances transaction throughput by 1.2×-4× and simultaneously decreases the average latency by 40%-45% lower than the best counterpart (§VIII-C). In write-heavy scenarios, OOCC's throughput is only about 30% lower than DSLR [11], a system specifically optimized for write-intensive workloads (§VIII-E). Conversely, OOCC significantly outperforms DSLR in read-heavy contexts and achieves minimal latency in both read- and write-heavy workloads.

II. Background

A. Disaggregated Transaction

Traditionally, computing systems have relied on a tightly integrated architecture, where memory is closely coupled with processing units within a single server. Disaggregated systems redefine this model by separating memory resources from individual servers, consolidating them into a shared pool linked to compute nodes via a high-speed network. This design enhances resource utilization, scalability, and failure isolation by allowing each compute and memory pool to be deployed and scaled independently. A pivotal focus in this field is the application of Remote Direct Memory Access (RDMA) to implement advanced concurrency control schemes, notably two-phase locking (2PL) [4] and optimistic concurrency control (OCC) [5].

2PL based methods. For instance, an exemplary implementation in this context is DrTM [10], which utilizes RDMA to apply a 2PL scheme. DrTM's novel lease-based shared lock mechanism overcomes the drawbacks of releasing the locks. It attaches a timestamp along with the record as the valid time of the lease. When the coordinator visits the record, it will first extend this timestamp using RDMA CAS operation. This approach requires two or more operations to read a record and the length of the lease is hard to tune. A small lease will cause frequent extensions of the lease which will further degrade the performance, while a large lease will block the read-write transactions. In a different approach, DSLR [11] employs RDMA's Fetch-And-Add (FAA) operation to create a ticket lock system. Unlike the potentially failing Compare-And-Swap (CAS) operations, the FAA operation is guaranteed to succeed, thus enabling DSLR to avoid starvation and

unnecessary retries. This mechanism ensures first-come-first-serve scheduling, promoting fairness in transaction processing and benefits the conflict transactions.

Despite these advancements, it's noteworthy that 2PL-based concurrency control systems rely heavily on atomic RDMA operations, even in read-only transactions. This reliance can significantly impede performance, making these systems less suitable for read-dominant scenarios.

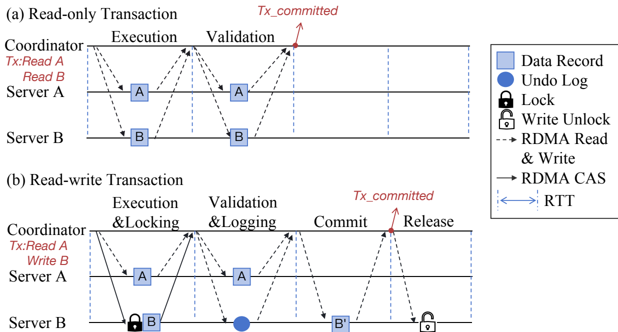


Fig. 3: State of art solution of OCC. Read-only transaction only needs two phases. The read-write transaction needs 4 phases in which the Execution Phase and the Locking Phase overlap, and so does the Validation Phase and Logging Phase.

OCC based methods. Thus, in our paper, we focus on optimizing solutions based on OCC. Figure 3 presents a standard framework used by existing RDMA systems for processing distributed transactions via OCC. The protocol only encompasses two sequential phases for the read-only transactions: Execution and Validation. The read-only transactions will be committed if the results of the Validation phase match the results of the Execution phase. Otherwise, they will be aborted.

In contrast, the OCC protocol encompasses four sequential phases for the read-write transactions: Execution and (overlapped) Locking, Validation and (overlapped) Logging, Commit, and Release. During the Execution phase, the coordinator reads records from the read set and locks the write set. In the Validation and Logging phase, the coordinator's tasks include: *1)* verifying the integrity of the read set records (ensuring they have not changed); *2)* concurrently sending the undo log to the server; and *3)* waiting for the completion of all these operations, ideally within a single RTT. The transaction then moves to the Commit phase, where records are updated. Finally, during the Release phase, the records are unlocked, which prevents partial reads and allows these modifications to become visible to other transactions. To avoid the issue of partial reads, existing works typically update all the records before releasing any locks. Consequently, the Commit and Release phases cannot be overlapped.

B. Read-only Transaction

Due to the importance of read-only transactions, there are also many specialized optimizations in traditional non-disaggregated environments. A notable concept is the NOCS Theorem [12], which highlights a fundamental challenge: it is not possible to simultaneously achieve non-blocking read-only transactions (N), one-round communication (O), constant metadata (C), and strict serializability (S). Solutions such as SNOW-optimal, PORT, and DST have managed to attain three attributes (N+O+C) at the cost of strict serializability (S).

Commonly, these optimizations utilize timestamps to convert read-only transactions into snapshot reads, thereby ensuring the transactions are non-blocking. This process typically includes the coordinator checking and atomically updating the timestamp of a tuple when the transaction's commit timestamp is larger. Such a step is essential to maintain serializability. However, applying one-sided RDMA operations for these tasks in disaggregated memory systems can lead to increased round trips and dependence on expensive atomic operations, ultimately impairing performance.

III. Global Lease Period

For a better understanding, we initially assume uniform clock rates across all replicas, thereby sidestepping the issue of clock drift. We will delve into strategies for mitigating clock drift in Section VII-B, but it does not significantly alter the primary concepts outlined in our subsequent discussion.

As described above, the primary objective of our research is to obviate the need for a validation phase in OCC. To achieve this, we must ensure that all data records accessed during the execution phase remain unaltered by concurrent updates. The cornerstone of our approach is the use of leases to establish a global lease period. In OOCC, leases are commitments made by updaters. They ensure that the releasing of locks for updates is temporarily deferred for a **specific, limited, and predetermined** period. Consequently, if all the data records accessed during a read-only transaction fall within the lease period, their consistency can be directly inferred from the initial states, eliminating the need for subsequent verification. It significantly reduces the likelihood of needing double reads, especially in the context of short read-only transactions.

A. Transactional Write

The process for handling writes in OOCC adheres to the principles of OCC, but it incorporates an additional constraint that writes must secure the safety of the lease. Furthermore, Sections 4 and 5 outline optimizations to the write procedure aimed at reducing write contention and enhancing performance. However, we will first discuss the fundamental procedure. When a coordinator successfully locks a tuple, it operates under the assumption that an implicit read lease has been set on the record, even in the absence of an actual concurrent read transaction. The coordinator waits until this implicit lease expires before releasing the locks.

As depicted in Figure 4, read-write transactions in OOCC still undergo four phases. In the initial Execution&Locking phase, the coordinator sets the lock using RDMA CAS, and later, in the Releasing phase, it releases the lock with RDMA Write. The primary distinction between OOCC and traditional OCC lies in the lock release timing. In OOCC, the lock is only released after the lease has expired. This ensures that the lock

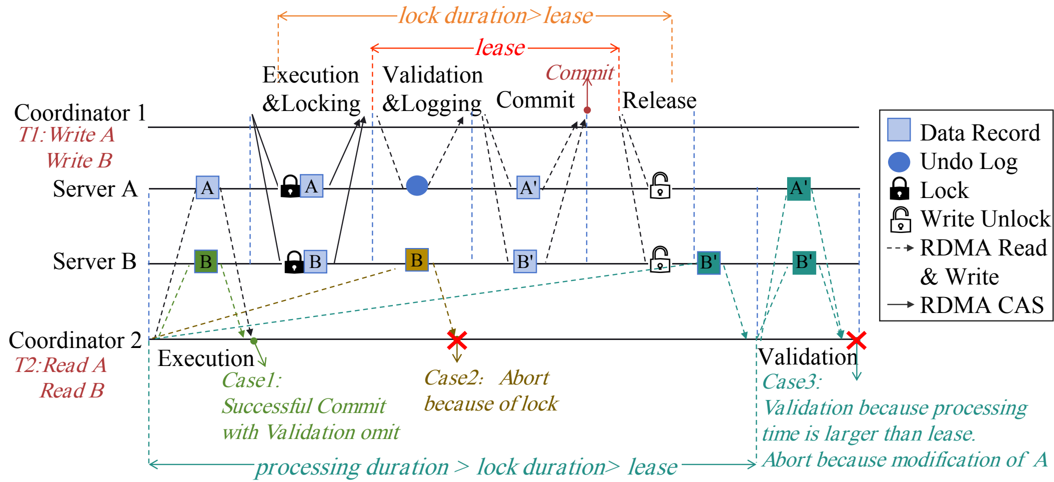


Fig. 4: The process of OOCC handling transactions. The coordinator1 handles a read-write transaction while the coordinator2 handles a conflicting read-only transaction. The three cases are the possible situations that happened in OOCC.

duration always exceeds the lease period. This characteristic is crucial for verifying the atomicity of a read-only transaction.

B. Transactional Read

With the help of the aforementioned leases, OOCC enables read-only transactions to be executed efficiently, often requiring just a single round of RDMA reads. In OOCC, data and its associated lock are co-located, allowing them to be accessed simultaneously through a single RDMA Read operation during the Execution phase of read-only transactions. When performing read operations, the coordinator retrieves the record along with its lock status. The data is deemed valid if the corresponding lock field indicates that the record is not locked by other concurrent transactions.

Importantly, this read operation serves a dual purpose. Apart from fetching the record, it also implicitly establishes a read lease. This implies that if all the records read are unlocked and the read duration does not surpass the lease period, there is a certainty that no concurrent read-write transaction will intervene, thus preventing inconsistent states. This guarantee is based on the protocol that even if a concurrent writer locks a record immediately after the read, this writer will intentionally delay releasing its locks for at least a lease period.

To be more explicit, we illustrate all three possible scenarios of executing read-only transactions in OOCC with Figure 4. In this example, Coordinator 2 is attempting to commit a read-only transaction involving two shards, while Coordinator 1 concurrently executes a read-write transaction. In Case 1, all read operations by Coordinator 2 occur before Coordinator 1's locking, resulting in both returning an unlocked status and within the lease period. This indicates that Coordinator 2 has secured implicit read leases, valid during its Execution phase, ensuring that no data changes occurred. Consequently, the Validation phase can be omitted. OOCC thus ensures that most read-only transactions can complete in a single RTT in conflict-free scenarios. In Case 2, however, the transaction will abort as the read operation for record B encounters a lock.

More importantly, Case 3 presents a more complex scenario where a read-only transaction T2 reads two records updated by transaction T1 but only encounters one modified record B. This partial read scenario cannot be identified merely from the lock status as Coordinator 1 has already released the lock. However, the processing duration of T2 exceeds T1's lock duration, which is guaranteed to surpass the lease. Thus, Coordinator 2 can identify a potential serializability violation by differentiating Case 3 from Case 1. In such instances, Coordinator 2 reverts to the standard OCC procedure, executing an additional validation phase to ensure atomicity.

In summary, the implicit lease-setting method employed in OOCC simplifies the read process by eliminating the need for additional validation. However, this approach may potentially increase latency for write operations. Each write must wait until this lease expires to maintain consistency. Therefore, carefully determining the lease duration is crucial. A lease duration that is too long could unnecessarily extend the latency for read-write transactions, whereas a very short lease might result in unnecessary validations in read-only transactions. Further details and analysis are discussed in §VIII-F.

C. Unknown Read Set or Write Set

The design of OOCC targets non-interactive transactions, which is a common approach [3], [11]–[14]. There are also transactions in this category where the read and write sets are unknown. For these transactions, we utilize a method that is leveraged in OCC. The coordinator issues RDMA reads during the execution phase to obtain the necessary read and write sets. After that, we can continue executing the transaction based on the read data and validate this data before committing. Clearly, the execution phase of such transactions will take longer. Therefore, we need a longer lease to skip the validation phase.

The results in VIII-C show that OOCC still outperforms other systems by up to 1.5× in this kind of transaction.

IV. Reduce Lock Duration

Since OOCC is still based on OCC, it is also susceptible to high transaction abortion ratios in scenarios of intense conflict. To alleviate this issue, a key strategy is to minimize the lock duration, thereby reducing the potential for conflicts. Prior research has explored lock duration reduction through techniques like early lock release [15], [16], which allows a transaction to release its lock on a data record after its final update, permitting other transactions to access the data. Such methods have been shown to enhance performance by enabling transactions to read uncommitted data. However, they necessitate complex algorithms and data structures for tracking dependencies and managing cascading aborts, making them less suited for disaggregated systems. In our approach, we introduce a mechanism called **"delayed write lock"**.

Notably, in OCC, the standard approach is to maintain consistency by detecting potential conflicts during the Execution phase. If a transaction finds that a record is locked, the pragmatic approach is to abort the transaction which ensures strict serializability. However, for read-only transactions concurrently executed with read-write transactions, this decision can lead to numerous false aborts. This issue is especially pronounced in OOCC due to the intentional lease waiting.

For instance, the aborted Case 2 in Figure 4 exemplifies a false abort. Transaction T2 could have been committed instead of aborted since record B had not yet been modified by transaction T1. T2 could be scheduled before T1 without violating strict serializability. The issue is that the lock is held too long for other read-only transactions to read, leading to false aborts. We observe that strict serializability between read-write and read-only transactions can be maintained if the read-write transaction sets the locks before updating the record.

Therefore, to commit T1, it's essential to find a method to delay the write lock causing aborts **only** for concurrent read-only transactions, without hindering the lock for read-write transaction conflicts. OOCC addresses this by splitting the write lock into an intention lock and a write lock. During the execution phase, the coordinator sets the intention lock for the write set to block other read-write transactions, but read-only transactions can still read the records. However, the record is forced to undergo the validation phase as the coordinator does not secure the read lease for the record.

The introduction of intention locks adds an additional lock operation, leading to questions about efficiently sending lock requests to remote servers. A straightforward approach is to allocate a dedicated round trip for sending the lock request before the Commit phase, incurring an extra RTT. However, we note that lock requests can be initiated at any stage before the actual modifications. This realization has prompted the strategy of a parallel locking scheme, where lock requests are generated and dispatched simultaneously during the Commit phase. This strategy enables the combination of the lock and data update requests into a single RDMA Write operation,

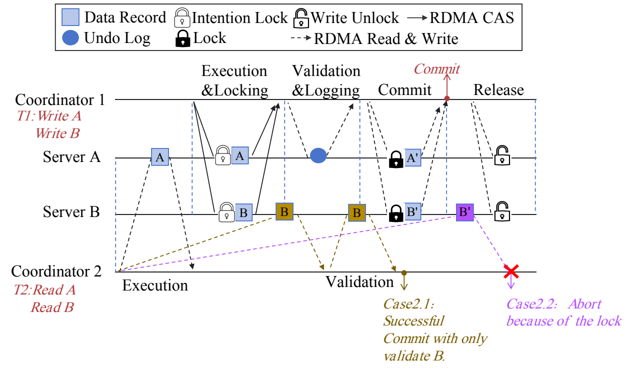


Fig. 5: Progress of intention lock.

thus obviating the need for an additional RTT. Illustrated in Figure 5, Coordinator 1 sets intention locks in the Execution & Locking phase and the write lock in the Commit phase.

This mechanism further refines the original aborted Case 2 in Figure 4 into two subcases in Figure 5. In case 2.1, Coordinator 2 does not abort the transaction but instead validates record B and commits since record B remains unchanged. Conversely, in Case 2.2, the transaction is aborted due to the presence of a write lock on record B. By employing delayed locking, the actual lock duration for read-only transactions is effectively reduced to a mere one RTT.

V. One-op Write Unlock Optimization

The above "delayed write lock" optimization we introduced effectively reduces the duration of write locks to a single RTT, which in turn decreases the likelihood of unnecessary aborts of read-only transactions. However, the read-write transactions still need four RTTs to commit. In this section, we aim to reduce the number of RTTs required for read-write transactions. While the introduced lease can reduce the additional validation phase's RTT, simply applying leases to read-write transactions does not decrease the required RTTs.

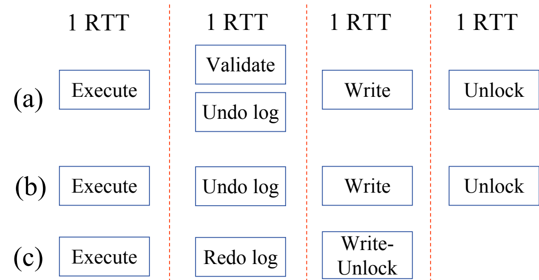


Fig. 6: The brief progress in which (a) is the origin transaction progress, (b) is the progress with lease shows that the RTTs can't be reduced since sending logs requires an RTT and (c) is the progress with lease and write-unlock.

As process (a) shows in Figure 6, existing approaches leverage the undo log to ensure the crash consistency because the undo log can be sent during the validation phase which saves 1 RTT. However, this overlapping can not fit with the

lease to reduce the transaction procedure. As process (b) shows, the transaction still needs 4 RTTs to commit even with the validation omitted as the undo log still needs 1 RTT to be stored. To further reduce the round trips of the transaction, we have proposed the *Write Unlock* to combine the data update operation and the lock releasing operation into a single operation. As shown in progress (c), the read-write transaction only needs 3 RTTs with lease and *Write-Unlock*.

The "write unlock" optimization leverages the fact that RDMA writes occur in increasing address order [17]–[19]. It is important to note that, in OOCC, the write lock is located at the highest address, typically as a footer. Consequently, an RDMA write that updates a value can simultaneously release the lock. OOCC employs this RDMA Write operation for in-place updates of the remote replica, with the entire write process being completed once the write lock is released. This effectively inlines the unlock operation into the data updates, hence updating the record and releasing the lock in a single operation and reducing the transaction process by 1 RTT.

While the *Write-Unlock* enhances performance, it also introduces a challenge in the event of a failure with undo logs. The issue with the undo log is that when a failure occurs, the transaction needs to roll back. This characteristic can lead to inconsistencies when combined with write unlock, as a record may be read by other transactions but then rolled back during recovery. To solve this, it is essential to ensure that once any updated data record is unlocked, the entire transaction has already been committed. In such situations, replaying the transaction is more appropriate rather than aborting it, to guarantee that any committed data, which might have been read by another transaction, is not lost. Thus, OOCC relies on redo logs instead of undo logs. In case of failure, the transaction will be replayed with the redo logs if the redo logs are successfully stored. Otherwise, the transaction can be simply aborted since the transaction has not been committed and all the data have not been modified.

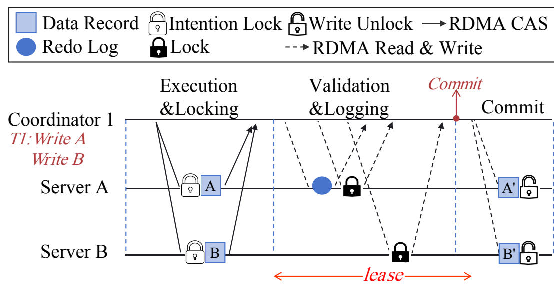


Fig. 7: The progress of OOCC after applying the "One-OP Write Unlock" optimization.

The modified process, after applying the "write unlock" optimization, is illustrated in Figure 7. The primary change is that both the logging and locking operations are executed within the same RTT. The operations for updating the record and releasing the lock are consolidated into a single RDMA Write in the Commit phase. Note that this operation needs to be operated after the lease expires to prevent partial reads. Consequently, the entire process comprises only three phases.

Algorithm 1: OOCC algorithm

```
1  Function Execute&Locking(T):
2      // read and set the implicit lease
3      issue RDMA read requests to get all the records in
         the read set and set the implicit lease
4      if any record is locked or (intention-locked and T
         is read-write) then
5          abort the transaction
6      // handle write set
7      if T is not read-only then
8          issue RDMA CAS requests to set the intention
             locks on the records in the write set
9          if any CAS request fails then
10             abort the transaction
11 Function Validate&Logging(T):
12     // check if the validation can be omit
13     if lease expires or any record is intention-locked
         then
14         issue the RDMA reads to validate the records
             with intention-lock or expired lease
15         if any record version does not match then
16             abort the transaction
17     // handle write set
18     if T is not read-only then
19         issue the RDMA writes to set the locks on the
             write records
20         issue RDMA writes to store the redo logs
21     Commit(T)
22 Function Commit(T):
23     if T is not read-only then
24         wait for the lease if it is not expired and the
             redo log acks
25         issue the RDMA write to update the records
             and release the locks
26     return to client
```

VI. Algorithm and Correctness

A. OOCC Algorithm

The algorithm of OOCC is shown in Algorithm1. The whole algorithm consists of multiple functions corresponding to the different phases of OOCC.

Execute&Locking(T). During this phase, the coordinator will issue the RDMA reads to get the records in the read set. The implicit lease is also set by this operation. If any record is locked by other coordinators, the transaction will be aborted. For the write set, the coordinator will issue the RDMA CAS requests to set the intention lock. If any CAS request fails which means the record is locked by other concurrent transactions, the coordinator simply aborts the transaction.

ValidateTxn&Logging(T). During this phase, the coordinator will validate the records in the read set. For each record

in the read set, if the lease is still valid and no intention-lock encounters, the validation can be omitted. Otherwise, the record needs to be validated through RDMA reads. If the validation fails due to a version has been changed, the coordinator aborts the transaction. Otherwise, the coordinator continues to handle the write set. For the write set, the coordinator will update the intention lock to the lock to prevent other readers. Since all the records in the write set have already been intention-locked, the coordinator can simply issue RDMA writes to upgrade the locks. Additionally, the redo-logs need to be stored using the RDMA writes. As we can see, if the validation is omitted, the read-only transactions can skip this phase and commit directly. The read-write transactions only need to update the locks and store the redo logs in parallel.

Commit(T). If the transaction is read-only, the coordinator directly returns to the client. Otherwise, the coordinator will first wait for the lease to expire to ensure the correctness of the lease and the redo log acks to ensure that the redo logs have already been stored. Then, the coordinator can update each record and release the lock using one RDMA write operation.

Figure 8 shows some examples of the OOCC algorithm. Transaction T1 is a read-write transaction that modifies records A and B and transaction T2 is a read-only transaction that reads records A and B. Coordinator1 sets the intention locks during the *Execution&Locking* phase. In the *Validation&Logging* phase, it sets the locks and stores the redo log without validation, then commits in the *Commit* phase after the lease expires. Transaction T2 may encounter several cases. In case 1, T2 commits with validation omitted since it gets A and B within the lease, and no intention locks are encountered. In case 2, T2 encounters the intention lock on record B, so it has to validate record B as in *Validate&Logging()* line 12. Coordinator 2 finds the record has not been modified during the validation and then commits the transaction. In case 3, T2 will abort due to the lock on record B.

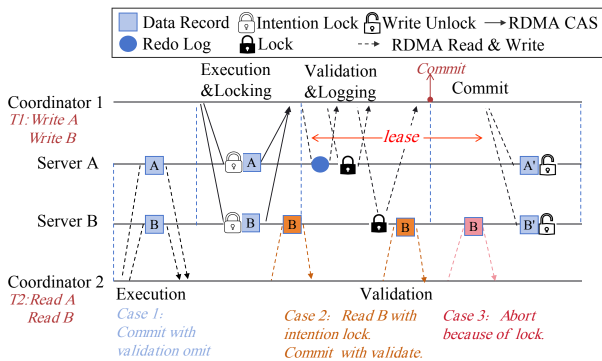


Fig. 8: The examples of OOCC algorithm.

B. Formal Proof

Theorem VI.1 (SERIALIZABILITY)**.** *OOCC implements serializable read-only transactions, which always read a consistent snapshot generated by concurrent serializable read-write transactions.*

PROOF SKETCH. In this section, we provide a proof sketch by contradiction, based on the intuition that if a read-only transaction cannot be serialized with read-write transactions, it leads to a contradiction. Before presenting the proof, we need to establish the following lemma:

Lemma VI.2. *Given a committed read-only transaction TX_{RO} and a committed read-write transaction TX_{RW}, if TX_{RO} observes any TX_{RW}'s update, then TX_{RO} can observe all the updates made by TX_{RW}.*

Proof. Assume TX_{RW} updates record A and B to $A^{'}$ and $B^{'}$. TX_{RO} has already read $A^{'}$. It will read B in either 1) before the Execution&Locking phase of TX_{RW}, or 2) during the Validation&Logging phase of TX_{RW}. In scenario 1), TX_{RW} will wait until the lease of record B expires before it releases the lock on record A. Since TX_{RO} has already read $A^{'}$, the execution time of TX_{RO} must have exceeded the lease. TX_{RO} will have to go through the Validation phase and abort due to lock or modification of B. In scenario 2), TX_{RO} will also go through the Validation phase since it does not get the lease of record B and abort due to the reasons above. Thus TX_{RO} can only commit with reading both $A^{'}$ and $B^{'}$. □

Proof Of The Theorem. TX_1 updates record A to $A^{'}$, TX_2 updates record B to $B^{'}$and TX_2 depends on TX_1. Assume a read-only transaction TX_{RO} only observes TX_2's update on record B but does not observe TX_1's update on record A(i.e., inconsistent reads). Since TX_2 depends on TX_1, we assume the TX_2 reads the record C updated to $C^{'}$ by TX_1 during the Execution&Locking phase. Then TX_{RO} will observe the $C^{'}$ otherwise it will fall to scenario 1 in **Lemma** VI.2 and abort if it reads record C. Therefore, TX_{RO} observes $C^{'}$ but does not observe $A^{'}$, which is contradictory to **Lemma** VI.2. □

Serializability proved above in OOCC is also strict: the serialization point is always between the start of execution and the completion being reported to the application. A TLA+-based automatic proof of OOCC is provided in the supplementary material.

C. Discussion of Generalizability

Using leases to accelerate read-only transactions is a general approach that can be applied to other non-RDMA environments. Our current design mainly leverages certain characteristics of RDMA's one-sided verbs, such as the optimal performance of one-sided reads, which is why we incorporated implicit leases into OCC. In non-RDMA environments, like RPC, explicit leases can be implemented and handled by the remote CPU. Additionally, other concurrency control protocols, such as 2PL, can be explored for further possibilities. Besides leases, other optimizations in the paper can also be applied in non-RDMA settings; for instance, the write unlock optimization can be more easily implemented in RPC.

VII. Dynamic Adjustment of the Lease

When the workload changes, the lease needs to be changed to suit the workload for better performance. In this section,

we will show how to dynamically change the lease during the transaction executions. The challenge is that, since each client uses the lease stored in the local machine, different leases may cause the transaction to read the partial updates, leading to an inconsistency. Even using the Two-Phase Commit (2PC) [20] to update the lease value across coordinators cannot resolve this issue, as there can always be scenarios where some coordinators receive updates sooner than others.

To address this problem, we separate the lease into two leases. The lease in OOCC serves two functions during the commit phase. Firstly, it is used to verify the validity of records in the read set, using the **read-validate lease**. Secondly, it is employed for coordinators to wait for the lease to expire before applying updates. We denote this lease as **write-wait lease**. Previously, we used the single term *lease* for both read and write operations. This is because, typically, "read-validate lease = write-wait lease = lease". We maintain the following property. **Property 1.** All coordinators hold the same leases when they are not in the lease-adjustment process. By differentiating these leases, we ascertain that the system remains secure as long as "read-validate lease $\leq$ write-wait lease". Specifically, to maintain safety, the following property must be upheld during lease adjustments: **Property 2.** max(read-validate lease of all coordinators) $\leq$ min(write-wait lease of all coordinators).

A. Lease Adjustment

To facilitate this, a new role known as the *Lease Manager* has been introduced to update each coordinator's lease in a 2PC-like manner. The lease adjustment process also comprises two stages: the *Prepare* phase and the *Commit* phase.

Prepare Phase. In this phase, the Lease Manager initiates the process by informing all coordinators about the new lease. Initially, all the read-validate lease and write-wait lease remain at $lease_{old}$. Upon receiving the prepare request, the coordinator transitions to an intermediate state where the read-validate lease is adjusted to $min(lease_{old},\ lease_{new})$, and the write-wait lease to $max(lease_{old},\ lease_{new})$. The corresponding possible states of leases are detailed in Table II. Even though in this phase the coordinator can be in one of two states, Property 2 is maintained in both states and hence ensures the system's safety.

	Before changing	Intermediate
read-validate lease	$lease_{old}$	$min(lease_{old}, lease_{new})$
write-wait lease	$lease_{old}$	$max(lease_{old}, lease_{new})$

TABLE II: The leases during the Prepare phase.

Commit Phase. The Commit phase begins once the Lease Manager has notified all coordinators of the new lease and they have moved to the intermediate state. The Lease Manager then sends a commit request to all coordinators. Upon receiving this request, a coordinator updates both the read-validate lease and the write-wait lease to $lease_{new}$. The coordinator can be in one of two states, as shown in Table III, where $max(read\text{-}validate\ lease) = lease_{new}$ and $min(write\text{-}wait\ lease) = lease_{new}$. Property 2 is also upheld during the Commit phase, maintaining safety.

	Intermediate	After changing
read-validate lease	$min(lease_{old}, lease_{new})$	$lease_{new}$
write-wait lease	$max(lease_{old}, lease_{new})$	$lease_{new}$

TABLE III: The leases during the Commit phase.

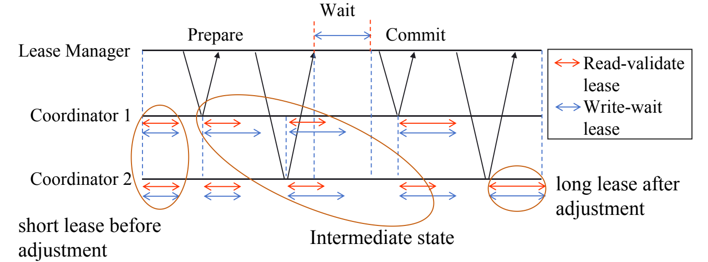


Fig. 9: An example of a successful lease adjustment in which the lease is adjusted to a longer lease.

The previous sections described how to adjust leases, ensuring that the leases remain the same when not adjusted. Next, we introduce the specific implementation of lease adjustment. In this implementation, the read-validate lease and write-wait lease are compressed into an 8B value shared among all coordinators on the same machine. When a coordinator begins a transaction, it reads the lease from this value. Therefore, the lease adjustment process only involves modifying this variable. When the *Lease Manager* needs to adjust the lease, it first issues RDMA write operations to update the leases to an intermediate state. Once this process is complete, the *Lease Manager* must wait a period of the old lease duration to expire the old lease set on the record. After that, the *Lease Manager* will issue another round of RDMA writes to update all leases to their latest state. The entire process is illustrated in the figure, and Properties 1 and 2 are maintained throughout. We observe that the entire lease adjustment process requires two rounds of RDMA writes and a waiting period, totaling $\sim$20us.

During transaction processing, the coordinator records statistics for read-only transactions, including delays for each phase, commit delays, and omission rates. The lease manager periodically reads these statistics via RPC and calculates the success rate of transactions that omit the validation phase. In our experimental setup, the statistics are read once per second. If the lease manager finds that the lease is either too large or too small for the transactions, it will compute a new lease and initiate adjustments. In our system, the lease is set to be large enough to omit the validation phase for at least 80% of read-only transactions. For example, if the omission rate drops below 80% in our setup, the lease manager will calculate a larger new lease based on the delays in the statistics. Conversely, if it finds that the lease is significantly larger than the execution delay, it will adjust the lease to a smaller value, as the lease is deemed too large.

B. Handle Clock Drift

In our system, each replica's clock is regulated by a crystal oscillator. Oscillators in production networks can have a frequency variation of ±100 ppm (parts per million), meaning a drift within the range of ±100µs per second [21]. However, the magnitude of this clock drift, typically within the range of ±1ns per lease, is significantly smaller than the lease duration(tens of microseconds, comparable to network RTT), thus having a negligible impact on design correctness.

To further accommodate clock drift, we introduce an offset to the lease, allowing tolerance for a bounded level of drift. Similar to Spanner's approach [22], the lease intervals in replicas can be conceptualized as uncertainty intervals of [lease - offset, lease + offset] in which the offset can be seen as the buffer bound. Then the read-validate lease is set to *lease - offset*, while the write-wait lease is set to *lease + offset*. Currently, we use a bound of 1000ppm which is 10 times higher than the maximum observed rate drift across 6.5 million server hours [23]. Then the offset in our system is set to be 20ns if the lease is 20us which does not impact the performance as the experiment VIII-G shows.

The above mechanism is sufficient assuming the buffer bound is ensured. To identify and isolate replicas with excessive clock drift, replicas periodically measure their clock's offset from others using the Precision Time Protocol (PTP). PTP [24] now can achieve a deviation of 100ns which provides sufficiently high accuracy for our design. Any replica exceeds *max_drift_offset* by more than 80% will self-terminates.

VIII. Evaluation

A. Evaluation Setup

Testbed. We use four machines to simulate a distributed environment and evaluate the performance benefits of OOCC, each with two Intel Xeon Gold 6240R CPUs (96 cores in total), 384 GB DRAM (32 GB×12), and a 200 Gbps Mellanox ConnectX-6 InfiniBand RNIC. These machines are installed with Ubuntu 20.04 LTS (Linux kernel 5.4.0) and Mellanox OpenFabrics Enterprise Distribution for Linux (MLNX_OFED) v5.3-1.0.0.1. Two machines are leveraged as the compute pools to run coordinators.

Comparisons. We implement OOCC using C++ and compare it with three state-of-the-art RDMA-based distributed transaction processing systems, i.e., FORD [1], DrTM [10], and DSLR [11]. We re-implement their transaction process in our framework for an apples-to-apples comparison.

Implemetion. In this framework, we use the hash tables with the MurmurHash [25] hash function to organize the data store. The data are stored in the hash table as the fix-size records for direct access. There is a global cache in a single machine shared by all the coordinators in the machine to cache the remote data addresses for efficient one-sided RDMA operations. The coordinator can directly access the record if the cache hits. Otherwise, the coordinator needs to calculate the remote bucket address by the MurmurHash function and retrieve the whole bucket to find the record. The lease of DrTM is set to be 400us as set in the origin paper.

B. Methodology

Following the methodology of existing disaggregated transaction works [1], [26], we conduct transactions on a disaggregated hash-based key-value store to assess the influence of various factors on our design. We vary the number of coordinators for each workload until the bandwidth is saturated and choose the configuration with the best performance to report. We explore five key questions about our OOCC design:

Q1: Advantages of OOCC in Read-Heavy Scenarios OOCC's primary strength lies in optimizing read-only transactions, significantly improving both throughput and latency. We use the YCSB [27], TAOBench [28], and TATP [29] as typical read-heavy workloads. Yahoo! Cloud Serving Benchmark(YCSB) is a representative workload of large-scale online services. It features queries accessing single random tuples following a Zipfian distribution, where the θ parameter dictates the level of contention. Similar to existing works, we report on two read-write ratios: **Read-only**, comprising solely read-only transactions, and **Read-intensive**, with 90% read-only transactions. The record size of YCSB in our experiment is set to be 8 Byte. The TAOBench that captures the social graph workload at Meta represents a read-mostly workload that contains 95% read-only transactions and 5% read-write transactions. A read-only transaction may query ranging from a few to hundreds of objects and edges. We generate 500k edges and 1 million objects following the distribution given by TAOBench. The TATP benchmark, derived from a telecom equipment manufacturer's test program, includes 4 tables with record sizes up to 48B, and a workload of 80% read-only and 20% read-write transactions, in which we implement the read-only transactions to include two key reads for our experiments.

Q2: OOCC's Performance in High Contention Scenarios To assess how effectively OOCC handles high contention, we utilize various configurations of the YCSB benchmark. We focus on skewed access patterns, with a Zipfian distribution skewness of 0.99, and write ratios ranging from 0 to 20%. This experiment is designed to showcase the effectiveness of OOCC's optimizations, such as delayed write lock and write unlock, in read-intensive yet high-contention environments.

Q3: OOCC's Impact on Read-write Transactions Although OOCC is optimized for read-heavy environments, our goal is to implement it without compromising the performance of write transactions. To assess this, we incorporate the TPCC [30] benchmark, a widely recognized benchmark for OLTP systems that includes 9 tables and 92 columns. It features five transaction types, with three being read-write. This experiment will help determine whether OOCC adversely affects write-intensive workload performance.

Q4: Impact of Lease Adjustments We employ the YCSB benchmark with varying lease durations to assess the lease's influence on the design. This experiment aims to highlight performance improvements associated with lease adjustments.

Q5: Impact of Offset and Data Size We employ the YCSB benchmark with varying offsets to assess the influence of offsets and demonstrate the performance degradation caused

by various offsets. The lease of OOCC is set to be 10 microseconds with three different offsets—0.5, 1.0, and 1.5 microseconds. We also employ the read-intensive YCSB workload with different value sizes to see the impact on OOCC.

C. Read Heavy Workloads

The key is selected uniformly in the read-only and read-intensive experiments. The subscriber ID is generated using a non-uniform-random method following the specification.

As indicated in Fig. 10a, OOCC displayed superior performance in all scenarios. It achieved a remarkable throughput of approximately 41 million transactions per second with only read-only transactions. When compared to OCC/DrTM/DSLR, OOCC exhibited throughput improvements of $2\times/3.3\times/2.8\times$ and reduced the median latency by 40%/70.6%/60% in this read-only experiment. In read-intensive, TAOBench, and TATP experiments, OOCC enhanced throughput by 1.2-4$\times$ and reduced median and tail latency by 39%-87%.

Our analysis reveals that OOCC's advantage over OCC stems primarily from the omission of the validation phase. OOCC surpasses DrTM and DSLR because they both leverage 2PL which uses expensive atomic operations. Although DrTM employs a lease-based shared lock to avoid explicit shared lock release, it did not perform optimally. Selecting an appropriate lease duration in DrTM is challenging; a longer lease excessively delays read-write transactions, while a shorter lease often expires under uniform workload access patterns. In such cases, DrTM coordinators initiate additional communications using RDMA CAS to set the lease and RDMA Read to access data until the lease is valid, which are expensive. In YCSB, DrTM processes a single read in two or more RTTs. In TATP, due to TATP's non-uniform access pattern, the hotkeys are more likely to be read again by coordinators in a short time. DrTM's read operations are more likely to encounter records with a valid lease, allowing these reads to be committed in one RTT. However, DrTM still needs to issue two operations, one RDMA CAS to get the read lock and one RDMA Read to get the data. As a result, although DrTM's latency in TATP is lower than OCC, it remains higher than OOCC.

The tail latency represents the latency of the read-write transactions as the read-write transactions have a longer transaction processing progress. Although OOCC needs to wait for the lease to expire, it saves one RTT for the read-write transactions than OCC. Thus the tail latency of OOCC is slightly lower than OCC which achieves the lowest tail latency. DrTM and DSLR have a much larger tail latency due to the use of expensive atomic operations.

For the transactions with an unknown read set or write set, we conduct an experiment of the read-intensive workload that contains 95% read-only transactions. In the experiment, we simulate the transactions with an unknown read set in which the transaction will read two records in the first round and generate another two keys based on these two records. The lease is set to be 12us due to the longer execution phase. In this experiment, all the systems will need 1 more RTT to commit the transactions. Thus, the read-only transaction will cost 2 RTTs for OOCC and 3 RTTs for OCC. As the result shows, OOCC exhibited throughput improvements of $1.7\times/2.3\times/2.1\times$ when compared to OCC/DrTM/DSLR. OOCC also achieves the lowest median latency and tail latency as explained above.

D. High Contention.

Reducing lock duration plays a critical role in improving the performance of high-contention workloads. To show how OOCC works in these scenarios, we conduct experiments with varying write ratios with high conflict. For instance, a 10% write ratio indicates that 10% of transactions are write operations while the remaining 90% are read-only. We also conduct experiments to see the performance of varied contention using the read-intensive workload. The θ is controlled to vary the contention. For instance, $\theta = 0.6$ indicates very few conflicts and $\theta = 0.9$ indicates very high conflicts. We also analyze the performance differences between OOCC and OCC by incrementally implementing each technique: (1) adding Lease; (2) incorporating Write Unlock; and (3) employing Delay Write Lock. Additionally, we also assess the maximum theoretical throughput, where read-write conflicts are intentionally not checked, allowing read-only transactions always commit in one RTT. These results are shown in Figure 11.

+Lease. As shown in Figure 11a, introducing the lease improves throughput by $1.46\times$ in scenarios with only read-only transactions. However, this performance gain diminishes as the write ratio increases. At a 5% write ratio, throughput reverts to OCC levels. As shown in Figure 11b, the throughput drops as the θ increases which means more conflicts. The throughput improves by $1.28\times$ when $\theta = 0.99$. The abort rate in Figure 11c shows that transaction abort rates increase to 70% and drop about 10% than OCC when $\theta = 0.99$.

+Write Unlock. The implementation of 'Write Unlock' reduces the lock duration by 1 RTT, leading to a $1.2\times$ improvement in throughput at a 5% write ratio. This is also reflected in Figure 11b, where the throughput increases by about 5% than OCC+Lease. The abort rates drop more slowly than OCC as the contention increases.

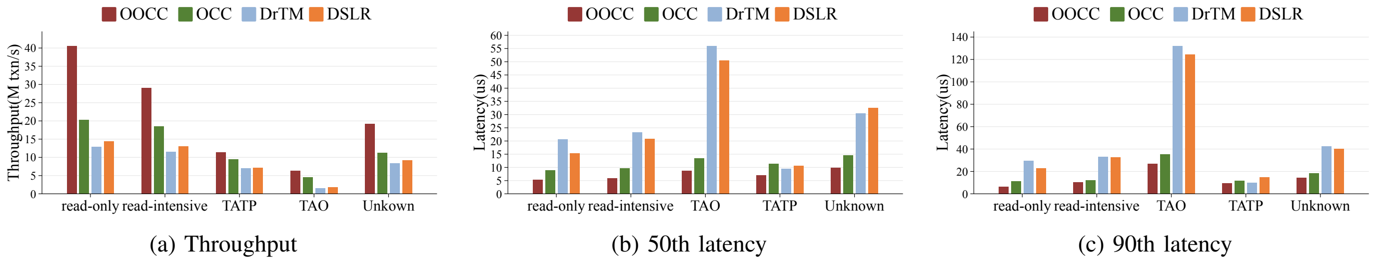
+Delay Write Lock. With 'Delay Write Lock', the lock duration for read-only transactions is shortened to just 1 RTT, enabling more read-only transactions to be committed. This strategy enhances throughput further by $1.1\times$ by the optimization above. The abort rates drop about 17% than OCC which shows the optimization works in contentions.

	New-Order	Payment	Order-Status	Delivery	Stock-Level
ratio	45%	40%	4%	4%	4%

TABLE IV: TPC-C mix ratio in our experiment.

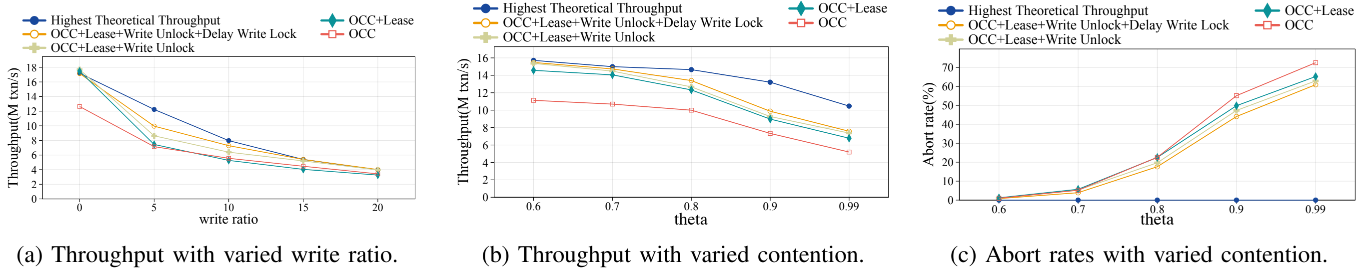
E. TPCC Results

TPCC is a widely used write-intensive benchmark to evaluate transaction performance. We utilize this benchmark to demonstrate if the write-intensive workload is harmed by the OOCC. Table IV shows the mix ratio of the transactions used in our experiment. The experiment uses 8 threads and each



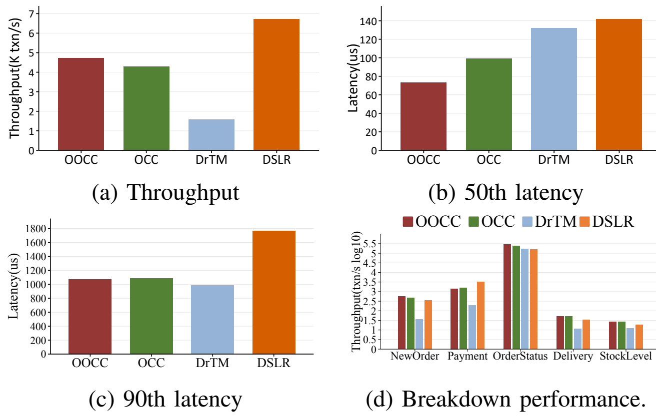
(a) Throughput (b) 50th latency (c) 90th latency

Fig. 10: Performance of read-heavy benchmarks.



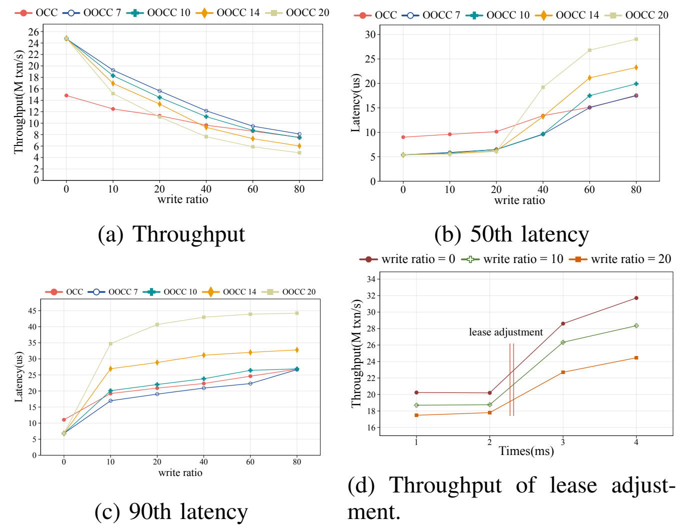
(a) Throughput with varied write ratio. (b) Throughput with varied contention. (c) Abort rates with varied contention.

Fig. 11: Performance of micro-benchmark with varying write ratio and contention.



(a) Throughput (b) 50th latency (c) 90th latency (d) Breakdown performance.

Fig. 12: Performance of TPCC.



(a) Throughput (b) 50th latency (c) 90th latency (d) Throughput of lease adjustment.

Fig. 13: Performance of micro-benchmark with varying lease time and write ratio.

thread generates 4 coroutines as a total of 32 coordinators to evaluate the throughput and latency of each system. We generate 8 warehouses with 10 districts per warehouse. The lease is set to be 80us since the progress of the transaction is very complicated. The performance of each system is shown in Figure 12. As we can see, OOCC still outperforms OCC and DrTM by $1.1\times/3.1\times$ and it also achieves the lowest 50th latency since it can omit the validation phase. The tail latencies of all the systems are the latencies of *StockLevel* transaction as this type of transaction requires dozens of rounds of retrieving the records. In this case, the lease does not help with OOCC as it will expire. Thus OOCC and OCC have similar tail latency.

In contrast, DSLR is a queue-based read-write lock design that is specially designed for write-intensive workloads. DSLR will queue the conflict transaction instead of aborting them. However, DSLR will wait for dozens of locks before executing the transaction due to the complexity of TPCC. OOCC's throughput is 30% slower than DSLR due to a higher abort ratio. While the latency of OOCC is still 40% lower since DSLR will wait for all the conflict transactions to commit.

The performance breakdown is shown in Figure 12d. OOCC achieves the best throughput in all the transaction types except the *Payment*. The transaction type *OrderStatus* is the read-only transaction that reads dozens of keys. OOCC achieves the best throughput of 282K in this transaction type with an improvement of $1.3\times/1.6\times/1.8\times$ than OCC/DrTM/DSLR. *StockLevel* is also a read-only transaction but with dozens of rounds of reads. All systems achieve the lowest throughput in *StockLevel* transactions. The other types of transactions are all

read-write transactions. The operations in *Payment* are writes mostly which benefits DSLR. OOCC only achieves a throughput of 1.5K which is only half of DSLR. But in *NewOrder* and *Delivery*, OOCC has an improvement of 1.2×/15.3×/1.52× and 1.1×/4.6×/1.46× than OCC/DrTM/DSLR.

F. Varying The Lease

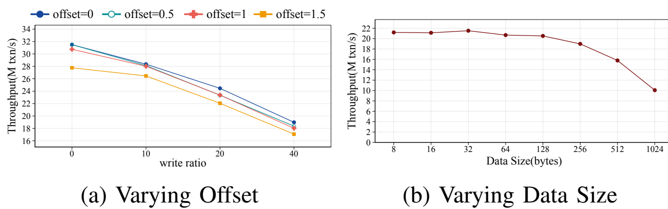
OOCC uses the lease to optimize the read operations with a trade-off of delaying the updates until the lease expires. Clearly, the performance of OOCC highly depends on the length of the lease. To analyze how the lease impacts the performance of the transactions, we use the YCSB workload to show the results with increasing lease length. The leases used are set to be 4, 7, 10, and 14us respectively.

The results are shown in Figure 13a. As the write ratio increases, the throughput drops quickly with the longer lease because it hampers write transactions more. When the lease is 4us, the lease is too small for transactions to omit the validation phase. The performance of OOCC and OCC is the same. When the lease is 7us and 10us, the throughput of OOCC is higher than OCC in all experiments.

As shown in Figure 13b, the median latency of the transactions will all increase as the write ratio increases. We can see that with a proper lease, the median latency will not increase. As shown in Figure 13c, when the lease is 7us, the tail latency of OOCC is smaller than OCC because of the *Write-Unlock*. With a longer lease, the latency of read-write transactions becomes longer due to the waiting.

OOCC dynamically changes the lease to achieve better performance. To verify the effectiveness of the lease-changing scheme, we conduct a simple experiment with the YCSB workload which doesn't change throughout the whole experiment. The lease is first set to be 4us which is too small to omit the validation. Then the lease is changed by the *Lease Manager* lately, so most of the transactions will commit the transaction within the lease. The result is shown in Figure 13d which increases by up to 1.5×. And the lease adjustment can be quickly finished without harming the performance.

G. Offset Experiment



(a) Varying Offset (b) Varying Data Size

Fig. 14: Performance with different offsets and data sizes.

In OOCC, the lease is calculated by the offset to handle the clock drift problem Clearly, the length of the offset can impact the performance of OOCC. To explore the effect of the offset on OOCC performance, we demonstrate the outcomes with increasing offset length. The result is shown in Figure 14a. The performance all degrades as the offset increases. The throughput barely degrades when the offset is 0.5us and 1us. And we believe these two offsets are large enough for most scenarios. The throughput drops quickly with 10% degradation after the offset passes 1.5us with less validation omitted.

To explore the impact of the data sizes, we have conducted an experiment with varied record sizes from 8B to 1024B. The results are shown in Figure 14b in which the performance remains steady until the record size becomes very large such as 512B or 1024B. The performance quickly drops as the large record size consumes the bandwidth. Thus, the record size has little impact on OOCC unless the record is too large.

IX. Related Works

Lease. Lease is widely used in file systems for the client-side metadata cache such as NFS v4 [31], PVFS [32], and LocoFS [33]. However, these designs only work in the client-server architecture, not for the disaggregated architecture. Sundial [34] utilizes logical lease and data caching which benefits from the high workload skewness and OOCC works better in read-intensive workloads. However, the logical lease can only guarantee serializability while OOCC guarantees strict serializability. Additionally, the implicit lease of OOCC is more suitable for disaggregated systems.

Disaggregated Memory. Disaggregated memory has become popular in data centers due to high resource utilization and elasticity. Existing works explore memory disaggregation in networks [35], KV stores [36], [37],and hash indexes [38]. rTX [2] uses OCC to build the index on disaggregated memory. Our proposed OOCC is orthogonal to these systems to build a faster transaction system.

Read-Only Optimized Transactions. Due to the importance of read-only transactions, Several works are proposed to optimize read-only transactions with a variant of timestamp schemes [12], [39], [40]. These works all trade the isolation level for the performance of read-only transactions. OOCC shows comparable performance of read-only transactions while preserving strict serializability.

X. Conclusion

This paper introduces One-round Optimistic Concurrency Control (OOCC), a technique that integrates a lease mechanism to bypass the validation phase in disaggregated transactions. By processing the majority of read-only transactions within a single RTT, OOCC achieves a throughput that is 1.2 to 4 times higher in read-heavy workloads. Additionally, we have developed "delayed write lock" and "write unlock" strategies to reduce the lock duration to just 1 RTT. Consequently, OOCC not only demonstrates comparable performance in write-intensive workloads but also achieves minimal latency.

XI. Acknowledgments

We would like to thank the anonymous reviewers for their comments and feedback. This Work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (62141216, 61877035) and Tsinghua University Initiative Scientific Research Program.

References

[1] M. Zhang, Y. Hua, P. Zuo, and L. Liu, "Ford: Fast one-sided rdma-based distributed transactions for disaggregated persistent memory," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022, pp. 51–68.

[2] X. Wei, H. Wang, T. Wang, R. Chen, J. Gu, P. Zuo, and H. Chen, "Transactional indexes on (rdma or cxl-based) disaggregated memory with repairable transaction," *arXiv preprint arXiv:2308.02501*, 2023.

[3] M. Zhang, Y. Hua, P. Zuo, and L. Liu, "Localized validation accelerates distributed transactions on disaggregated persistent memory," *ACM Transactions on Storage*, vol. 19, no. 3, pp. 1–35, 2023.

[4] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[5] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[6] S. Kashyap, D. Qin, S. Byan, V. J. Marathe, and S. Nalli, "Correct, fast remote persistence," *arXiv preprint arXiv:1909.02092*, 2019.

[7] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance rdma systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 437–450.

[8] A. Cheng, X. Shi, A. Kabcenell, S. Lawande, H. Qadeer, J. Chan, H. Tin, R. Zhao, P. Bailis, M. Balakrishnan *et al.*, "Taobench: an end-to-end benchmark for social network workloads," *Proceedings of the VLDB Endowment*, vol. 15, no. 9, pp. 1965–1977, 2022.

[9] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte *et al.*, "F1 query: Declarative querying at scale," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1835–1848, 2018.

[10] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 87–104.

[11] D. Y. Yoon, M. Chowdhury, and B. Mozafari, "Distributed lock management with rdma: Decentralization without starvation," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1571–1586.

[12] H. Lu, S. Sen, and W. Lloyd, "{Performance-Optimal}{Read-Only} transactions," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 333–349.

[13] X. Chen, H. Song, J. Jiang, C. Ruan, C. Li, S. Wang, G. Zhang, R. Cheng, and H. Cui, "Achieving low tail-latency and high scalability for serializable transactions in edge computing," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 210–227.

[14] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 1493–1509.

[15] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch, "Controlled lock violation," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 85–96.

[16] Z. Guo, K. Wu, C. Yan, and X. Yu, "Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 658–670.

[17] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[18] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper, "Low-latency communication for fast dbms using rdma and shared memory," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1477–1488.

[19] T. Ziegler, C. Binnig, and V. Leis, "Scalestore: A fast and cost-efficient storage engine using dram, nvme, and rdma," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 685–699.

[20] Y. J. Al-Houmaily and G. Samaras, "Two-phase commit." 2009.

[21] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkipati, P. Chandra *et al.*, "Sundial: Fault-tolerant clock synchronization for datacenters," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 1171–1186.

[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[23] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro, "Fast general distributed transactions with opacity," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 433–448.

[24] F. Rezabek, M. Helm, T. Leonhardt, and G. Carle, "Ptp security measures and their impact on synchronization accuracy," in *2022 18th International Conference on Network and Service Management (CNSM)*. IEEE, 2022, pp. 109–117.

[25] A. Appleby, "Murmurhash," *URL https://sites. google. com/site/murmurhash*, 2008.

[26] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 54–70.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[28] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "Tao:facebook's distributed data store for the social graph," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 49–60.

[29] S. Neuvonen, A. Wolski, M. Manner, and V. Raatikka, "Telecom application transaction processing benchmark," *Retrieved January*, vol. 18, p. 2022, 2011.

[30] T. P. P. C. TPC-C, "Standard specification," *Version*, vol. 1, p. 0, 2014.

[31] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow, "The nfs version 4 protocol," in *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*. Citeseer, 2000.

[32] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "{PVFS}: A parallel file system for linux clusters," in *4th Annual Linux Showcase & Conference (ALS 2000)*, 2000.

[33] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, "Locofs: A loosely-coupled metadata service for distributed file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[34] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas, "Sundial: harmonizing concurrency control and caching in a distributed oltp database management system," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1289–1302, 2018.

[35] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 249–264.

[36] J. Shen, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "Fusee: A fully memory-disaggregated key-value store," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 81–98.

[37] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, "Rolex: A scalable rdma-oriented learned key-value store for disaggregated memory systems," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 99–114.

[38] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed b+ tree index on disaggregated memory," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1033–1048.

[39] X. Wei, R. Chen, H. Chen, Z. Wang, Z. Gong, and B. Zang, "Unifying timestamp with transaction ordering for {MVCC} with decentralized scalar timestamp," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 357–372.

[40] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The snow theorem and latency-optimal read-only transactions," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 135–150.