Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions

Tianyang Jiang, Guangyan Zhang, Zhiyue Li,
and Weimin Zheng, *Tsinghua University*

https://www.usenix.org/conference/fast22/presentation/jiang

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions

Tianyang Jiang, Guangyan Zhang*, Zhiyue Li, Weimin Zheng
Department of Computer Science and Technology, BNRist, Tsinghua University

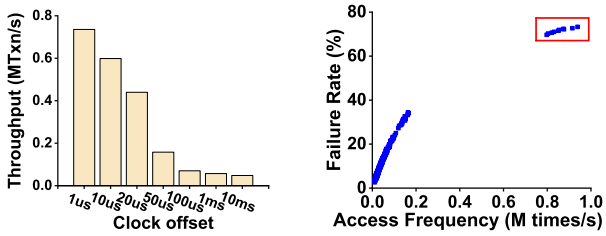
Abstract

Flourishing OLTP applications promote transaction systems to scale out to datacenter-level clusters. Benefiting from high scalability, timestamp ordering (T/O) approaches tend to win out from a number of concurrency control protocols. However, under workloads with skewed access patterns, transaction systems based on T/O approaches still suffer severe performance degradation due to frequent transaction aborts.

We present Aurogon, a distributed in-memory transaction system that pursues taming aborts in all execution phases of a T/O protocol. The key idea of Aurogon is to mitigate request reordering, the major cause of transaction aborts in T/O-based systems, in all phases: in the timestamp allocation phase, Aurogon uses a clock synchronization mechanism called 2LClock to provide accurate distributed clocks; in the request transfer phase, Aurogon adopts an adaptive request deferral mechanism to alleviate the impact of nonuniform data access latency; in the request execution phase, Aurogon pre-attaches certain requests to target data in order to prevent these requests from being issued late. Our evaluation shows that Aurogon increases throughput by up to 4.1× and cuts transaction abort rate by up to 73%, compared with three state-of-the-art distributed transaction systems.

1 Introduction

Many online transaction processing (OLTP) applications such as Web service and e-commerce scale out to massive servers with the growing computational and storage demands. Distributed transaction systems pursue high throughput and low latency to meet the requirements of OLTP applications. OLTP applications have two characteristics. First, OLTP workloads feature severe skew in data access frequency, making data hotspots common [4, 6, 18, 35, 39, 51, 56]. Second, requests in transactions usually contain data dependency [3, 11, 12] such as read-modify-write (RMW) operations, which are more complex to handle than overwrite operations due to the dependency of write on read. Multiple RMWs accessing data hotspots concurrently will bring about high contention, which triggers numerous transaction aborts. Aborts will decrease the throughput of distributed transaction systems and increase transaction latency due to retrying aborted transactions.



(a) (b)

Figure 1: (a) Throughput of distributed transaction system with varying clock accuracy. (b) Relationship between request failure rate and access frequency (hotspots are in red box).

Different concurrency control protocols are used in plenty of transaction systems. These protocols are mainly categorized into two-phase locking (2PL) [10, 49], optimistic concurrency control (OCC) [8, 13, 44], and timestamp ordering (T/O) [2, 26]. Under OLTP workloads with the aforementioned characteristics, 2PL systems suffer performance degradation due to low concurrency levels of locks on data hotspots, while OCC systems encounter frequent aborts since numerous RMWs interrupt the execution of read requests. Compared with them, T/O systems show better performance [19] since they can support flexible concurrency so long as multiple requests are served in the timestamp order.

Transaction execution in T/O systems undergoes four phases: allocating timestamps, transferring requests, executing requests and committing transactions. Existing T/O systems can be categorized into *clock-driven approach* [7, 15, 29, 40, 47, 55] and *data-driven approach* [53, 54], both of them still have deficiencies in addressing the problem of transaction aborts. The former only optimizes one of the first three phases. The latter seeks opportunities in the commit

*Corresponding author: gyzh@tsinghua.edu.cn

phase to save transactions that will be aborted, but aborts are often already inevitable at that time. To this end, this paper explores how to tame aborts in all phases of a T/O protocol when designing a distributed transaction system.

A transaction will abort once the execution of any request in this transaction fails. The key reason for request execution failures in T/O systems is that **requests are not executed in the order of their timestamps**. We call this phenomenon *request reordering*. We examine the aforementioned four phases of T/O systems and find that request reordering only occurs in the first three execution phases. Here we introduce three reasons for request reordering in different phases briefly.

1) Inaccurate distributed clocks in the timestamp allocation phase. The inaccurate timestamps allocated for transactions degrade the performance of T/O systems [33, 54]. To verify this, we adjust the clock offset among servers from 1µs to 10ms manually in a T/O system and see a throughput reduction of up to 94% (Figure 1(a)) under the YCSB workload [9].

2) Nonuniform data access latency in the request transfer phase. The nonuniformity of data access latency aggravates request reordering. The latency of accessing remote servers via the network (≈2µs) [22] is at least one order of magnitude larger than that of accessing local memory (≈100ns) [50]. This perhaps makes the order of request arrivals at the destination mismatch the order of request timestamps.

3) Requests with data dependency in the request execution phase. Requests that depend on the previous requests' results in the same transaction are called *dependent requests* in the rest of this paper. In the request execution phase, dependent requests will be issued late due to dependency wait, leading to an execution failure with high probability.

We present *Aurogon*[1], an **all-phase reordering-resistant** distributed in-memory transaction system ensuring serializability. To avoid request reordering, we devise three key techniques to address the three issues, respectively.

To design the clock synchronization for transaction systems, one key challenge is how to achieve high accuracy of distributed clocks under serious CPU interference from foreground transaction processing. With the growing capability of high-speed networks (*e.g.*, RDMA [13, 43]), CPU resources nowadays are becoming the bottleneck in distributed systems [17, 52]. Both foreground transaction processing and background clock synchronization share and even contend for CPU resources. We have observed that *CPU-NIC clock synchronization and NIC-NIC clock synchronization are heterogeneous*. Inspired by this observation, we propose a *two-layer clock synchronization mechanism* called *2LClock*. 2LClock provides a distributed clock with an average accuracy of 41ns under foreground interference.

To mitigate the impact of nonuniform data access latency, one challenge is how to trade off between transaction processing latency and abort rate. Performing requests in a first-come-first-serve (FCFS) order violates the original timestamp order of requests [7], leading to potential transaction aborts. Thus, we have an opportunity to buffer those requests with larger timestamps and defer their execution to tolerate late arrivals of requests. But request deferral will increase the latency of the deferred requests. We further examine the request failure rate of individual data records (Figure 1(b)) and find that transaction aborts mostly arise from failures of requests on data hotspots. In other words, deferrals of requests performed on cold data hardly reduce aborts. So, Aurogon adopts an *adaptive request deferral mechanism* that detects data hotspots dynamically and only defers those requests performed on hotspots.

Finally, to prevent dependent requests from being issued late, we *pre-attach dependent requests to data* when transactions including these requests access the target data for the first time. Specifically, within a transaction, the metadata of dependent requests will be transferred to target servers together with the requests they depend on, and then the metadata will be attached to the corresponding data. Thus dependency wait will not lead to the late arrivals at the destination of these dependent requests.

We implement Aurogon on an RDMA-capable cluster. We compare Aurogon with three state-of-the-art distributed transaction systems (i.e., Sundial-CC[2] [54], DrTM+H [48], and DST [47]) by evaluating them under two typical OLTP workloads with different degrees of contention. Our experiments show that Aurogon achieves up to 4.1× higher throughput and up to 86% lower average latency than prior systems. Our further experiments also show that Aurogon decreases the abort rate by up to 73%.

2 Background and Motivation

In this section, we first show the characteristics of workloads we target (§2.1). Then we profile prior T/O systems and point out their deficiencies (§2.2). A main disadvantage is that existing timestamp allocation schemes are inaccurate, so we need to introduce clock synchronization approaches to solve this issue. One common concern is why existing clock synchronization approaches cannot work well in transaction systems, which is analyzed in §2.3.

2.1 Working Scenario

We conduct detailed profiling on typical OLTP workloads and characterize two important features below.

Dependent requests. There are two kinds of dependent requests in transactions: 1) *key-dependent requests*, determining which key to read or write with the indexing information from previous read requests, 2) *value-dependent requests*, determining the value that they update the records with using the return

[1] Aurogon is a god who controls day and night in Chinese mythology.

[2] We denote the work [54] as Sundial-CC and the work [28] as Sundial-Clock in this paper for distinction.

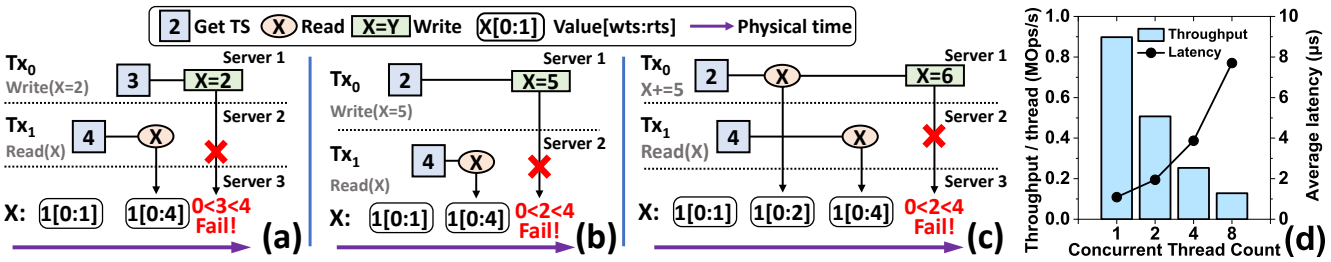


Figure 2: (a-c) Transaction aborts resulting from request reordering, which is caused by inaccurate distributed clocks (a), by nonuniform latency of data accesses (b), and by dependent requests (c), respectively. For brevity, we only plot one request for each transaction. (d) Per-thread throughput and latency when different numbers of threads query the clock of one NIC.

	smallbank	TPC-C	TPC-E
Ratio of transactions with RMWs	90%	92%	25%
Avg. RMW count per transaction	1.41	7.22	3.49

Table 1: Statistics of RMWs in typical OLTP workloads.

results of previous read requests. For instance, in "a[i]+=5", WRITE(a[i]) depends on the result of READ(i) so it is a key-dependent request, and in "a=b+5", WRITE(a) depends on the result of READ(b) so it is a value-dependent request.

With dependent requests, developers can implement complex business logic. For instance, RMWs, the most typical value-dependent requests, are widely used in OLTP applications (Table 1). However, it is complicated to execute dependent requests efficiently in a serializable transaction system. Taking a RMW as an example, any other write performed between the executions of the read and write in the RMW will make the whole RMW fail.

Skew in data access frequency. Data hotspots exist widely in real-world database workloads [4, 6, 18, 35, 39, 51, 56]. A small fraction of data is accessed frequently in a burst with the occurrence of hot events while other data remains cold. For instance, the analysis [5] shows that the top Twitter users had a disproportionate amount of influence indicated by a power-law distribution, so their tweets become hotspots.

2.2 Limitations of T/O Transaction Systems

There is a great deal of prior work [2, 7, 15, 29, 40, 47, 53–55] studying T/O transaction systems. The core idea of T/O systems is to take the partial order relation revealed by timestamps as the serializable order in transaction systems. All transactions in the system reach a consensus on the order that timestamps reveal. A transaction attaches its timestamp to all requests it issues, and requests are performed in the timestamp order on each data record. According to the sources of their timestamps, existing T/O systems lie in two categories:

1) Clock-driven approach. In these systems [7, 15, 29, 40, 47, 55], timestamps allocated for transactions are obtained from local clocks. **Those timestamp allocation approaches pursue scalability but relax the requirement for timestamp accuracy**. We study three typical systems [7, 29, 47] and conclude that they utilize a kind of loosely synchronized clocks, called *chasing clocks*.

Specifically, requests record their timestamps on data during execution and return the largest timestamp recorded on data. These returned timestamps help servers to update their lagging clocks if servers find their clock values are smaller than the returned timestamp. The process makes it seem that all clocks *chase* the fastest one in the system. Although all clocks will catch up with the fastest one eventually, clocks are not accurate instantaneously. Therefore, chasing clocks decrease the performance of transaction systems although they do not violate the correctness.

2) Data-driven approach. Transactions' timestamps are determined by their read or write dependency of committed transactions in these systems [53, 54]. A transaction first obtains the dependencies when accessing data and uses them to determine its timestamp in the commit phase. Data-driven approach seeks opportunities in the commit phase to save transactions that will be aborted by reordering their commit timestamps without violating the known dependency. However, a majority of aborts have been inevitable at that time due to request reordering during execution.

Implications. Existing T/O systems hardly alleviate aborts due to either inaccurate timestamps caused by chasing clocks or inevitable request reordering during execution. So our design adopts high-accuracy clock synchronization to allocate timestamps and prevent request reordering in each phase of execution.

Request reordering in each phase. T/O systems execute a transaction via four phases: 1) allocating a timestamp for each transaction, 2) transferring requests to the servers containing required data, 3) executing requests in their timestamp order and returning execution results, and 4) committing the transaction after receiving all acknowledgements successfully.

After analyzing request reordering phase by phase, we list three key reasons: inaccurate distributed clock, nonuniform latency of data accesses, and requests with data dependency. Figure 2(a-c) shows three cases of the reasons, respectively. There are two transactions (Tx_0, Tx_1), which issue read/write/RMW requests to one record (X). [wts, rts] repre-

sents the live period of a certain record version (more details in §4.1). For a version, a write will fail if its timestamp ts satisfies wts ≤ ts < rts, and a successful read may increase rts.

In Figure 2(a), inaccurate distributed clocks cause the order of two transaction timestamps to not be consistent with the physical time order in which they get timestamps. If it takes the same time for Tx_0 and Tx_1 to access X, Tx_0 starting later will abort due to its smaller timestamp.

Figure 2(b) illustrates that nonuniform latency of data accesses incurs request reordering though clocks are accurate. Tx_1 starting later accesses X earlier than Tx_0 since X and Tx_1's execution reside on the same server. So Tx_0 aborts due to the late arrival of its request. The case occurs frequently in a large heterogeneous cluster because the latency of accessing data from different servers is usually nonuniform due to different network hop counts and uneven traffic distribution.

Figure 2(c) shows that dependent requests also lead to reordering. In this case, "X=6" in Tx_0 is such a request since it depends on the result of the previous read. The event "X=6" arrives at X later than Tx_1's read results in Tx_0's abort because "X=6" must wait for the return of Tx_0's read to calculate the value to be written.

2.3 Clock Synchronization Approaches

Clock synchronization dedicated to transaction systems should meet three requirements: 1) high accuracy, 2) low calling overhead, and 3) resistant to CPU interference. To achieve the three requirements, clock synchronization should adopt differentiated methods on CPU-NIC synchronization (CPU-NIC sync) and NIC-NIC synchronization (NIC-NIC sync). Prior work [16, 28, 32, 40] ignore the heterogeneity between CPU-NIC sync and NIC-NIC sync, so they cannot be applied to transaction systems directly.

Observation. *CPU-NIC sync probes show a larger latency fluctuation than NIC-NIC sync ones in transaction systems.*

The large fluctuation of probe latency will impair the accuracy of synchronization [16]. A longer execution time of a probe suggests the readings of two probed clocks can differ, leading to inaccurate clock synchronization. Prior work [16, 28] believes network traffic is the main reason for inaccurate probes so they focus on eliminating the effect of link noise. However, that does not cover all cases especially when foreground applications are CPU-intensive instead of network-intensive. Table 2 illustrates the latencies of both CPU-NIC sync and NIC-NIC sync probes under DrTM+H [48], a CPU-intensive distributed transaction system which saturates RDMA networks to accelerate transaction processing. The P999/median ratio of network probe latency only rises by 37% when adding the DrTM+H load. It suggests that peak network traffic from this advanced distributed transaction system does not affect the stability of network probes seriously. On the other hand, this ratio reaches 6.75 for CPU-NIC sync probes. This happens because getting CPU timestamps suffers from latency spikes when the CPU is preempted by foreground transaction processing.

	CPU-NIC sync		NIC-NIC sync	
	w/o load	w/ load	w/o load	w/ load
median (µs)	1.26	1.86	0.95	1.05
P999 (µs)	1.33	12.56	0.97	1.47
P999/median ratio	1.06	6.75	1.02	1.40

Table 2: Execution time of two kinds of probes without load or under the DrTM+H load.

Based on the heterogeneity we observed, existing clock synchronization approaches have four limitations. First, they consume precious CPU resources in transaction systems when using a dedicated core [40] to poll requests via user-level network interfaces, decreasing transaction throughput. Second, without separating CPU-NIC sync and NIC-NIC sync [32, 40], high accuracy cannot be achieved due to highly variable software stack latencies [16]. Third, the synchronized NIC clocks (*e.g.*, PTP [21], HUYGENS [16]) cannot serve intensive queries from transaction systems. Figure 2(d) illustrates that as the number of threads querying the clock of one NIC increases, the average query latency rises sharply and the per-thread throughput also decreases. Finally, some of them [27, 28] rely on customized modification to NICs, making their extensive deployment in datacenters difficult.

3 System Overview

We propose Aurogon, an all-phase reordering-resistant distributed in-memory transaction system, with solutions targeting the three issues discussed in §2.2.

Design rationale. The key idea of alleviating request reordering is to maintain the timestamp order for requests in all phases of transaction execution.

First, in the timestamp allocation phase, we propose a clock synchronization mechanism to improve the accuracy of timestamps obtained by transactions. The clock exploits a two-layer architecture, divided into CPU-NIC sync and NIC-NIC sync. The high accuracy achieved by 2LClock resists the CPU interference from foreground transaction processing.

Second, in the request transfer phase, Aurogon buffers received requests and defers the execution to mitigate the impact of nonuniform data access latency. To cut the latency rise caused by request deferrals, Aurogon only defers the requests performed on hotspots and shortens the request deferral time heuristically.

Third, in the request execution phase, to prevent dependent requests from being issued late, Aurogon pre-attaches the metadata of dependent requests to data when their transactions issue requests to the same data at the first time. This mechanism not only diminishes execution failures of dependent requests, but also saves one network communication for dependent requests.

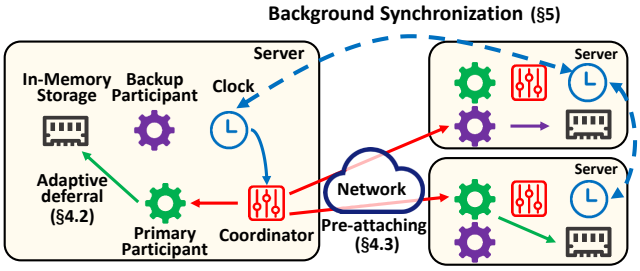


Figure 3: Aurogon architecture.

Aurogon Architecture. Figure 3 illustrates the Aurogon architecture. Servers are connected with a fast network. Data is partitioned into multiple shards and each server maintains one of data shards in its memory. All servers are homogeneous and transactions can start to execute in any server. Each server has four modules: a coordinator, a primary participant, a backup participant and a clock.

- The coordinator is responsible for coordinating transactions. It sends read/write/commit requests to primary participants and it decides to commit or abort transactions based on the replies. The coordinator also *pre-attaches the metadata of dependent requests to data* (§4.3).
- The primary participant processes requests in their timestamp order by accessing local in-memory storage, and finally sends their execution results to coordinators. Meanwhile, the primary participant conducts the method of *adaptive request deferral* to cut aborts.
- The backup participant replicates the commit messages from a coordinator before transactions are committed to the primary participant to tolerate a server failure.
- Clocks provide accurate timestamps for transactions and are synchronized via *2LClock* (§5).

4 Concurrency Control Protocol Design

4.1 Protocol Specifications

We first introduce the data management in Aurogon. Aurogon leverages a multi-version mechanism to store historical versions of data records and distinguishes data versions with timestamps obtained from 2LClock. Each data version has an active range of timestamps bounded by the write timestamp (*wts*) and the read timestamp (*rts*). Specifically, *wts* is the timestamp of the transaction that has created this version and *rts* indicates the maximum timestamp of transactions that read this version. The versions of a record are organized by a linked list sorted by their *wts* in descending order. Each version also contains a *status* that indicates whether this version is committed.

We now describe the rules for handling requests in Aurogon. Read requests always succeed while write requests may fail during the execution if they conflict with the committed reads. Reads are regarded as `COMMITTED` upon finishing execution so they do not demand an extra commit message, but writes require an extra communication with participants to ensure that all writes of the transaction are performed successfully.

Algorithm 1 shows the protocol of transaction processing in Aurogon. The coordinator performs a transaction T beginning with getting a timestamp (line 3) from local 2LClock (§5). The timestamp indicates the serial order of the transaction and is attached to all requests issued by T. Aurogon encodes the acquired timestamp together with the server and thread IDs of the coordinator into a 64-bit timestamp to make it globally unique.

In the request transfer phase, the coordinator traverses T's read and write sets dynamically, and sends its read requests to the participants[3] maintaining the required data (line 5-7). If a write depends on the return of the read accessing the same data record (*e.g.*, the write in a RMW), the read will piggyback the metadata of these *dependent writes*. Meanwhile, *independent writes*, which do not contain data dependencies, are issued to corresponding participants directly (line 8-9). It should be noted that transactions in Aurogon do not require knowing read and write sets before their executions. Specifically, transactions can add a key-dependent request into the read or write set while running, and then the coordinator will issue such a key-dependent request.

The coordinator receives return messages of T's requests from participants and aborts T once it obtains the failure return of any request (line 10-13). If T is aborted, the coordinator will notify the participants which have performed T's requests successfully to rollback the changes brought by T's requests. When T's all requests are returned successfully, the coordinator will start the commit phase, issuing commit messages to the participants containing the records in T's write set (line 17-20). Finally T can be committed to users.

`Tx_read` shows the execution of read requests in participants. If a read request R is issued together with a dependent write W_d, the participant will first install W_d into the data list (line 24-26). The installation process is illustrated in `Tx_write` later. Then the participant determines whether the execution of R will be deferred according to the hotness of data R accesses (line 27-28).

After a possible deferral, R starts to read by searching for the correct version based on its timestamp: it traverses the data list in the descending timestamp order (line 29) and chooses the first version v which satisfies $v.wts < R.ts$. In other words, v has the largest *wts* among versions whose *wts* are smaller than $R.ts$ (line 31). The participant will extend $v.rts$ to $R.ts$ if $v.rts < R.ts$ to show the existence of R (line 32). If $v.status$ is `PENDING`, which indicates that v is not yet committed, R will wait until $v.status$ turns to `COMMITTED` (line 33-34). Aurogon chooses to block R instead of reading uncommitted data [26] to prevent the high overhead caused by cascading aborts. Finally, R returns the correct data version to the coordinator.

[3] Participants refer to primary participants in the rest of this paper unless explicitly stated otherwise.

Algorithm 1: Pseudo code of Aurogon's concurrency control protocol.

```
1  Function Coordinate(txn)
2      // timestamp allocation phase
3      ts = get_local_clock_ts()
4      // request transfer phase
5      for read,record in txn do
6          send(read.dest_node, Tx_read, read, record)
7          //piggyback the metadata of dependent writes
8      for independent_write,record in txn do
9          send(write.dest_node, Tx_write, write, record)
10     while receiving an ACK do
11         if ACK.status == failure then
12             abort txn
13             return
14         if key-dependent requests exist then
15             update read and write sets and issue requests
16         // commit phase
17         if all ACKs are received then
18             replicate updates to backup participants
19             for write,record in txn do
20                 send(write.dest_node, Tx_commit, write, record)
21             commit txn // notify upper users
22 // request execution phase
23 Function Tx_read(record, req)
24     if req.dependent_write exists then
25         if install(record, req.dependent_write) fails then
26             return dependent_write failure // reply to coordinator
27     if record.is_hot == True then
28         wait for deferred_interval
29     for version in record.list do
30         // in descending order of version timestamp
31         if version.wts < req.ts then
32             version.rts = max(version.rts, req.ts)
33             if version.status == PENDING then
34                 wait until version.status == COMMITTED
35             reply version's data to coordinator
36             return success
37 Function Tx_write(record, req)
38     if install(record, req) fails then
39         return failure
40     else
41         return success
42 Function Tx_commit(record, req)
43     find req's version in record.list
44     version.status = COMMITTED
```

`Tx_write` shows the execution of write requests in participants. Before a write request W is processed to update the record r, the participant needs to ensure that no version v of r satisfies $v.wts \leq W.ts < v.rts$. If such v exists, W will conflict with requests which read v and each have a timestamp larger than $W.ts$. It is because these reads should return the updates of W but they have already returned v. W has to fail if such a conflict occurs as it is expensive or even impossible to change these reads' return and abort relevant transactions.

The participant validates W and searches for the correct installation position in this way: it traverses the version list and chooses the first version v_f which satisfies $v_f.wts \leq W.ts$, followed by validating whether $v_f.rts \leq W.ts$. The conflict would only occur on v_f since all versions are sorted in the descending order of wts. If W passes the validation, the participant installs a new version v' in front of v_f, setting the status of v' to `PENDING` and $v'.wts = v'.rts = W.ts$. Now the installation of W succeeds. The same process is also used to install a dependent write in `Tx_read`. Finally the participant returns the success of W to the coordinator.

`Tx_commit` shows the commit process of writes in participants. If the coordinator commits the transaction T, the participant finds the versions of all pending writes issued from T and turns the status of these writes to `COMMITTED`. Then the participant returns the pending reads blocked by these writes.

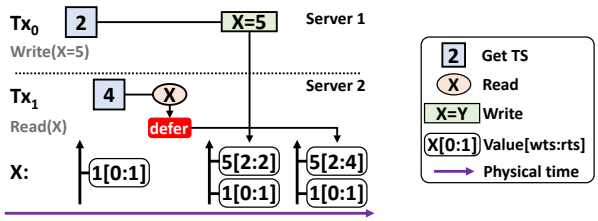


Figure 4: An example of request deferral.

4.2 Adaptive Request Deferral

To decrease the number of transaction aborts, we focus on reducing the failures of write requests since reads never fail in Aurogon. As depicted in Figure 2(a) and 2(b), writes will fail if they conflict with committed reads. If a read R is committed and returns the version v to the coordinator, an incoming write W with $v.wts \leq W.ts < R.ts$ has to fail in order to ensure serializability. Besides inaccurate distributed clocks, nonuniform data access latency causes that W with a smaller timestamp arrives later than R. The issue results from both the data location and the network queuing on transmitting links.

To solve this problem, we propose an adaptive request deferral mechanism to defer the execution of reads until the straggling writes arrive and finish. Figure 4 shows such a deferral case. Compared to Figure 2(b), Aurogon buffers Tx_1's read and defers its execution. "X=5" in Tx_0 benefits from the deferral and can be installed successfully when it arrives at X. Then the deferral ends, and Tx_1's read is performed, returning the latest value 5.

However, a raised challenge is that a read's deferral will increase the transaction latency inevitably. We tackle the difficulty by two methods: reducing the number of unnecessary deferred reads and shortening the request deferral time.

Cutting the deferred read count. We examine the failure rate of individual data records and observe that transaction aborts usually arise from request reordering on hotspots. The hotspots account for a small portion of user data but encounter enormous request failures. Figure 1(b) illustrates that the

failure rate rises sharply when data is hot. So the request deferral on hotspots can be much more effective.

To this end, we propose a lightweight hotspot detection scheme. The scheme counts the number of receiving requests for each record individually in last 10 milliseconds using accurate timestamps allocated for requests. If the throughput of a particular record exceeds a preset threshold, we consider that the record is being accessed frequently and at great risk of request failures. Then the request deferral mechanism starts to defer the execution of reads on high-contention records. The space overhead of the detection scheme is quite modest (only an 8-byte counter for each data record). Moreover, the detection scheme can discover the change of hotspots easily so that we will conduct deferrals on the new hotspots.

Shortening request deferral time. Aurogon exploits a heuristic method to increase the deferral time instead of setting an overlong deferral time in advance. The deferral time of each record is determined individually. Aurogon calculates the request failure rate of each record dynamically based on the hotspot detection scheme. The deferral time will be increased heuristically to tolerate late arrivals of more writes if the failure rate exceeds a preset threshold. If the request failure rate of the hotspot remains low, Aurogon could cut the deferral time conservatively to avoid the waste of deferral time. Our evaluation shows that a 20-microsecond deferral for reads on hotspots cuts 50% write failures in most cases.

Note that the request deferral mechanism is a best-effort method which cannot eliminate the write failures completely. However, we could trade a moderate latency rise of a handful of transactions for reducing aborts.

4.3 Pre-attaching Dependent Requests to Data

A dependent request may fail during execution since it has to wait for the results of requests it depends on within the same transaction (Figure 2(c)). Dependent requests are classified into key-dependent requests and value-dependent requests (§2.1) and we first target value-dependent requests.

Value-dependent requests are writes depending on the previous reads' results and the write set of value-dependent requests are deterministic. We first consider the situation that writes are issued to the same records with previous reads. such as an auto-increment counter. Taking RMW as an example, Figure 5 illustrates that three network communications are consumed in T/O systems [2, 19] without pre-attaching between the coordinator and the participant: 1) a read R is issued to the record X and obtains the correct data, 2) a preparing write request "X=8", PW, is sent to the record carrying the updated data, and 3) a commit or abort message C notifies the participant to commit or rollback the updates from PW. It leaves a *vulnerable interval* from the time point R finishes to the time point PW arrives, in which any arriving write W with $W.ts > PW.ts$ will break the integrity of RMW's execution and

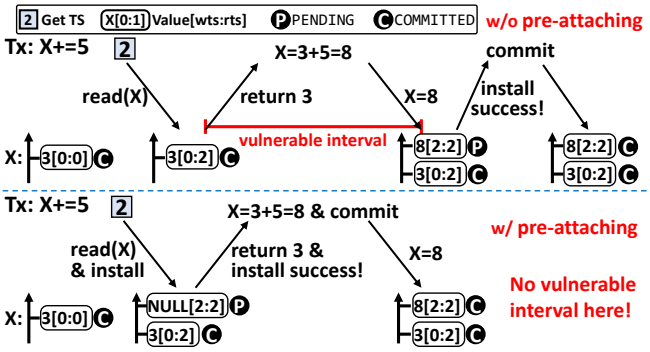


Figure 5: Dependent request handling with or without pre-attaching.

cause PW's failure.

To this end, Aurogon piggybacks the metadata of dependent requests to target data while transferring requests they depend on, and pre-attaches these dependent requests to data records for the purpose of eliminating the vulnerable interval (Figure 5). Specifically, the metadata of PW is issued to the participant together with R in the first communication, and the participant first tries to install PW to the data list before performing R (line 25 in Algorithm 1). If the installation fails, there is no need to perform R and the participant can return a failure message to the coordinator, resulting in an early abort of the transaction which saves CPU resources (line 26).

After installing the version of PW, the participant sets the version's status to `PENDING` and leaves its value NULL since the coordinator has not obtained R's return and could not have calculated PW's new value. Note that leaving the version's value NULL does not violate the correctness as the `PENDING` versions will block incoming reads. The participant performs R and returns the result immediately after installing PW, which eliminates the vulnerable interval. In the commit phase, the participants will receive the message C and commit the version of PW with new data if it is determined the transaction will commit.

Besides eliminating the vulnerable interval, pre-attaching dependent requests brings two extra advantages. First, it saves one network communication compared to prior T/O systems. Second, it assists the transaction in finding early aborts if the installation fails in the first network communication, which cuts unnecessary network bandwidth usage compared to encountering PW's failure after sending PW's data to the participant in Figure 5. Meanwhile, since the metadata piggybacked is just a boolean variable to show the existence of a dependent write, the latency of transferring requests are not affected.

We now discuss other situations of dependent requests. For other value-dependent requests (*e.g.*, PW accesses the records different from R), Aurogon supports issuing R followed by transferring the metadata of PW to its corresponding record for pre-attaching, which shortens the vulnerable interval greatly as well. For key-dependent requests, it is inevitable to wait for previous reads' results to determine which data

to access. Aurogon introduces the modification of adaptive request deferral mechanism to benefit these requests. Previous reads' executions are accelerated by disabling their possible deferral in participants selectively. Furthermore, for performing key-dependent requests, a key-dependent read can be performed directly since its dependency wait in the coordinator is regarded as an implicit deferral, while a key-dependent write may benefit from read deferrals on hotspots.

4.4 Aurogon's Isolation Level

Aurogon ensures serializability and here we only give a simple proof sketch. Each transaction has a globally unique timestamp. Requests are executed in the order of their transactions' timestamps. So each transaction can achieve a view of the system and update the system's status consistently with its globally unique timestamp. Therefore, transactions in Aurogon are serializable, and the timestamp order is the serial order of transactions.

Aurogon also ensures strong partition serializability (SPS) [1, 7, 41] as an extension. SPS is an isolation level slightly weaker than strict serializability [20, 38]. In a SPS system, for any two transactions T_1 and T_2, T_1 must precede T_2 in the serial order if two conditions are satisfied: 1) T_2 starts after T_1 finishes, 2) T_1 and T_2 access the same record. One can achieve strict serializability easily in SPS systems by adding explicit out-of-band dependencies with a cross-record read, solving the anomaly usually called "causal reverse".

Two main anomalies which can happen in serializable Aurogon are "stale reads" and "immortal writes" [1] compared to SPS. The key reason for both anomalies is that timestamps 2LClock allocates cannot match the physical time completely in a distributed system. A transaction could obtain a smaller timestamp so it cannot see the newest updates.

We solve the two anomalies by adding some modifications to request processing in participants. To prevent stale reads, reads cannot return a version v if there exists a `COMMITTED` version $v^{'}$ that $v^{'}.wts > v.wts$. To prevent immortal writes, a write W cannot be inserted if there exists a `COMMITTED` version v that $v.wts > W.ts$. Besides, coordinators should not commit a transaction T to users until receiving all of its commit acknowledgement messages. An early commit of T may cause that transactions starting later will find the status of T's updates is `PENDING`, which should have been `COMMITTED`, leading to the incorrect execution of two aforementioned modifications.

4.5 Fault Tolerance

To tolerate the server-level crash, Aurogon leverages the widely-used primary-backup replication similar to prior work [14,24,48]. Aurogon does not require to replicate coordinators since their commit decisions and states can be recovered from the states of primary participants and backup participants. If a transaction T is determined to be committed, its coordinator first replicates T's updates to backup participants. Backup participants perform transactions' updates asynchronously to survive primary participant failures. After receiving all ACKs from backup participants, T's updates can be committed to primary participants and T can be returned to users simultaneously.

Failures of coordinators. The approach for handling the coordinator failure is to finish transactions having been committed to users and abort running ones it coordinates. If primary participants detect the failure of a coordinator with periodic heartbeats, they need to judge whether the transactions coordinated by this coordinator has been committed to users. They first check if updates of the transaction exist in corresponding backup participants. If all backup participants retain the updates, it means the transaction is allowed to return to users so that primary participants will search for the transaction's updates in memory and commit them. Otherwise, primary participants will discard these `PENDING` updates and inform backup participants to roll back the possible changes.

Failures of participants. The updates of a transaction will first be replicated to backup participants followed by primary participants, so coordinators can select a new primary participant from backups with a lightweight consensus [25, 37] after detecting a primary participant failure. The correctness is guaranteed since backup participants have retained all transactions' updates that corresponding primary participants received. The failure of a backup participant can also be recovered by the consensus protocol.

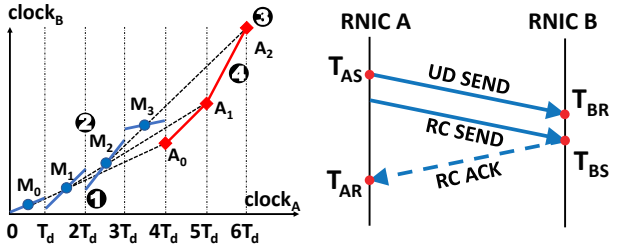
5 2LClock Design and Implementation

Inspired by the observation in §2.3, we propose *2LClock*, a clock synchronization mechanism using a two-layer mapping scheme to provide a global clock for each server. We further implement 2LClock with the help of an emerging network technique, RDMA [23, 31, 34, 43, 46]. 2LClock meets three requirements of clock synchronization proposed in §2.3.

- To achieve high accuracy, 2LClock separates CPU-NIC sync and NIC-NIC sync to alleviate the latency fluctuation from the software stack, and issues synchronization probes frequently to resist clock drift.
- To reduce the overhead of querying timestamps for transactions, 2LClock implements CPU-NIC sync to enable transactions to avoid querying NIC clocks directly.
- To resist the CPU interference from transaction processing, 2LClock filters inaccurate synchronization probes and cuts the CPU usage of synchronization with the help of RDMA.

5.1 Clock Mapping Functions

In 2LClock, all CPU clocks in the cluster synchronize with one preset "reference" NIC clock. Specifically, for a server, its CPU clock is synchronized with its local NIC clock, using the mapping function F_1. Then the local NIC clock uses the mapping function F_2 to synchronize with the reference NIC



(a) Clock mapping function (b) Probing of NIC-NIC sync

Figure 6: Mapping and probing in clock synchronization.

clock. The principle of constructing a mapping function F_i (F_1 or F_2) is to ensure their increasing monotonicity since distributed transaction systems cannot tolerate individual clocks going backwards. We now present how to construct F_i.

The synchronization accuracy of F_i will be influenced by the fluctuation of individual probes if we update the F_i upon receiving a new probe. So 2LClock divides the time into multiple successive timeslices with a fixed-length T_d and filters out "bad" probes collected in each timeslice to achieve high accuracy. As shown in Figure 6(a), 2LClock derives F_i in four steps. First, 2LClock generates a linear function for each timeslice (❶), which is different between F_1 and F_2 and is elaborated in §5.2. Second, for each timeslice $[i \cdot T_d, (i+1) \cdot T_d)$, 2LClock gets the middle point M_i of the line segment (❷). The third step is to get the set of *anticipated points* (❸). By drawing a line across two middle points M_i and M_{i+1}, 2LClock gets an anticipated point A_i. Finally, by connecting each two successive anticipated points (❹), 2LClock gets a mapping clock function (red lines with diamond symbols in Figure 6(a)).

One may wonder why 2LClock exploits an extension method to obtain a piecewise mapping function. It is because the linear function generated in the first step may not be successive at the junction of two timeslices, which may violate the monotonicity of allocated timestamps.

5.2 Synchronization in a Timeslice

As depicted in the first step in §5.1, to construct either F_1 or F_2, 2LClock first needs to generate a linear function for a timeslice. Here we introduce the ways to generate such linear functions for F_1 and F_2 via synchronization.

5.2.1 CPU-NIC Synchronization

In a timeslice, 2LClock first produces a set of mapping pairs between a server's CPU clock and its local NIC clock by probing and measurement. Then, it fits those mapping pairs with linear regression, generating a linear function for F_1.

To produce a mapping pair dataset, a CPU-NIC sync thread periodically queries its local NIC to get *mapping pairs* $< t_c, t_n >$, which gives an estimation that the time in local NIC is t_n when time in CPU is t_c. Each NIC query generates three timestamps: a NIC timestamp (T_n) from the query reply, and two CPU timestamps (T_{c1} and T_{c2}) that the host CPU records when CPU begins and finishes the NIC query request. So we get the mapping pair $< t_c, T_n >$, where $t_c = \alpha \cdot T_{c1} + (1-\alpha) \cdot T_{c2}$. α is a parameter in $[0,1]$ and we set it offline[4].

After collecting enough mapping pairs within one timeslice, we discard those outlier pairs whose execution duration, $T_{c2} - T_{c1}$, deviates far from the normal value. Resource contention between clock synchronization and transaction processing can occasionally make the execution duration of some mapping pairs to millisecond level. So filtering out outlier pairs can effectively enhance the accuracy of CPU-NIC mapping.

5.2.2 NIC-NIC Synchronization

For NIC-NIC sync, 2LClock uses three steps to generate a linear function in each timeslice for F_2. First, 2LClock produces a set of mapping pairs between local NIC and the reference NIC. The reference NIC is on a randomly picked server in the cluster. 2LClock first gets a quadruple $<T_{AS}, T_{BR}, T_{BS}, T_{AR}>$ by issuing probes from local NIC to the target NIC, and then obtains a mapping pair $<t_n, t_g>$ from such a quadruple. Here we adopt the link symmetric assumption, widely used in prior work [16, 27, 28, 32]. Second, 2LClock uses supported vector machine to filter the mapping pairs deduced from those outlier probes as prior work [16] does. This is because a probe with a finishing time beyond the normal range of measurement encounters network fluctuation with high probability [16], which will violate the link symmetric assumption. Finally, 2LClock generates a linear function for F_2 by fitting mapping pairs with linear regression.

Tree structure of NIC-NIC sync. 2LClock builds a K-ary tree to organize the synchronization links, whose root is the reference NIC. Each NIC issues probes to its parent NIC, so that each NIC can ultimately get a mapping pair synchronized with the reference NIC clock. Each tier of the tree will introduce an additional synchronization error [28]. Thus we set K to 10 intuitively to limit the synchronization error. For example, only a four-tier tree is required to enable 2LClock to work in a 1000-server cluster ($1+10+10^2+10^3=1111$).

RDMA characteristics. To reduce CPU usage, 2LClock exploits two characteristics of RDMA when obtaining the clock mapping pair between a local NIC and its parent NIC with probes. First, RDMA *ibv_cq_ex* can automatically record accurate times when a network request leaves and arrives at an RDMA NIC (RNIC). This interface allows buffering these NIC timestamps in both sides' host memory via direct memory access without CPU involvement. Second, RDMA has multiple transport modes [24, 43] including Reliable Connected (RC) and Unreliable Datagram (UD). Specifically, RC mode uses acknowledgment packets to ensure reliable transmission, while UD mode removes these packets. We have

[4] Theoretically, α is the ratio between NIC query uplink latency and the sum of uplink and downlink latencies. In a homogeneous cluster, α's values on different servers are the same so the relative time of CPU clocks is not affected by α's values.

found an interesting feature of *ibv_cq_ex* in RC mode and used it to probe target RNIC: *ibv_cq_ex* records the leaving and arriving times of RC request's ACK instead of RC request itself. Next we show how to generate a probe and calculate a mapping pair.

Using a combination of RDMA modes. Figure 6(b) plots how a probe gets a quadruple when local RNIC A synchronizes with its parent RNIC B. 2LClock first sends a UD request from A to B with *ibv_cq_ex* and records two timestamps, T_{AS} and T_{BR}, indicating when the UD request leaves A and arrives at B, respectively. Then 2LClock sends an RC request from A to B with *ibv_cq_ex* as well and two timestamps, T_{BS} and T_{AR}, are recorded, indicating when the RC's ACK leaves B and arrives at A. If UD requests suffer packet loss, we discard the probes directly though it seldom happens [24].

Transferring timestamps asynchronously. Finally, the server that B resides on obtains existing timestamps T_{BS} and T_{BR} in a batch by polling local *ibv_cq_ex* [30, 36] periodically, and transfers them to A's server. The clock offset between B and A can be calculated as: $\texttt{offset}_{B-A} = ((T_{BR} - T_{AS}) - (T_{AR} - T_{BS}))/2$, and the mapping pair is $(T_{AS}, T_{AS} + \texttt{offset}_{B-A})$. We also measure the difference of one-way delay between RC and UD requests offline, and then subtract this part when calculating the offset to avoid violating the link symmetric assumption.

The design of probing in 2LClock reduces the CPU utilization of the parent nodes in the tree since we connect a parent node with K (10 by default) child nodes to alleviate additional errors. Specifically, it brings two advantages. First, using RDMA *ibv_cq_ex* to obtain accurate timestamps lowers CPU utilization compared with polling network interfaces frequently with a dedicated core [40]. Second, the combination of RC and UD requests offloads tasks of issuing probes from parent servers to child servers.

5.3 Fault Tolerance

2LClock uses a two-phase mechanism to tolerate a server failure. If a server detects a failure of its parent server with the heartbeat mechanism, it will turn to a new parent to synchronize with it. The selection of the new parent cannot violate the requirements in the child count and the height of the K-ary tree. Such new parent and child servers will perform a two-phase recovery since it will take a while for them to build new connections and warm up their clock mapping functions.

Figure 7 depicts how a child server A safely switches from parent server B to a new parent server C, without breaking its increasing monotonicity guarantee. For brevity, here we consider F_1 and F_2 as a whole, and denote their composite mapping function as F. When A detects that B fails at t_1, A will immediately connect to a new parent C and probe it to construct a new mapping function F_C synchronized with the clock in C (green line in Figure 7).

Given that 2LClock finishes warming up the new mapping

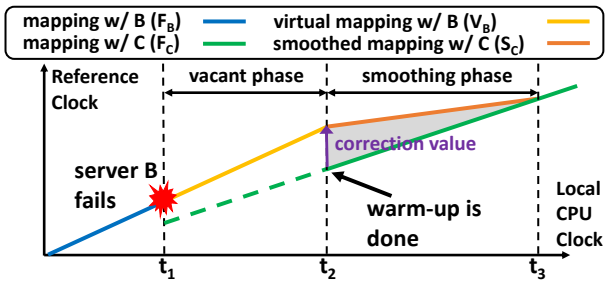


Figure 7: Server failure handling in 2LClock.

F_C at t_2, a new problem is that no mapping is available in A during the *vacant phase* from t_1 to t_2. So 2LClock provides a temporary clock by extrapolating the old mapping function synchronized with failed server B during the vacant phase. We call this mapping as virtual mapping V_B, as it may be inaccurate after a while.

To prevent the time from going backwards, a correction value is added when switching from V_B to F_C. However, different servers add different correction values, leading to inaccuracy of 2LClock. To eliminate the impact of correction value, we further add a smoothing phase from t_2 to t_3 (corresponding to the mapping S_C), during which the correction value decreases gradually till it becomes zero. After t_3, 2LClock comes back to the normal state again.

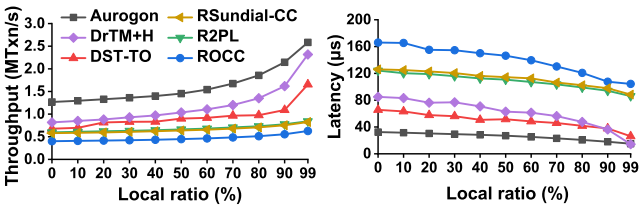
6 Performance Evaluation

6.1 Experiment Setup

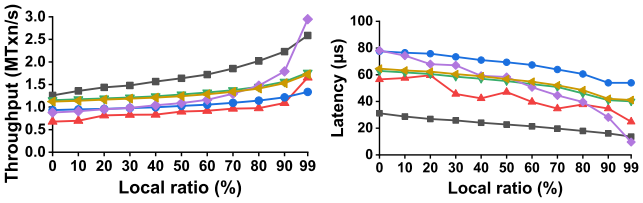
Transaction system setup. We compare Aurogon with five distributed transaction systems. First, we pick the state-of-the-art T/O systems from *clock-driven approach* and *data-driven approach*, respectively. For clock-driven approach, we integrate DST [47] into our system and build a compared system called *DST-TO*. In DST-TO, we replace 2LClock with DST and turn off Aurogon's reordering-resistant techniques. For data-driven approach, we choose Sundial-CC and replace the original TCP network with RDMA for a fair comparison based on an open-sourced implementation [45], called *RSundial-CC*. Second, we compare with *DrTM+H* [48], which saturates RDMA networks to accelerate distributed transactions. Third, we compare with systems based on traditional concurrency control protocols. We use the implementation [45] to evaluate an RDMA version of 2PL and OCC, called *R2PL* and *ROCC*, respectively.

Workloads. We use two workloads, TPC-C and YCSB.

TPC-C [11] is the industry standard benchmark for evaluating OLTP transaction systems, which simulates a warehouse-centric order processing application and partitions all data based on their warehouse IDs. The warehouse count determines the contention degree of TPC-C. We adopt two contention configurations: one warehouse per server to model high contention and one warehouse per thread to model



(a) Under high contention.



(b) Under medium contention.

Figure 8: Throughput and average latency of TPC-C/N.

medium contention. We implement two kinds of TPC-C, named TPC-C/NP and TPC-C/N. TPC-C/NP contains two representative transactions, `NewOrder` and `Payment`, comprising 88% of the default TPC-C mix. TPC-C/N only includes the most complicated transaction `NewOrder` since some compared systems do not implement `Payment`.

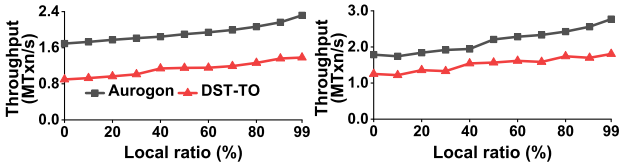
YCSB is a benchmark commonly used for key-value store evaluation as well as transaction system evaluation [29, 42]. We list two main configurable parameters in YCSB: the RMW request ratio `RMW_ratio` and the skew factor θ. Each transaction contains multiple requests (8 in our evaluation) and each request is either read or RMW determined by `RMW_ratio`. Each request accesses a random record based on the Zipf distribution and the θ shows the degree of skew in data access (a larger value means more skew). The YCSB benchmark uses 2 M records in total, uniformly partitioned among servers.

Testbed. All experiments were conducted on a cluster with 5 servers. Each server has two 10-core Intel Xeon Silver 4210R processors and 64GB DRAM, running CentOS 7.6. Each server is equipped with a ConnectX-5 MCX556A 100Gbps Infiniband NIC connected to a Mellanox SB7890 Infiniband Switch.

6.2 TPC-C Results

Figure 8 and Figure 9 illustrate the aggregated throughput and average latency of evaluated distributed transaction systems under TPC-C/N and TPC-C/NP, respectively. We vary the `local_item_ratio` from 0% to 99% to adjust the ratio of records that `NewOrder` transactions access in remote servers.

Under high contention. Figure 8(a) reveals the performance results in a high contention scenario using TPC-C/N with one warehouse per server. When `local_item_ratio` is small, most of accessed records reside on remote servers, resulting in more network communications. Aurogon improves throughput by 1.55×-3.16× compared with other systems



(a) Under high contention (b) Under medium contention

Figure 9: Throughput and average latency of TPC-C/NP.

when `local_item_ratio` is 0%, which is attributed to Aurogon's capability to reduce aborts caused by accesses from different servers.

The runner-up of performance is DrTM+H. Although it enumerates the RDMA primitive combinations to accelerate network accesses, it suffers throughput degradation as RMWs on hotspots lead to many transaction aborts. Each `NewOrder` contains 11 RMWs on average and the accessed records are nonuniform. Concurrent RMW operations result in a high possibility of transaction aborts in the validation phase.

Besides, DST-TO is a clock-driven transaction system utilizing DST to achieve high scalability. Trading accuracy for scalability in DST decreases the throughput under high contention as inaccurate distributed clocks trigger request reordering. Two reasons lead to the inaccuracy of DST. First, the timestamps that DST uses to update lagging clocks are inaccurate since DST does not consider one-way network delay when obtaining these timestamps. Second, the frequency drift among distributed clocks cannot be corrected in DST.

With increasing `local_item_ratio`, the throughput of Aurogon rises due to the growing local accesses. Note that DST-TO improves the throughput a lot since local accesses use the same clock to get timestamps, easing the reordering. However, Aurogon still outperforms the runner-up (DrTM+H) in throughput by 11% when all transactions are local.

Consequently, Aurogon increases throughput by 1.11×-4.12× compared with other systems, reaching 1.27M transactions per second (0% `local_item_ratio`) and 2.58M (99% `local_item_ratio`, default configuration in standard TPC-C). Meanwhile, Aurogon cuts average latency by up to 86%.

When mixing `NewOrder` and `Payment`[5], the aggregated throughput of Aurogon further expands since `Payment` accesses less records. Figure 9(a) shows that Aurogon outperforms DST-TO by 70%-99% in throughput and reduces average latency by 62%-67%.

Under medium contention. Figure 8(b) plots the systems' performance using TPC-C/N with one warehouse per thread. When each warehouse is bound to a dedicated thread, the contention is alleviated since some variables in `NewOrder` (*e.g.*, `next_o_id`) are never shared among threads.

When `local_item_ratio` is 50%, Aurogon improves throughput by 28%-82% compared to peer systems. The throughput of all systems rises gradually with the growing

[5]We only compare Aurogon with DST-TO under TPC-C/NP and YCSB because other peer systems do not implement these workloads.

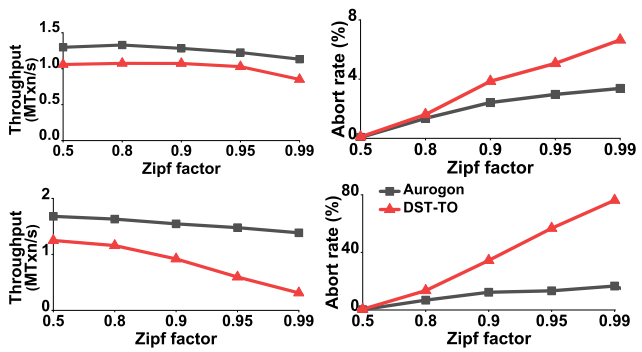


Figure 10: Performance under YCSB (100% `RMW_ratio` in the top two and 50% `RMW_ratio` in the bottom two).

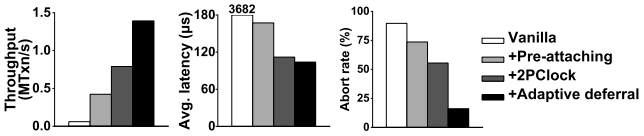


Figure 11: Incremental impact of proposed techniques.

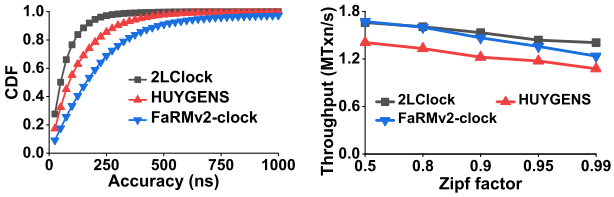
`local_item_ratio` because the contention shrinks. Aurogon shows a moderate slowdown by 1.14× compared to DrTM+H when `local_item_ratio` is 99%. This is because recycling stale versions in Aurogon consumes extra CPU resources when memory consumption rises due to the increase of warehouse count.

Moreover, Figure 9(b) shows that Aurogon outperforms DST-TO by 19%-42% in throughput and reduces average latency by 24%-34% under TPC-C/NP.

6.3 YCSB Results

Figure 10 shows the performance comparison of Aurogon and DST-TO under YCSB with skew factor θ varying from 0.5 to 0.99. When all requests are RMWs, Aurogon increases throughput by 19%-33% compared to DST-TO.

When `RMW_ratio` decreases to 50%, the throughput of both systems rises under low contention (θ is 0.5). Read requests are executed faster than RMWs since reads only require one network communication and never fail. Unfortunately, DST-TO suffers a 4× throughput slowdown when θ rises from 0.5 to 0.99. It is because the throughput improvement increases the running request count in the system. Although conflicts never occur between reads, more reads will make incoming RMWs' execution fail since reads may extend old data version's *rts* larger than RMW's timestamp. So the abort rate of DST-TO reaches up to 76%. On the contrary, Aurogon still maintains a low abort rate (16%) when θ is 0.99, since adaptive request deferral tolerates late arrivals of RMWs. If `RMW_ratio` further decreases, the performance improvement of Aurogon will shrink because Aurogon does not target read-dominant workloads.



(a) CDF of accuracy (b) Throughput in YCSB

Figure 12: (a) CDF of accuracy under CPU interference from foreground transaction system. (b) Aurogon's performance with different clocks.

6.4 Impact of Individual Techniques

To isolate the improvement brought by Aurogon's key techniques, we implement a vanilla version of Aurogon and add three techniques into Aurogon in turn. Here we take the performance under YCSB as an example, setting θ to 0.99 and `RMW_ratio` to 50%.

Figure 11 illustrates that pre-attaching increases the throughput by 7.01× and cuts the abort rate by 16% compared to vanilla Aurogon. Pre-attaching is significant in high contention workload as it not only saves one network communication but also avoids RMW's execution being interrupted.

2LClock further brings a 87% throughput improvement and reduces the average latency by 33%. The decrease of average latency mainly comes from reducing the P99 latency. Specifically, the overhead of retrying transactions after aborts incurs long tail latency since the high contention extends the transaction's execution time and increases the abort rate. Aurogon benefits from 2LClock's high accuracy (41ns) and boosts performance.

Finally, Aurogon improves the throughput by 70% after adding adaptive request deferral. One may wonder whether request deferral is compatible with 2LClock. In fact, 2LClock helps to shorten the required deferral time. To achieve the same abort rate, our test shows that 2LClock requires a deferral time of 24.1µs while DST requires 38.4µs.

6.5 Performance and Impact of 2LClock

We first evaluate the accuracy of 2LClock under foreground CPU interference. Here we compare 2LClock with two state-of-the-art clock synchronization approaches: HUYGENS [16] and FaRMv2-clock [40]. HUYGENS synchronizes the NIC clock of distributed servers with limited CPU involvement. FaRMv2-clock obtains timestamps directly from CPU to synchronize clocks. We remove FaRMv2-clock's guarantee of global increasing monotonicity by skipping its uncertainty wait and support it to provide timestamps directly.

We adopt a common way [16] to test the clock accuracy: two clocks (C_1, C_2) are started on the same NUMA node of one server, and synchronized with the clock C_3 on another server, respectively. The discrepancy of C_1 and C_2 would be 0 since they use the same clock. We take the absolute value of measured discrepancy between C_1 and C_2 as the clock error.

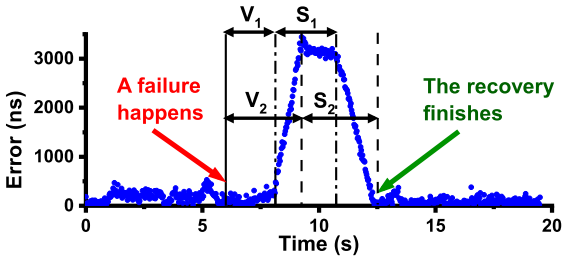


Figure 13: 2LClock failure handling process.

Figure 12(a) shows error CDFs of three clocks under foreground transaction load. The average clock error of 2LClock is 41ns, achieving a cut of 51% and 87% compared to HUYGENS and FaRMv2-clock, respectively. Furthermore, 2LClock reduces the P99 errors by 2.3× and 31× compared to HUYGENS and FaRMv2-clock. Note that FaRMv2-clock encounters a large accuracy fluctuation since CPU timestamps are inaccurate under foreground interference.

We further replace 2LClock with HUYGENS and FaRMv2-clock in Aurogon followed by evaluating these systems under YCSB to show 2LClock's advantage. Figure 12(b) illustrates that 2LClock helps Aurogon to improve the throughput by up to 31% and 14% compared to HUYGENS and FaRMv2-clock, respectively. Aurogon with HUYGENS shows the lowest throughput since obtaining timestamps from HUYGENS takes a longer time (3.3µs) than 2LClock (200ns).

6.6 2LClock Failure Handling

We evaluate how 2LClock handles a server failure of the root clock. Figure 13 plots the clock accuracy in this process. We use the same configuration in §6.5 and simulate the situation where C_1 and C_2 turn to synchronize with a new reference clock C_4 after the crash of C_3.

As illustrated in Figure 13, 2LClock detects the crash of C_3 at time 6s. Then, C_1 and C_2 start their virtual phase (V_1 and V_2), in which they build connections with C_4 and warm up the new clock. Note that they still use the previous measured value of C_3 although C_3 crashes in this phase.

C_1 finishes virtual phase first, taking 2.1s and starts to use the new clock C_4. To smooth the correction value, C_1 begins the smoothing phase S_1, which generates a rising error up to 3.5µs. Then C_2 starts S_2 as well and the error becomes stable since two clocks smooth the correction value at the same speed. As C_1 finished S_1, we can see an error curve plunge before S_2 ends, after which 2LClock turns to normal state. This total process takes 6.4s in our experiment.

6.7 Discussion of Scalability

Here we discuss how Aurogon performs in a large cluster. The rising server count results in more network hops when accessing data. Meanwhile, the network heterogeneity leads to more nonuniform data access latency in the cluster.

Let us examine the three techniques in Aurogon when scaling to a large cluster. First, if distributed clocks aim at achieving high accuracy in a large cluster, the problem is the high CPU usage of parent servers in synchronization topological tree. 2LClock utilizes asynchronous transfers and a combination of two RDMA modes to solve this problem. Second, the more nonuniform data access latency makes the adaptive deferral more effective to tolerate straggling requests. Third, pre-attaching method, saving one RTT for RMWs, is more effective when the network delay rises.

7 Related Work

T/O transaction systems. T/O systems [2,7,15,29,40,47,53–55] lie in two categories. 1) Clock-driven approach: DAST [7] determines timestamps of transactions with a two-phase protocol to anticipate the best execution timing. Cicada [29] separates read and write timestamps during allocation to accelerate read-only transactions. 2) Data-driven approach: TicToc [53] first obtains the data dependencies and determine the transactions' timestamps in commit phase.

Clock synchronization. HUYGENS [16] exploits the network effect to synchronize NIC clocks. Spanner [10], FaRMv2 [40] and Sundial-Clock [28] utilize uncertainty wait to ensure increasing monotonicity of global clocks. The monotonic guarantee is orthogonal to 2LClock since 2LClock can provide it with moderate modifications. Furthermore, 2LClock increases the accuracy of distributed clocks.

RDMA-enabled transaction systems. FaRM [13], DrTM [49], FaSST [24], and DrTM+H [48] exploit RDMA networks to accelerate distributed transaction processing. Their core target is to fully utilize CPU resources to saturate RDMA's high bandwidth. Aurogon proposes a new idea that RDMA can help to reduce transaction aborts.

8 Conclusion

In this work, we propose, implement and evaluate Aurogon, an all-phase reordering-resistant distributed in-memory transaction system. We alleviate request reordering in all phases by three techniques: high-accuracy clock synchronization 2LClock, adaptive request deferral, and pre-attaching dependent requests to data. Aurogon reduces distributed transaction aborts significantly and boosts the performance. The source code of Aurogon is available at `https://github.com/THU-jty/Aurogon.git`.

Acknowledgements

We thank all reviewers for their insightful comments and helpful suggestions, and especially our shepherd Florentina Popovici for her guidance during our camera-ready preparation. We also thank Xingda Wei and Kezhao Huang for helping us run peer systems. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Daniel Abadi. Correctness Anomalies Under Serializable Isolation. http://dbmsmusings.blogspot.com/2019/06/correctness-anomalies-under.html, 2019.

[2] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

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

[4] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 209–223, 2020.

[5] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proceedings of the Fourth International Conference on Weblogs and Social Media (ICWSM'10)*, volume 4, 2010.

[6] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 239–252, 2020.

[7] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*, pages 210–227, 2021.

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, pages 1–17, 2016.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[11] The Transaction Processing Council. TPC Benchmark C. http://www.tpc.org/tpcc/, 2010.

[12] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 401–414, 2014.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 54–70, 2015.

[15] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 173–184. IEEE, 2013.

[16] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 81–94, 2018.

[17] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F Wenisch. Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 337–350, 2019.

[18] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao 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 (SIGMOD'21)*, pages 658–670, 2021.

[19] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment*, 10(5):553–564, 2017.

[20] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[21] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE Standard 1588 (2008).

[22] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 1–16, 2019.

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*, pages 437–450, 2016.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 185–201, 2016.

[25] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

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

[27] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 454–467, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 1171–1186, 2020.

[29] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, pages 21–35, 2017.

[30] Linux. ibv_create_cq_ex. https://man7.org/linux/man-pages/man3/ibv_create_cq_ex.3.html, 2016.

[31] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC'17)*, pages 773–785, 2017.

[32] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[33] Pulkit A Misra, Jeffrey S Chase, Johannes Gehrke, and Alvin R Lebeck. Enabling lightweight transactions with precision time. *ACM SIGARCH Computer Architecture News*, 45(1):779–794, 2017.

[34] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 103–114, 2013.

[35] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 511–524, 2014.

[36] NVIDIA. Time-Stamping Service in RDMA. https://docs.mellanox.com/display/OFEDv502180/Time-Stamping#TimeStamping-EnablingTime-Stamping, 2021.

[37] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, pages 305–319, 2014.

[38] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[39] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD'12)*, pages 61–72, 2012.

[40] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, pages 433–448, 2019.

[41] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD'20)*, pages 1493–1509, 2020.

[42] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. An analysis of concurrency control protocols for in-memory databases with ccbench. *Proceedings of the VLDB Endowment*, 13(13):3531–3544, 2020.

[43] Mellanox Technology. RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[44] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 18–32, 2013.

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

[46] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22. Association for Computing Machinery, 2022.

[47] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for mvcc with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 357–372, 2021.

[48] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 233–251, 2018.

[49] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 87–104, 2015.

[50] Wikipedia. DDR4 SDRAM. https://en.wikipedia.org/wiki/DDR4_SDRAM, 2014.

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

[52] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 633–651, 2021.

[53] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, pages 1629–1642, 2016.

[54] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: harmonizing concurrency control and caching in a distributed oltp database management system. *Proceedings of the VLDB Endowment*, 11(10):1289–1302, 2018.

[55] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6), 2017.

[56] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD'20)*, pages 511–526, 2020.