NOC-NOC: Towards Performance-optimal Distributed Transactions (Technical Report)

ABSTRACT

Substantial research efforts have been devoted to studying the performance optimality problem for distributed database transactions. However, they focus just on optimizing transactional reads, and thus overlook crucial factors, such as the efficiency of writes, which also impact the overall system performance. Motivated by recent studies from Twitter and Facebook showing the prominence of write-heavy workloads in practice, we make a substantial step towards performance-optimal distributed transactions by also aiming to optimize writes, a fundamentally new dimension to this problem. We propose a new design objective and establish impossibility results with respect to the achievable isolation levels. Guided by these results, we present two new transaction algorithms with different isolation guarantees that fulfill this design objective. Our evaluation demonstrates that they outperform the state of the art.

Data Availability. Our prototypes and experimental data are available at [6].

CCS CONCEPTS

• **Information systems** → **Distributed database transactions**; **Database performance evaluation**.

KEYWORDS

Transaction processing, isolation levels, impossibility results

1 INTRODUCTION

Modern web services are layered atop high-performance database systems running in partitioned, geo-distributed environments for system scalability and data availability. Distributed transactions, encapsulating user requests, are an important building block of such database systems. To balance the inherent trade-off between data consistency and system performance [11, 22], there is, therefore, a plethora of isolation levels (or guarantees) for distributed databases. These include not only the gold-standard *serializability* but also new weaker guarantees such as *read atomicity* [7] and *transactional causal consistency* [2, 29], catering for various web applications.

Substantial research efforts [3, 16, 17, 25, 30, 31, 44] have recently been devoted to studying performance optimality for distributed transactions with respect to isolation levels. Two representative results are the recent SNOW [30] and NOCS [31] theorems. Both of these are impossibility results that capture conflicts among desirable properties. SNOW claims that it is impossible to design a **S**trictly serializable system with **N**on-blocking read-only transactions that finish in **O**ne round-trip when transactional **W**rites are present. NOCS proves that a read-only transaction cannot terminate with **O**ne round of **N**on-blocking communication with **C**onstant-size metadata, while achieving **S**trict serializability. In both cases, three of the four properties can be achieved at best, i.e., SNOW-optimality or NOCS-optimality; in particular, under a *weaker* isolation level than strict serializability, NOW and NOC are achievable, respectively.

Although these theorems state that achieving all four desirable performance criteria is impossible, one should aim to achieve three of the four. For example, achieving N, O, and C, would boost the performance of read-only transactions. These theorems are then considered as design objectives and used to assess existing systems for potential performance improvements. For example, the performance of Eiger [29], a causally consistent system, has been significantly improved by optimizing its transactional reads to meet NOC [31].

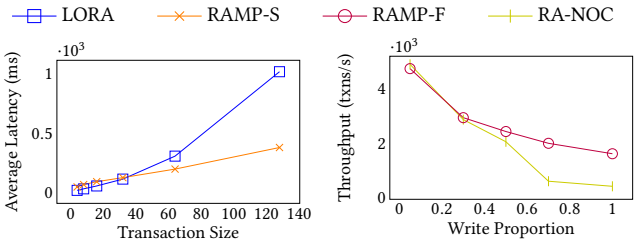


Figure 1: Counterexamples to SNOW (left) and NOCS (right). LORA and RA-NOC are SNOW-optimal and NOCS-optimal transaction algorithms that satisfy read atomicity (RA), respectively. RAMP-F and RAMP-S are the original RAMP-family algorithms designed for RA.

Unfortunately, these performance criteria miss crucial factors that also impact on the overall system performance. As shown in Figure 1, we have found two counterexamples to SNOW and NOCS, respectively. First, LORA [27], a SNOW-optimal algorithm satisfying *read atomicity* (RA), is expected to deliver optimal latency compared to other RA algorithms that do not meet SNOW such as RAMP-S [7]. However, LORA incurs significantly more latency under large-sized transaction workloads. As the metadata size is not considered by SNOW, LORA is designed with the metadata, carried by both its reads and writes, being linear in the transaction size.

Second, RA-NOC, a NOCS-optimal transaction algorithm, exhibits much lower throughput than another original RAMP-family algorithm RAMP-F [7] when writes become heavy. This is because NOCS only optimizes reads, while a NOCS-optimal algorithm like RA-NOC may use expensive locks for processing writes.

Given that such important factors are missed by the state-of-the-art design criteria, a natural question then to ask is "What would an ideal design objective be for distributed transactions?" SNOW and NOCS have already provided a promising baseline, i.e., improving the system performance by optimizing reads. While reads still dominate the workloads in some applications (e.g., Google's AdWords [38]), according to the studies from Twitter and Facebook [5, 46], write-heavy workloads, with 30% or more writes, are

significantly more common in practice than previously thought and expected to rise in prominence.

The NOC-NOC Design Objective. In this paper, we make a substantial step towards performance-optimal distributed transactions by aiming to additionally optimize writes, a fundamentally new dimension to the performance optimality problem.

Optimizing writes can improve overall system performance, especially given that system components are quite often co-designed. In particular, less write latency leads to less average latency of the entire system; moreover, higher throughput for writes gives rise to higher overall throughput. As we will see in our case studies, even under read-heavy workloads, making writes efficient significantly improve the overall system performance over the state of the art.

We propose the NOC-NOC design objective that aims at improving *both latency and throughput* for *both reads and writes*. Along with the aforementioned NOC for reads, we define fine-grained criteria, also called NOC, for writes as: an optimal write transaction shall proceed under **N**on-blocking concurrency control, safely commit in **O**ne round-trip,[1] and carry only **C**onstant-size metadata, while fulfilling the promised isolation guarantee, together with the accompanying transactional reads.

Blocking for a write, due to, e.g., the use of locks or validation in optimistic concurrency control, would increase the latency of the write transaction and decrease the system throughput due to, e.g., CPU underutilization [47]. One-phase commit would incur less write latency and thus overall system latency. Extra metadata (or message payload), e.g., increasing with the number of database partitions, would burden the transmission or processing, thus negatively affecting both latency and throughput.

Moreover, to save system developers' effort trying to achieve impossible objectives, we also identify the upper bound of achievable isolation guarantees for NOC-NOC. We prove that no transaction system that provides parallel snapshot isolation (PSI) [39] (a slightly weaker variant of SI) or any stronger isolation guarantees can achieve all the NOC-NOC criteria. This suggests room for potential improvements to existing transaction algorithms that offer weaker isolation guarantees such as RA and transactional causal consistency (TCC).

There is an inherent trade-off between data freshness and one round-trip reads with atomic visibility [44]. We prove that, with additional one-phase writes, no transaction system that supports *read committed* or any level stronger can make its writes visible to readers from a different session immediately after the prepare phase completes. This, however, suggests that the desired *read-your-write* (RYW) session guarantee [43] is still achievable under NOC-NOC.

Through the lens of NOC-NOC, along with its impossibility results, we examine the state-of-the-art transaction algorithms and identify a gap in the design space. We focus in particular on two recent isolation guarantees, namely RA and TCC, which have attracted the attention of both academia and industry (see Section 2 and Section 3.4).

[1]This one-phase commit is from a client's perspective: when running a two-phase commit protocol, a client can return after the prepare phase and then execute the commit phase asynchronously.

We present a new transaction algorithm for each isolation guarantee that fulfills the NOC-NOC design objective. The key to achieving NOC-NOC common to both algorithms is the incorporation of two novel ideas: dual view and version vector. The dual view extracts global safe snapshots of the database with respect to a certain isolation level and local safe snapshots for reading one's own writes without breaking the isolation guarantee, even when they are only prepared. We also leverage a version vector to encode the dual view, with one element per database partition. In both algorithms, a read request always carry two timestamps, one for each of the dual view, and a server always return one single timestamp to update the views.

Contributions. Overall, we make the following contributions:

(1) At the conceptual level, we address the performance optimality problem by proposing the NOC-NOC design objective that requires optimizing both reads and writes for both latency and throughput. We also established impossibility results with respect to its achievable isolation guarantees.
(2) At the technical level, we present two new transaction algorithms, each for a different isolation guarantee, that fulfill all the NOC-NOC design criteria. We also established their correctness. Along with both designs, we propose a novel combination of two techniques, dual view and version vector, which can be leveraged to optimize transaction algorithms of other isolation guarantees.
(3) At the practical level, we implement and extensively evaluate both algorithms. Our experimental results show that both of them significantly outperform the state of the art and achieve promising data freshness results. This demonstrates NOC-NOC's effectiveness.

2 DISTRIBUTED TRANSACTIONS AND THEIR ISOLATION LEVELS

Modern web applications are built on distributed databases. With data partitioning, very large amounts of data are divided into multiple smaller parts stored across multiple servers (or database partitions) for scalability. User requests are submitted as transactions to the database, typically represented by front ends called clients. Each client then executes the transactions in its own session.

Distributed databases provide various isolation guarantees (or levels), depending on the desired data consistency and availability. Figure 2 shows a spectrum of prevalent isolation guarantees, ranging from weak levels such as *read committed*, through various forms of *snapshot isolation*, to strong guarantees like *strict serializability*. We briefly explain two isolation levels and their variants, which we focus our case studies on.

Read Atomicity (RA). This ensures that either all or none of a transaction's updates are observed by another transaction [7]. It prohibits *fractured reads* anomalies, e.g., in a social network Cara only observes that Ann is a friend of Bob, but Bob is not a friend of Ann. Many industrial databases have integrated read-atomic transactions as an important building block. For example, RAMP-TAO [14] has recently layered RA on Facebook's TAO data store [12] to provide atomically visible and highly available transactions.

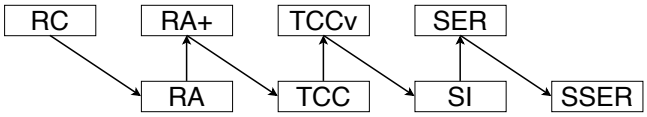


Figure 2: A spectrum of prevalent isolation levels. $A \rightarrow B$ **means that** A **is strictly weaker than** B**. RC: read committed [8]; RA: read atomicity [7]; RA+: RA with read-your-writes [7, 43]; TCC: transactional causal consistency [13]; TCCv: TCC with convergence [2, 29]; SI: snapshot isolation [8], including its weaker variant parallel SI [39]; SER: serializability [36]; SSER: strict serializability.**

RA with Read-your-writes (RA+). On a social networking website, all of a user's requests, submitted as transactions between login and logout, typically form a session. Read-your-writes (RYW) [43] is commonly provided by many production databases such as Facebook's TAO [12] and MongoDB [33]. It guarantees that the effects of all writes performed in a client session are visible to its subsequent reads. For example, after Ann tweets and refreshes, she should be able to see her own tweet. All RA systems we are aware of actually provide RA+, even including the RAMP-family algorithms [7] for which RA is originally proposed.

Transactional Causal Consistency (TCC). This combines two properties: RA and causal consistency (CC) [1, 37]. The latter guarantees that two transactions that are causally related must appear to all client sessions in the same causal order. It prevents causality violations, such as Ann observing Bob's comment on Cara's post without seeing the post itself.

TCC with convergence (TCCv). With TCC, transactions that are *not* causally related may be observed in different orders by different sessions. This may result in permanent divergence of these sessions' views under conflicting concurrent updates. TCCv's *convergence* property prohibits such phenomena by requiring these views to converge to the same state [2, 28]. For example, without convergence, Ann's and Bob's updates on the meeting place in a trip planner may end up differently to Cara and Dan, while, with TCCv, they would see the same final place. To the best of our knowledge, all production database systems claiming to support causal consistency provide convergence, e.g., MongoDB [33], Cosmos DB [32], and ElectricSQL [21].

3 THE NOC-NOC DESIGN OBJECTIVE

In this section we present the NOC-NOC design objective and establish impossibility results with respect to its achievable isolation guarantees. Examining the state-of-the-art RA(+) and TCC(v) transaction algorithms in the light of NOC-NOC, we identify a large gap in the design space.

3.1 System Model and Assumptions

System Model. We follow the system model as in SNOW [30] and NOCS [31]. A distributed (transaction) system consists of a set of client processes and server processes that communicate by sending and receiving messages. Processes behave deterministically: in each atomic step, they receive messages (if any), perform deterministic local computations, and send messages (if any) to other processes. The network is asynchronous with no global clock. Processes run at arbitrary speeds and messages can be arbitrarily delayed.

Definitions. A transaction starts when the client sends the requests to associated servers and ends when the client receives all necessary responses. A transaction T_1 *happens before* another transaction T_2 if T_1 ends before T_2 starts. Two transactions are *concurrent* if neither happens before the other, i.e., their lifetimes overlap. Two transactions *conflict* with each other if they access the same key and at least one of them writes to this key. Two transactions *write-conflict* with each other if they both write to the same object.

Given its wide adoption in practice, we employ the two-phase commit (2PC) protocol as the atomic commitment protocol for committing transactions. Each 2PC instance runs the prepare phase first, followed by the commit phase. We call a 2PC variant *one-phase commit* (or *one-phase writes*) if the 2PC coordinator (quite often the client) returns after completing the first phase, where all writes are fully prepared on the associated servers. The coordinator then executes the second phase asynchronously, which races with its subsequent transactions.

A transaction system satisfies the *one-phase global visibility* property if every transaction is visible to all transactions that start after this transaction completes the first phase of 2PC.

Assumptions. We assume that the system, the processes, and the network are failure-free. We also assume that every message will be delivered eventually. Our impossibility results also apply to systems with faults.

We assume one-shot transactions [24] which are common in practice. A one-shot transaction knows the database partitions that store the keys accessed by its reads/writes a priori. For one-shot transactions, we can send read/write requests to database partitions in parallel, as there are no key dependencies. Our impossibility results for one-shot transactions also apply to multi-shot transactions.

Moreover, we assume that clients issue one-shot read-only and write-only transactions to the database. Our transaction algorithms designed in this paper, along with the competing algorithms, can be naturally extended to support general read-write transactions [7]. Moreover, our impossibility results for read-only and write-only transactions also apply to general read-write transactions.

Finally, we focus on the single-datacenter setting that supports data partitioning. Our impossibility results also apply to the multi-datacenter with data replication.

3.2 Defining NOC-NOC

3.2.1 Performance Criteria for Reads. NOC-NOC adopts the performance criteria for read-only transactions defined in NOCS [31].

Non-blocking Reads (N_R). Read-only transactions are considered to be non-blocking if transactional reads are processed by servers without waiting for any external events, e.g., a lock to release, a message to receive, or a timer to expire. Recall that we assume failure-free servers; otherwise, read requests to faulty servers would be stalled. N_R is desirable as blocking reads naturally increase the latency of read-only transactions. Moreover, system throughput can also be reduced due to, e.g., CPU underutilization [47].

One Round-trip Reads (O_R). This property requires a read-only transaction to finish in one round-trip. Specifically, a client sends a parallel round of transactional reads to all the database partitions

involved, and each partition sends at most one response back. O_R excludes read-only transaction algorithms that may abort, since retries are essentially extra rounds of communication. Read-only transactions that rely on "off-path" messages to ensure their correctness are also disallowed. For example, COPS-SNOW [30] uses extra messages during writes to help transactional reads find consistent snapshots, which are, however, not a necessity for processing writes. The O_R criterion is desirable as otherwise system latency and throughput would be negatively affected; in particular, extra off-path messages would burden servers for processing writes.

Constant-size Metadata for Reads (C_R). Metadata, such as transaction identifiers and timestamps, are typically required by servers to find the correct versions of data for read requests. The metadata associated with each transactional read are of constant size if they do not increase with the transaction size, the number of database partitions, or the number of conflicting operations. Transmitting/processing extra metadata naturally increases latency and decreases system throughput.

3.2.2 Performance Criteria for Writes. While missing C_R, SNOW defines a "write transactions" property for reads (in addition to N_R and O_R), requiring a read-only transaction algorithm to be able to coexist with conflicting transactional writes (W). NOC-NOC further turns this compatibility criterion into fine-grained performance criteria for transactional writes. The key insight is that reducing the communication complexity for writes can improve overall system performance, especially given that system components, e.g., read and write transaction algorithms, are quite often co-designed.

Non-blocking Writes (N_W). This criterion requires a non-blocking concurrency control mechanism, which excludes the use of locks and any optimistic concurrency control mechanisms that may block during validation. An example negative effect of using locks has been shown in Figure 1, where RA-NOC incurs significant overhead due to blocked writes. Hence, similar to N_R, N_W is desirable not only for less write latency but also for overall throughput improvement.

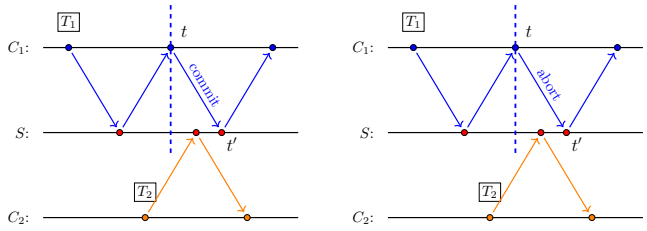
One-phase Writes (O_W). After finishing the prepare phase of 2PC, the client should commit the transaction. Subsequently, it issues the next transaction, if any, and runs the commit phase asynchronously. Achieving O_W decreases write latency, as a transaction can start immediately after the prepare phase of the previous write transaction, rather than wait for the commit phase to complete. Moreover, committing a transaction in one phase enables earlier visibility of the versions written. This improves data freshness that may be traded off for O_R [44].

Constant-size Metadata for Writes (C_W). Similar to C_R, this criterion requires each request or response of a write transaction to carry constant-size metadata. Otherwise, extra metadata would downgrade overall system performance. As we have observed in Figure 1, LORA incurs significantly more average latency under large-sized transaction workload as the metadata carried by its writes (and reads) are linear in the transaction size.

3.3 Impossibility Results for NOC-NOC

We present two impossibility results for NOC-NOC.

Theorem 3.1. *No transaction algorithms that support PSI or SI can achieve all six NOC-NOC criteria.*



(a) T_1 commits **(b) T_1 aborts**

Figure 3: Two indistinguishable scenarios for the client C_2.

Ardekani et al. [4] have proved that both PSI and SI are inherently non-scalable. The key to our proof of Theorem 3.1 is to show that NOC-NOC implies scalability. Hence, both PSI and SI would be incompatible with NOC-NOC.

Proof. It has been proved that no transaction algorithms that support PSI or SI can simultaneously achieve the GPR (Genuine Partial Replication), OFU (Obstruction-free Updates), and WFQ (Wait-free Queries) properties [4]. We now show that the NOC-NOC criteria imply these three properties.

First, GPR requires that each transaction contacts only the servers that store the keys accessed by its reads/writes. Communication with other servers is a special kind of external events, which are disallowed by both N_R and N_W. Second, OFU requires every write transaction to eventually terminate in a failure-free system and commit if it does not write-conflict with another concurrent transaction. This is implied by O_W. Third, WFQ requires a read-only transaction to never wait for another transaction and eventually commit. This is implied by N_R. □

Theorem 3.1 implies that isolation levels weaker than PSI/SI, e.g., RC, RA(+), and TCC(v), are achievable under NOC-NOC.

Theorem 3.2. *No transaction algorithms that support RC can achieve all six NOC-NOC criteria with global visibility.*

This impossibility theorem holds as a read-only transaction, occurring after a write-only transaction T completes its prepare phase but before T finishes the commit phase, is unable to conclusively determine whether it is safe to read the value written by T.

Proof. We prove the theorem using an indistinguishability argument. Consider a scenario where the client C_1 issues a write-only transaction T_1. Suppose that one of the writes in T_1 accesses the key K stored on the server S. The client C_1 first performs the prepare phase, where it sends the prepare message to the server S and receives the vote at time t. Then, C_1 proceeds with the commit phase, where S receives the notification from C_1 at time t'. Figure 3 depicts this scenario, where, for simplicity, we omit the other servers involved in the write-only transaction.

Now consider another client C_2 which issues a read-only transaction T_2. Without loss of generality, suppose that T_2 only reads from the key K. The client C_2 issues a read request to S, which arrives in between t and t', i.e., when S is waiting for C_1's notification.

Suppose, by contradiction, that the system satisfies O_W with global visibility. Hence, T_1 is globally visible after t, which implies that C_2 can then safely read the value written by T_1.

Figure 3 shows two scenarios, where T_1 commits and aborts (because, e.g., another server failed to prepare the write) after t', respectively. However, these two scenarios are indistinguishable to C_2, and thus C_2 would behave the same. If T_2 reads the value written by T_1, then the execution in Figure 3b violates RC. Otherwise, the scenario in Figure 3a violates O_R. □

Despite being incompatible with global visibility under RC (and any isolation level stronger), NOC-NOC still allows a client's writes to be visible to its own subsequent reads. As we will see in our case studies, with the dual-view design and co-location of client sessions in practice, promising data freshness results can still be achieved.

3.4 Examining Existing Transaction Algorithms

The aforementioned two impossibility results suggest that it is possible to design a NOC-NOC-optimal transaction algorithm with "local visibility", which provides an isolation guarantee weaker than PSI/SI. By examining the state-of-the-art transaction algorithms, we identify a significant gap in the design space.

We focus on two isolation levels and their variants, namely RA(+) and TCC(v). The reason is fourfold. First, all these guarantees are weaker than PSI/SI which fit the possibilities of NOC-NOC; in particular, TCCv is, to the best of our knowledge, the strongest isolation level weaker than SI and its variants. Second, they are prevalent and have attracted the attention of both academia and industry; in fact, they have already been adopted by production database systems [14, 21, 32, 33]. Third, they present various underlying properties: atomic visibility for RA, RYW in addition for RA+, causal consistency for TCC, and convergence in addition for TCCv. Fourth, the upper bound of achievable isolation levels for NOCS-optimal read-only transactions in the presence of transactional writes remains an open research question–TCC (without convergence) is conjectured as the upper bound [31].

Table 1 shows the comparison results, where we focus on two state-of-the-art families of transaction algorithms, namely the RAMP-family [7, 27] and the Eiger-family [29, 31] algorithms. See Section 9 for a discussion on other algorithms.

RA(+) Algorithms. None of the existing RAMP-family algorithms achieves NOC-NOC. In particular, all the original RAMP-family algorithms (i.e., RAMP-F/S/H), as well as the conjectured optimization RAMP-OPW [7], fail to provide O_R and C_R, which do not even satisfy SNOW and NOCS. Moreover, only RAMP-OPW achieves O_W at the cost of loosing RYW. Only RAMP-S attempts at constant-size metadata, yet only for writes. LORA [27] is SNOW-optimal but not NOC-optimal as its metadata in both reads and writes are linear in the transaction size. RA-NOC is NOCS-optimal; however, its writes are blocking and do not commit in one phase. Our new design, RA-NOC2 fully meets all six NOC-NOC performance criteria and provides RA+ (see Section 4).

TCC(v) Algorithms. Eiger [29] supports TCCv but fails to achieve O_R and O_W. Eiger-PORT [31] optimizes Eiger with O_R by sacrificing the convergence property. We present in Section 6 a NOC-NOC-optimal transaction algorithm, called Eiger-NOC2, which guarantees TCCv.

Table 1: Comparison of the state-of-the-art RAMP-family and Eiger-family transaction algorithms. RAMP-OPW does not satisfy the RYW session guarantee. Eiger-PORT provides a weaker isolation guarantee without convergence.

System	Isolation	N_R	O_R	C_R	N_W	O_W	C_W	Optimal
RAMP-F	RA+	✓	≤ 2	×	✓	×	×	None
RAMP-S	RA+	✓	2	×	✓	×	✓	None
RAMP-H	RA+	✓	≤ 2	×	✓	×	×	None
RAMP-OPW	RA	✓	≤ 2	×	✓	✓	×	None
LORA	RA+	✓	✓	×	✓	✓	×	SNOW
RA-NOC	RA+	✓	✓	✓	×	×	✓	SNOW, NOCS
RA-NOC2	RA+	✓	✓	✓	✓	✓	✓	All
Eiger	TCCv	✓	≤ 3	✓	✓	×	✓	None
Eiger-PORT	TCC	✓	✓	✓	✓	×	✓	SNOW, NOCS
Eiger-NOC2	TCCv	✓	✓	✓	✓	✓	✓	All

4 THE RA-NOC2 ALGORITHM

Following NOC-NOC's performance criteria, we now present our new design, RA-NOC2, that provides NOC-NOC-optimal distributed transactions with the RA+ isolation guarantee.

4.1 Overview

The major challenge for achieving NOC-NOC-optimal RA+ transactions is to satisfy *all eight properties together*, including six criteria of NOC-NOC and two consistency properties (i.e., RA and RYW). Compared to the state of the art, the most challenging part is to achieve O and C for both reads and writes (each criterion is missed by at least four of the six existing algorithms) without trading off the RYW session guarantee (RAMP-OPW sacrifices this property for optimizing writes).

To provide both O_R and atomic visibility (which RA+ subsumes), RA-NOC2 may incur data staleness [44], i.e., it cannot be guaranteed to return the most recently committed data. Given that users prefer recent data [23, 44], another challenge is, despite this theoretical result, for RA-NOC2 to achieve satisfactory data freshness in practice.

RA-NOC2 addresses these challenges by incorporating two key ideas: *version vector* and *dual view*.

Version Vector. Inspired by [31], we leverage version vectors to encode client views of database states. Each client maintains a version vector, with one element per database partition. Each element stores the *latest safe time* (LST) for the corresponding partition, which is defined as the minimum of the timestamps of prepared-only and committed transactions, i.e., the most recent snapshot of the database from the partition's perspective where all writes are committed and safe to return.

Unlike the RAMP-family algorithms and LORA, which rely on message metadata of linear size to compute the correct version to return, RA-NOC2 has only two timestamps for its dual view (see below) in a read request (C_R), and reads from either of these two snapshots. Moreover, the client-side version vector is updated upon receiving a response from the server, which always carries the single timestamp LST (C_W).

Dual View. Each client maintains dual views and reads from one of them (O_R): (i) a *global safe view* (GSV) for extracting RA-safe snapshots of the database, where each snapshot is the most recent in the version order with the incorporated versions all committed

on the servers (i.e., the minimum of LSTs across the associated partitions); and (ii) a *local safe view* (LSV) for computing RYW-consistent snapshots, when the client must fetch any potential prepared-only versions (due to O_W in RA-NOC2) to read its own writes. To incorporate these versions into the LSV, the client keeps track of its most recently prepared, possibly uncommitted, version of each key locally.

Note that an LSV is quite often beyond a GSV, e.g., when the commit phase of a write transaction races with subsequent transactional reads from the same client. Nonetheless, the associated prepared-only writes are guaranteed to be present on the server as, otherwise, the client would not have "found" them from its LSV, and safe as, otherwise, the client would have read within its GSV.

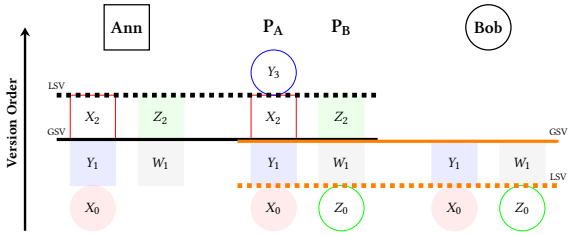


Figure 4: An illustrative example of the dual view. Colors represent different keys, and shapes different clients (squares for Ann and circles for Bob). Prepared-only and committed versions are transparent and filled, respectively. Black and orange lines refer to Ann's and Bob's dual view, respectively.

Example 4.1. Figure 4 illustrates the dual views with an example of two clients, Ann and Bob, and two database partitions, each storing two keys. We assume that the keys X and Z, resp. Y and W, are always siblings in a write transaction.[2] First, as a GSV is the minimum of the LSTs across all partitions, Ann's GSV is at 1, even though her actual view of partition B is at 2 by the version Z_2. Note that her GSV cannot be advanced by X_2 on partition A since the version is only prepared and thus not safe from A's perspective.

Moreover, a client's LSV is advanced by any write of its own as long as the associated write transaction is at least fully prepared. For instance, Bob's LSV is at 0 as the version Z_0 has been prepared and its sibling version X_0 is already committed. Similarly, Ann's LSV is advanced by the write transaction that writes X_2 (prepared only) and Z_2 (committed). However, since Y_3's sibling version W_3 has not been prepared on partition B, Bob cannot advance his LSV. Note that an LSV can only include one's own prepared-only versions. Hence, Ann cannot see Z_0. Note also that an LSV can be either ahead of a GSV like in Ann's case, behind a GSV as for Bob, or even the same as a GSV.

To reduce data staleness, RA-NOC2 tightly couples the dual view and version vector by design. Instead of naively returning stale versions that satisfy RA+, a client constantly updates (i) its LSV, upon completing a write transaction's prepare phase, to include the latest writes of its own, and (ii) its GSV of latest committed versions across partitions, during the processing of both reads and writes, to always fetch the most recent versions up to the safe frontier of RA+. Moreover, via co-location of sessions (or threads) in practice, clients can furthermore keep each other's dual view fresh by sharing.

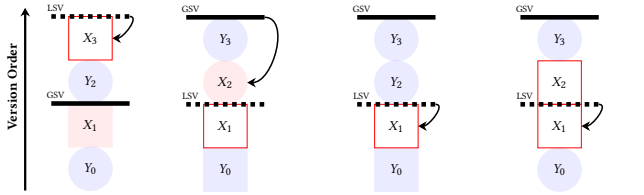
[2]For a key (or version), its sibling keys (or versions) are those keys (or versions) written in the same transaction.

4.2 Algorithm

We leverage the dual view and version vector to design RA-NOC2's read-only and write-only transaction algorithms, together with a multi-versioned database. Its pseudocode is given in Algorithm 1.

Client Dual View. A client stores a version vector *last* (line 2), associating one timestamp to each database partition. This allows the client to then compute its GSV, i.e., the latest database snapshot that it knows to be read-atomic. Moreover, the client also maintains the *prep* data structure (line 3), containing all fully prepared, possibly uncommitted, versions' timestamps, along with their sibling keys. These versions are used to compute its LSV, i.e., the latest safe version to date for a given key from its perspective.

Server-side Data Structures. Each partition stores part of a multi-versioned database, where a version maps each key to its value and timestamp (line 26). It also holds the highest committed timestamp, called *latest* (line 27), and a set *pending* of prepared, yet uncommitted versions (line 28). The latest safe time (LST) is computed by taking the minimum of *pending*, if it is nonempty, or otherwise *latest* (lines 45–48).



(a) LSV beyond GSV: return at LSV **(b) LSV within GSV: return in between** **(c) LSV within GSV: return at LSV** **(d) LSV within GSV: skipping own write**

Figure 5: Reading from different views. We use the same setup as in Figure 4. For simplicity, we use only one partition to illustrate different scenarios.

Transactional Reads. A client constructs two views when reading from a key. The GSV is encapsulated by a timestamp ts_s, the minimum of *last* across all partitions involved in the read-only transaction (line 17). The LSV is unique for each key k and corresponds to the highest timestamp in *prep* where the associated version has k as its sibling key (line 19).

If the LSV is *beyond* the GSV, the client reads from the LSV of its own write by requesting from the partition the version matched by ts_p (lines 20–21). Figure 5a shows an example where Ann is reading from key X. Since her LSV is beyond GSV, her own write X_3 is returned, although it is prepared only.

If the LSV is *within* the GSV, the client reads from its GSV by sending both ts_c and ts_p (lines 22–23). The partition then returns the most recent committed version in between these two timestamps (lines 41–43). This is the case in Figure 5b where the committed version X_2 between LSV and GSV is returned to Ann.

If no version can be found (as the client's GSV was advanced by other keys on this partition), the exact version at ts_p is sent back instead (line 44). For example, no version of X exists between

Algorithm 1 The RA-NOC2 Algorithm

```
1:  ___ Client-side Data Structures & Methods ___
2:  last[svr]: last committed timestamp on server svr
3:  prep[ts]: write set of prepared version ts

4:  procedure PREPARE_ALL(W : set of ⟨key, value⟩)
5:      ts ← generate new timestamp
6:      parallel-for ⟨k, v⟩ ∈ W do
7:          ts_svr ← PREPARE(⟨k, v, ts⟩)
8:          last[svr] ← max(ts_svr, last[svr])
9:      prep.add(ts, {w.key | w ∈ W})
10:     return

11: procedure COMMIT_ALL(V : set of versions)
12:     parallel-for ver ∈ V do
13:         ts_svr ← COMMIT(ver.ts)
14:         last[svr] ← max(ts_svr, ver.ts, last[svr])
15:     return

16: procedure GET_ALL(K : set of keys)
17:     ts_c ← min({last[svr] | svr storing k ∧ k ∈ K})
18:     parallel-for k ∈ K do
19:         ts_p ← max({ts | k ∈ prep[ts]})
20:         if ts_c < ts_p then
21:             rs[k], last[svr] ← GET(k, ts_p, null)
22:         if ts_c ≥ ts_p then
23:             rs[k], last[svr] ← GET(k, ts_c, ts_p)
24:     return rs

25: ___ Server-side Data Structures & Methods ___
26: vers: multi-versions ⟨key, value, timestamp⟩
27: latest: highest committed timestamp
28: pending: timestamps for uncommitted write txns

29: procedure PREPARE(ver : version)
30:     vers.add(ver)
31:     pending.add(ver.ts)
32:     return LST(pending,latest)

33: procedure COMMIT(ts_c : timestamp)
34:     latest ← max(ts_c, latest)
35:     pending.remove(ts_c)
36:     return LST(pending,latest)

37: procedure GET(k, ts_req, ts_p)
38:     lst ← LST(pending,latest)
39:     if ts_p = null then
40:         return vers[k].at(ts_req), lst
41:     for ver ∈ vers[k].between(ts_p, ts_req) do
42:         if ver.ts ∉ pending then
43:             return ver.value, lst
44:     return vers[k].at(ts_p), lst

45: procedure LST(pending, latest)
46:     if pending is empty then
47:         return latest
48:     return min(pending)
```

the two views in Figure 5c; the version X_1 situated at the LSV is therefore returned.

Note that we exclude any prepared version by the same client in between the aforementioned two timestamps, as the associated write transaction have not been fully prepared yet (otherwise, the client would have sent this version's prepared timestamp that is higher than ts_p). Figure 5d depicts such a case: Ann's own write X_2 sits above the LSV, but it is unsafe to return; instead, the read skips over it and fetches X_1. In both cases, the partition returns the LST as well, which is then used by the client to advance its GSV.

Transactional Writes. The traditional 2PC protocol [26] underlies RA-NOC2's write-only transactions. A client first generates a new timestamp for the versions that it intends to prepare on the associated partitions (in parallel). Upon receiving the prepared version, the partition adds it to the local database (line 30), as well as the *pending* list (line 31). After completing the prepare phase, the client incorporates the new prepared versions into its LSV so that any subsequent reads are guaranteed to see its own writes (line 9).

Upon finishing the prepare phase, the client asynchronously commits the transaction (in parallel) with the version's (commit) timestamp. The partition then updates *latest* and *pending* accordingly. Upon receiving the ack, the client advances its GSV (for freshness) by incorporating the timestamp of the version just committed and the partition's LST (line 14).

4.3 Correctness

RA-NOC2 naturally meets all NOC-NOC criteria by design. In particular, by encoding a client dual view into two timestamps in a read request (C_R), this enables a server to extract the precise version to return without any intervention (N_R), thus completing the read in one round-trip (O_R). RA-NOC2 meets O_W by incorporating a client's own writes into its LSV for future reads, thus enabling safe asynchronous commits. It also achieves C_W, since the only metadata per write request, is the client's identification, and only the server's latest safe timestamp is added to a response.

We prove that RA-NOC2 satisfies both RA and RYW. Intuitively, as a client's GSV captures the latest safe snapshot across all partitions which includes committed versions only, no fractured reads would ever happen, thus guaranteeing RA. Even though we may jump beyond the GSV for the client's own writes (RYW), the LSV then ensures that any versions fetched are at least fully prepared, thus no fractured reads either. The complete proof is given in Appendix A.

5 RA-NOC2 EVALUATION

In this section we extensively compare RA-NOC2 to the state-of-the-art RA(+) systems, demonstrating its throughput and latency improvement. We also show RA-NOC2's competitive data freshness.

5.1 Competitors

We consider five *strong* competitors (see also Table 1):

- RAMP-F and RAMP-S [7], which are the two original, yet state-of-the-art, read-atomic algorithms;
- the RAMP-OPW design [7], which optimizes RAMP with one-phase writes while sacrificing the read-your-writes session property (thus providing a weaker isolation guarantee than RA-NOC2);
- LORA [27], which is a SNOW-optimal read atomic algorithm, missing only C_R and C_W; and
- the NOCS-optimal algorithm RA-NOC, which we have designed following the NOCS design objective.[3]

We do not consider RAMP-H [7] since, as shown in [7, 27], its performance lies in between that of RAMP-F and RAMP-S.

5.2 Implementation, Setup, and Workloads

Implementation. For a fair comparison, we implement our algorithm RA-NOC2 (around 1000 LOC in Java), along with LORA, RAMP-OPW, and RA-NOC, atop a multi-versioned database in the original RAMP codebase [7]. For RAMP-OPW, we modify the RAMP-F implementation to commit write-only transactions after the prepare phase and complete the commit phase asynchronously. Keys are assigned to a database partition by a distributed hash table. Each client contacts the front-end of a partition that executes the requested transactions, i.e., the front-end plays the role of the client in Algorithm 1.

[3]The pseudocode of RA-NOC is given in C.

Experimental Setup. As RAMP is a concurrency control protocol, its codebase does not support data replication. We therefore run our experiments on a single CloudLab cluster [20], each machine with two 10-Core, 3.4 GHz, Xeon E5-2640 v4 CPUs (xl170 node from the Utah cluster) and a 10Gbps network interface. By default, we use 5 client machines and partition the entire database across 5 servers. We run five 60-second trials for each data point and plot the average.

Workloads. We use the same YCSB [15] benchmark in RAMP [7] to generate transactional workloads where multiple operations are grouped into read-only or write-only transactions. By default, we choose the transaction size of 16 operations and the value size of 1 byte, in order to fully expose the impact of metadata size on system performance. We match the key-access distribution (Zipfian with the skewing factor of 0.99), database size (1 million keys), and read/write ratio (95% read-heavy workloads) in RAMP. We also run 5000 YCSB client threads distributed over 5 client machines.

5.3 Evaluation

Summary of Results. RA-NOC2 shows significant performance improvement under various workloads over the competitors, which do not fulfill all the NOC-NOC criteria. In particular, under large-sized transaction workloads, RA-NOC2 achieves 100%–495% improvement in throughput, along with 40%–84% reduction in latency; it outperforms the competitors under full-spectrum workloads with varying read/write ratios, exhibiting up to 73% higher throughput and incurring 43% less latency. RA-NOC2 also demonstrates its scalability with increasing numbers of servers and clients. All these performance achievements are attributed to RA-NOC2's adherence to NOC-NOC for both reads and writes. Given its one round-trip reads, RA-NOC2 is expected theoretically to trade off data freshness [44]. Nonetheless, its data freshness still remains competitive with that of existing algorithms, with over 96% up-to-date reads.

We defer the plots for the comparison between RA-NOC2 and RA-NOC to the appendix D, where the conclusions we draw from this section (e.g., on throughput and latency improvements) also apply.

Latency Improvement. RA-NOC2 performs well with large-sized workloads. Figure 6a and Figure 6b show that, with an increase in the transaction size, RA-NOC2 significantly outperforms the competitors in both read and write latency. In particular, even compared to LORA (resp. RAMP-OPW), which also has one round-trip reads (resp. writes), it achieves up to 84% (resp. 40%) reduction in read (resp. write) latency. Overall, RA-NOC2's latency improvement owes to O_R, which the RAMP-family algorithms do not satisfy, and to O_W, which all RA+ algorithms except LORA fail to provide. Another two contributors are C_R and C_W that naturally improve RA-NOC2's latency when transferring and handling larger metadata.[4] We also measure 99th percentile latency with varying read/write proportions. As shown in Figure 6c, one-phase writes, along with their asynchronous commits, do not lead to more overhead, and RA-NOC2 constantly surpasses the competing algorithms under full-spectrum workloads, with 23%–43% latency reduction.

[4]LORA's latency jumps in Figure 6a and Figure 6b result also from its expensive local computation which is proportional to the transaction size.

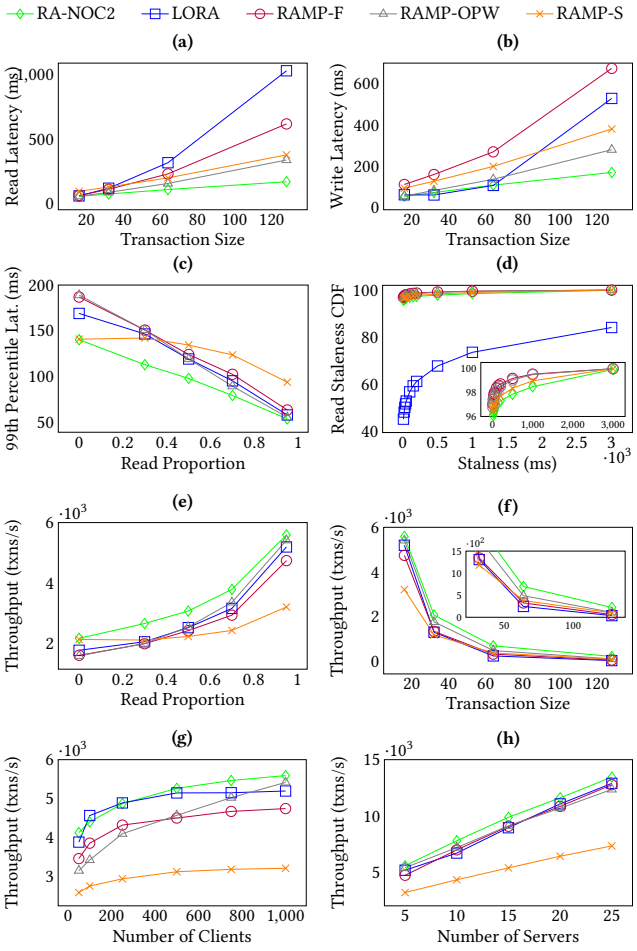


Figure 6: Performance and freshness comparisons. RAMP-OPW provides RA, which is strictly weaker than RA+ provided by the other algorithms including RA-NOC2.

Throughput Improvement. Overall, RA-NOC2 shows significantly higher performance than the state-of-the-art algorithms and owes this improvement to its adherence to C_R and C_W. Figure 6e shows that RA-NOC2 constantly outperforms the competitors as we vary the read/write ratio: it achieves up to 29% throughput improvement over the SNOW-optimal algorithm LORA; it also surpasses RAMP-S by 73%, even under 95% read-heavy workloads for which the RAMP-family algorithms are specifically designed. Moreover, RA-NOC2 exhibits significantly higher throughput under large-sized workloads, as shown in Figure 6f. In particular, with 128 operations per transaction, it achieves 495% improvement in throughput over LORA and at least 100% improvement over the optimized RAMP algorithm OPW.

Finally, we explore the scalability of RA-NOC2. Figure 6g depicts that RA-NOC2 scales well as the number of client sessions increases (while we keep client machines constant), with noticeably higher throughput. Additionally, along with an increasing number of partitions, we scale up the number of clients while keeping sessions per

client machine constant, in order to fully saturate the system. Figure 6h shows that, compared to the competitors, RA-NOC2 scales up better and obtains constantly higher throughput.

Data Freshness. We measure data freshness in terms of staleness defined as the time difference (in milliseconds) between the version read and the latest commit of the associated key. Overall, RA-NOC2 achieves competitive data freshness against the state-of-the-art algorithms, as shown in Figure 6d. In particular, despite the inherent loss of data freshness due to O_R [44], RA-NOC2 still reports 96% up-to-date reads, while LORA, also with one round-trip reads, only manages 40%. This owes to the design choice of RA-NOC2 where a client dual view is always advanced up to the most recent, yet safe frontier of RA+, and the implementation choice of sharing views among co-located clients.

6 THE EIGER-NOC2 ALGORITHM

Guided by the NOC-NOC design objective, we improve an existing NOCS-optimal algorithm called Eiger-PORT [31], that is, to the best of our knowledge, the most performant TCC algorithm to date.

The reason for choosing Eiger-PORT as a base algorithm to demonstrate NOC-NOC is threefold. First, Eiger-PORT already provides optimal read-only transactions (NOC for reads), plus highly efficient transactional writes (N_W and C_W). Moreover, it already satisfies a sufficiently strong isolation guarantee, TCC. Finally, the upper bound of achievable isolation levels for NOCS-optimal read-only transactions in the presence of transactional writes remains an open research question—TCC *without convergence* is conjectured as the upper bound [31]. All together, this renders Eiger-PORT a strong baseline and any improvement to it non-trivial and challenging.

6.1 Eiger-PORT in a Nutshell

The Eiger-PORT design is guided by NOCS, thus satisfying NOC for reads. The core idea is to capture a TCC-consistent snapshot of the database per client request (O_R), which is computed over the client-side version stamps (similar to version vectors; see Section 4.1). The value of a version stamp represents the safe time (similar to LST in RA-NOC2) on a partition, and the minimum of such values across partitions like a GSV is selected as the snapshot embedded in a read (C_R). Moreover, version stamps are extracted from the Lamport clocks used by the partitions to guarantee a causal ordering. Upon receiving a requested snapshot, the partition checks for the existence of a committed version by the client (to satisfy read-your-writes) that is strictly beyond the snapshot; given such a version, the client reorders it before the snapshot in the version order. Altervatively, if such a version does not exist, the partition performs a recursive backward search through the versions within the snapshot, finding a version that ensures read atomicity (RA). As a result, clients may observe versions in different orders; this is allowed by TCC, yet breaks convergence.

Write transactions proceed by following a variant of the traditional 2PC that always commits [29].[5] Between the two phases, the coordinator ensures that each cohort (or partition) commits with the same version timestamp by synchronizing the Lamport clocks across the cohorts up to the maximum of all the proposed timestamps. This enforces a consistent snapshot of the database by the write transaction (as we will see in Figure 7). The coordinator, as well as each cohort, returns its local safe time for updating the client's versionstamp.

6.2 Overview of Eiger-NOC2

Eiger-PORT does not provide O_W and satisfies only TCC without the convergence guarantee. O_W is valuable given that write-heavy workloads (with 30% or even more writes) are significantly more common in practice than previously thought [5, 46]. Moreover, system components are quite often co-designed, indicating that optimizing writes, even just their latency, would improve the overall system performance (as we have observed in RA-NOC2 and will see in Eiger-NOC2). Finally, convergence is the *de facto* guarantee by causally consistent systems in practice [21, 32, 33, 35].

Eiger-NOC2 leverages the *dual view* to improve Eiger-PORT with both O_W and convergence. The dual view computes two separate snapshots of a database, i.e., the local safe view (LSV) and global safe view (GSV), which underlies the fulfillment of O_W without sacrificing RYW (as we have seen in RA-NOC2). However, its integration into Eiger-PORT is challenging as we must guarantee (i) the causal ordering, a strictly stronger consistency requirement than RYW, for both views, especially LSVs that may include prepared-only versions, and (ii) convergence without losing RA; the authors [31] conjecture that convergence would be incompatible with RA in the presence of causally consistent reads that satisfy NOC.

Regarding challenge (i), like in Eiger-PORT, GSVs constructed over Lamport clocks across partitions can precisely capture the causality among committed versions. However, to further ensure a causal order of LSVs, we collect on the client side during its write transaction all the prepare timestamps proposed by the partitions (also based on their Lamport clocks), and advance the client's LSV by mimicking how the coordinator would commit the transaction. Therefore, we are able to correctly order the prepared versions for the client's subsequent reads in the same way as they would be causally ordered across partitions upon commitment, thus guaranteeing causality whilst achieving O_W.

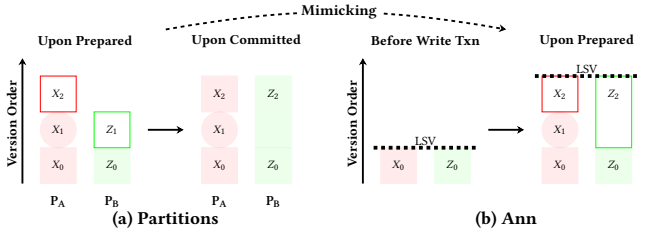


Figure 7: Advancing a client's LSV by mimicking the partition-side 2PC. Left: partition-side dynamics of 2PC. Right: advancing the client's LSV upon finishing the prepare phase. Squares refer to Ann's writes.

Example 6.1. In Figure 7, Ann is writing to the keys X and Z via a write-only transaction, and her original LSV is at X_0 and Z_0 (left in (b)). Upon finishing the prepare phase, the partitions have

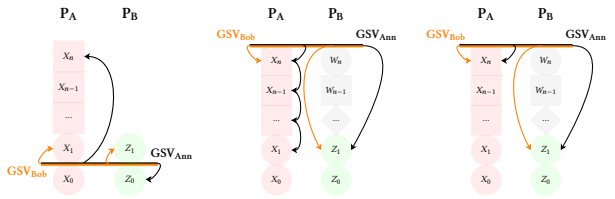
[5] Two major differences are that, unlike the traditional 2PC, (i) a client sends, in addition to the coordinator, to each cohort a prepared message directly, and (ii) upon receiving a query, a cohort proactively confirms with the coordinator the commitment of a transaction if it has voted "yes" but not yet received the commit.

Algorithm 2 The Eiger-NOC2 Algorithm

```
1:  /* Eiger-NOC2 adopts Eiger-PORT's 2PC variant, except
2:  coord/cohorts (i) return prepared timestamp in addition,
3:  and (ii) perform commits asynchronously. See E. */

4:  ____ Client-side Data Structures & Methods ____
5:  last[svr]: last committed timestamp on server svr
6:  latestWrite[key]: key → txnid, t_own

7:  procedure GET_ALL(K : set of keys)
8:      gsv ← min(last) // global safe view
9:      parallel-for k ∈ K do
10:         txnid, t_own ← latestWrite[k]
11:         rs[k], last[svr] ← GET(k, gsv, t_own, txnid)
12:     return rs

13: procedure PUT_ALL(W : set of ⟨key, value⟩)
14:     txnid ← generate new transaction ID
15:     parallel-for ⟨k, v⟩ ∈ W do
16:         if k.server is coordinator then
17:             ts_svr, t_prep ← WRITE_COORD(...) // see E
18:         else
19:             ts_svr, t_prep ← WRITE_COHORT(...) // see E
20:         last[k.server] ← max(ts_svr, last[svr])
21:         t_own ← max(t_prep, t_own) // t_own initialized as -1
22:     for k ∈ W.keySet do
23:         if latestWrite[k].t_own < t_own then
24:             latestWrite[k] ← (txnid, t_own)
25:     return

26: ____ Partition-side Data Structures & Method ____
27: vers: multi-versioned DB ⟨key, value, t_prep, t_com⟩
28: ts_svr: latest safe time
29: pending: uncommitted write txns txnid → (t_pend)

30: procedure GET(k, gsv, t_own, txnid)
31:     ver ← vers[k].at(gsv)
32:     if t_own ≥ ver.t_com then
33:         if txnid ∈ pending then
34:             return vers[k].at(pending[txnid]), ts_svr
35:         else
36:             return vers[k].at(t_own), ts_svr
37:     return ver, ts_svr
```

prepared X_2 and Z_1, respectively. Both timestamps are assigned by advancing the partitions' local Lamport clocks, respectively, which are not synchronized (left in (a)). The write transaction is eventually committed at version 2 (i.e., the maximum of the prepared timestamps), with Z_1 promoted to Z_2 in particular (right in (a)). This commit phase is mimicked on the client side even before it happens. Specifically, the prepared messages sent back to the 2PC coordinator are also received by Ann to advance her LSV according to the proposed prepare timestamps. Consequently, Ann is able to correctly order the prepared-only writes even before they are committed (right in (b)).

Regarding challenge (ii), we leverage GSVs to achieve convergence while keeping read-atomic reads. Similar to Eiger-PORT, an Eiger-NOC2 client also encodes a database snapshot using a version vector and takes the minimum as its GSV. Eiger-PORT has a conservative view of RA in the sense that it tends to return the exact versions written by a write transaction, while RA in fact allows part of the reads to fetch higher versions as long as no fractured reads are exhibited. This forces Eiger-PORT to search for a conservative snapshot from the database within a client's GSV, which depends on whether a version was written by itself or another client. Consequently, this would result in different orderings for the versions among clients. In contrast, we recognize that it is possible to return the highest committed version within a GSV without losing RA, and a convergent ordering can be agreed upon by all readers.



(a) Initial **(b) Eiger-PORT** **(c) Eiger-NOC2**

Figure 8: Illustrating convergence in Eiger-NOC2 and read atomicity in both algorithms. Squares, circles, and diamonds refer to Ann's, Bob's, and a third client's writes, respectively.

Example 6.2. In Figure 8, Ann and Bob are reading from X and Z using read-only transactions and both share the same GSVs. For simplicity, we assume no prepared versions, which suffices to distinguish the two algorithms. Figure 8a shows the initial scenario, where both algorithms behave the same. In particular, with the GSV at 0, Bob returns his own committed writes X_1 and Z_1 by jumping over the GSV; Ann reads X_n written by herself for the same reason, but Bob's Z_0 as it is the only version included in the GSV.

Both GSVs are then advance to n. Bob still behaves the same in both algorithms, as shown in Figure 8b for Eiger-PORT and Figure 8c for Eiger-NOC2, and returns the highest committed versions within the GSV that satisfy RA, i.e., X_n and Z_1.

These two algorithms differ when it comes to Ann. Eiger-PORT goes through an expensive backward search until it hits a write by a different client, i.e., X_1 (Figure 8b), while Eiger-NOC2 allows Ann to return her most recent write X_n (Figure 8c). Note that both returns exhibit no fractured reads, thus satisfying RA [7]. Nonetheless, in Eiger-PORT, Ann orders X_n before X_1, while Bob sees an opposite order. This is allowed by TCC, which, however, does not satisfy convergence. In contrast, a convergent order, i.e., X_1 before X_n, is established on both Ann and Bob. Moreover, returning X_n to Bob reduces the partition's overhead of backward search.

6.3 Algorithm

Eiger-NOC2 leverages the dual view to improve Eiger-PORT's read-only and write-only transactions. Both algorithms utilize version vectors to encode database snapshots and Lamport clocks to capture causal relations among transactions. We present its pseudocode in Algorithm 2, where we defer to Appendix E the partition-side procedures for transactional writes that are largely shared by both algorithms.

Transactional Reads. A client leverages its dual view when performing transactional reads. The GSV is taken as the minimum of *last* (line 8), a version vector encoding the most recent database snapshot that includes committed versions only (line 5). The LSV per key is represented by a pair $(txnid, t_{own})$, with t_{own} the commit timestamp of the latest write to the key by the client and *txnid* the associated write transaction's identifier. Upon receiving a read request, the server returns the highest committed version *ver* within the client's GSV (line 31), unless it is aware of a later, at least fully prepared, version of its own. Specifically, if the LSV t_{own} is larger than *ver*'s commit timestamp (line 32), meaning that there indeed exists a safe version by the client that is more advanced than *ver*, then the server returns the version at t_{own} (line 36). Note that, if the version has not been committed yet, the server finds the version from *pending* (line 34).

Transactional Writes. Both Eiger-NOC2 and Eiger-PORT adopt the variant of 2PC in [29] (see also Section 6.1). Eiger-NOC2 further

adapts it for O_W and convergence mainly in two ways. First, Eiger-NOC2 completes the commit phase asynchronously, allowing any subsequent transactions to race with it. Second, how the commit timestamp of a write transaction is decided by the coordinator is mimicked at the client side, so that clients can safely read prepared-only writes of their own without breaking TCCv.

More specifically, when processing a write transaction, each partition (or 2PC cohort) prepares a version with a timestamp extracted from its Lamport clock. The 2PC coordinator chooses the highest one among all the received timestamps as the commit timestamp for the write transaction. The cohorts then proceed with the commits asynchronously. Each of the coordinator and cohorts returns its latest safe time ts_{svr}, which the client uses to advance its GSV (line 20), alongside its proposed prepare timestamp t_{prep}, which is used to construct the client's LSV with respect to the write transaction (line 24), mimicking how the commit timestamp would be decided on the server side.

6.4 Correctness

Eiger-NOC2 adheres to NOC-NOC's performance criteria by design and improves Eiger-PORT by also providing O_W. The reasoning for RA-NOC2 (Section 4.3) applies to Eiger-NOC2 in general. In particular, to guarantee safe asynchronous commits for O_W, Eiger-NOC2 leverages the LSV that keeps track of a client's own writes *in a causally consistent order* with respect to other writes across the database. This enables a client's accesses to its own writes, even when they are not committed (on the server side) yet.

We also prove Eiger-NOC2's correctness: any read/write transaction satisfies both TCC and convergence properties. Intuitively, Eiger-NOC2 establishes the causal relations among transactions using Lamport clocks. Moreover, it leverages GSVs to represent safe snapshots of the database, where returning the most recent versions within a GSV guarantees no fractured reads. When a client jumps over the GSV to fetch its own writes, the LSV ensures that the jump is aligned along transactional boundaries and thus satisfies RA. Finally, convergence is achieved, since, by updating their dual views, all clients always share the same total order of versions per key, which is established on the partition via monotonically advancing Lamport clocks. We provide the proof in Appendix B.

7 EIGER-NOC2 EVALUATION

We assess Eiger-NOC2, showing its superior performance over a strong baseline, i.e., the NOCS-optimal Eiger-PORT algorithm (which provides a weaker isolation guarantee than Eiger-NOC2), and data freshness.[6]

7.1 Implementation, Setup, and Workloads

Implementation. We build Eiger-NOC2, along with Eiger-PORT, on the RAMP codebase [7]. This consists of around 1000 LOC in Java for each algorithm, where we reuse RAMP's facilities such as distributed hash table and serializer. Eiger-NOC2 utilizes the same variant of 2PC as in Eiger-PORT. Similar to RA-NOC2, the front-end of a partition executes client-requested transactions.

[6]We have also observed that Eiger-NOC2 significantly outperforms Eiger [29]; both algorithms provide the same isolation guarantee. See Appendix F for the experimental results.

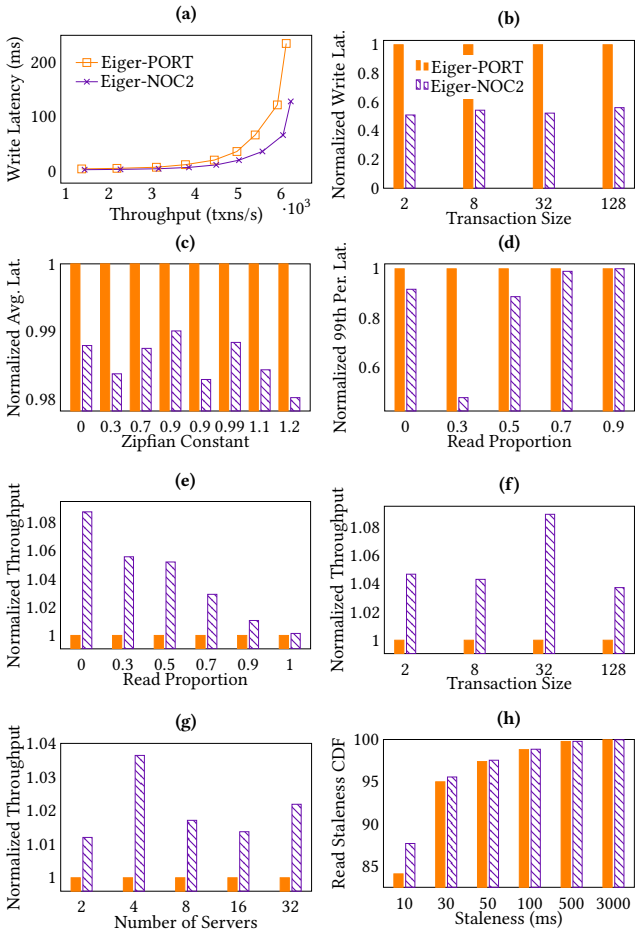


Figure 9: Performance and data freshness comparisons. Eiger-NOC2 provides TCCv which is strictly stronger than TCC provided by Eiger-PORT.

Experimental Setup. We run our experiments on an Emulab [45] cluster of machines, each with 2.4GHz Quad-Core Xeon CPU, 12GB RAM, and a 1Gbps network interface. By default, we use 8 client machines and 8 servers for partitioning the database. For each data point, we report the average over five trials, each lasting 60 seconds.

Workloads. We use the same workload generator as for RA-NOC2 and match Eiger-PORT's default parameters [31]: 32 threads per client, a Zipfian distribution with a skew factor of 0.8, 1 million keys, 128-byte values, 5 keys per transaction, and 90% reads.

7.2 Evaluation

Summary of Results. Our analysis results show that Eiger-NOC2, even with a stronger isolation guarantee, still surpasses Eiger-PORT in all experiments conducted. This demonstrates *the effectiveness of optimizing writes*, even only with O_W, in improving the overall system performance. In particular, Eiger-NOC2 exhibits significantly lower write latency and noticeably higher throughput. Moreover, the 99th percentile latency in Eiger-NOC2 is consistently lower or on par with Eiger-PORT, which demonstrates that the O_W optimization does not introduce extra overhead. Eiger-NOC2 also scales well with an increasing number of partitions and is resilient

to larger-sized transaction workloads. Finally, it achieves promising and slightly better data freshness results than Eiger-PORT.

Latency Improvement. Figure 9a depicts the write latency as a function of system throughput. Compared to Eiger-PORT, Eiger-NOC2 achieves higher throughput (up to 5%) with the same write latency and lower write latency (up to 46%) with the same throughput. The write latency improvement is significant, around 50%, independent of the transaction size; see Figure 9b. Despite varying skews (Figure 9c) and read/write ratio (Figure 9d), the latency in Eiger-NOC2 is overall lower than Eiger-PORT, and the improvement tends to increase under highly skewed and write-heavy workloads. Eiger-NOC2 owes all these improvements to the O_W optimization.

Throughput Improvement. Overall, Eiger-NOC2 exhibits higher throughput than Eiger-PORT under various workloads. Figure 9e shows that, when writes dominate the workload, the improvement becomes more pronounced. Figure 9f depicts that Eiger-NOC2 consistently outperforms Eiger-PORT regardless of transaction sizes, with up to 10% improvement. As shown in Figure 9g, Eiger-NOC2 also scales better when we increase the number of database partitions. All these throughput improvements can be attributed mainly to two factors: (i) the O_W optimization that boosts the overall system performance and (ii) the precise capturing of TCCv snapshots, which reduces the server-side overhead of backward search for the safe versions.

Data Freshness. From Figure 9h we can observe that over 87% of the reads in Eiger-NOC2 (slightly more than that in Eiger-PORT) are guaranteed to be 10ms staler than up-to-date values, and almost all reads experience less than 500 ms staleness. This is mainly because Eiger-NOC2 (like RA-NOC2) always pushes a client dual view to the most recent, safe snapshot of the database by synchronizing with the partitions and co-located clients.

8 DISCUSSION

Non-blocking Writes. N_W focuses on non-blocking concurrency control mechanisms. When coupling them with an atomic commitment protocol (ACP) for committing write transactions (NOC-NOC assumes 2PC given its wide adoption in practice), a transaction system may not make progress during failures as ACPs are inherently blocking [9] when, e.g., network partitions occur. Many solutions exist for mitigating this blocking problem in the literature. Both Eiger-NOC2 and Eiger-PORT employ the 2PC variant that always commits [29] (see Section 6.1).

RA-NOC2 runs the Cooperative Termination Protocol (CTP) [9], which has been demonstrated to be both lightweight and effective in practice [7]. CTP can always complete a transaction when failures occur during the commit phase and a server has prepared the transaction but times out waiting for the commit message. Note that, by further leveraging LSVs, the writes of the transaction which are already fully prepared can be safely returned, even before CTP recovers the blocked server.

Overhead of Local Computations. NOC-NOC, like many other design objectives or impossibility results such as SNOW and NOCS, concentrates on communication complexity. However, there may be other factors that negatively affect system performance, e.g., the overhead of local computations. In particular, even though this is usually negligible compared to network latency, especially in a geo-distributed setting, poor design choices or inefficient implementations could still accumulate system-wide overhead, hurting overall system performance. To return read-atomic versions in one round-trip, Eiger-PORT may perform expensive recursive scans of the database. In both RA-NOC2 and Eiger-NOC2, to achieve O_R and O_W, maintaining the dual views across clients may incur extra overhead under extremely skewed workloads, although we have not observed this in practice. Investigating the inherent trade-off between communication complexity and the overhead of local computations is interesting future work.

9 RELATED WORK

Improving Existing Algorithms. Through the lens of NOC-NOC, we have examined a collection of RA(+) and TCC(v) transaction algorithms in Section 3.4, with the focus on the RAMP-family and Eiger-family algorithms. There are many other algorithms in the literature which do not fulfill all the NOC-NOC performance criteria. These include MySQL Cluster [34] for read committed, RAMP with faster commit [7] for read atomicity, COPS [28] and COPS-SNOW [30] for causally consistent read-only transactions (with single-key writes), and a large number of TCCv systems, e.g., Orbe [18], GentleRain [19], Cure [2], Wren [40], Contrarian [17], PaRiS [41], UniStore [10], and OCC [42].

Note that, as NOC-NOC subsumes both SNOW and NOCS, any transaction algorithms, that are suboptimal, or even optimal (e.g., MySQL Cluster is NOCS-optimal; COPS-SNOW is SNOW-optimal), with respect to these two design objectives, can be potentially optimized to achieve better system performance. Note also that, as TCCv is compatible with NOC-NOC (Section 3.3), it is also achievable under NOCS. This resolves the conjecture that TCC is the upper bound of achievable isolation levels for NOCS-optimal read-only transactions in the presence of transactional writes [31].

Design Objectives. Substantial performance criteria have been proposed for designing highly efficient distributed transactions [3, 16, 17, 25, 30, 31, 44]. These criteria focus on optimizing reads. However, some of them, such as SNOW [30] and its followup [25], still miss crucial factors such as the metadata size, which also impact the system latency and throughput (see, e.g., Figure 1). Moreover, all these criteria, including NOCS, overlook how optimizing writes can potentially improve the overall performance, even under read-heavy workloads. In contrast, our NOC-NOC design objective aims at optimizing both reads and writes. We demonstrate its effectiveness through our two case studies.

Some design objectives have stronger data freshness requirements (e.g., minimal progress [16]), thus restricting achievable combinations of performance criteria [3, 16, 17, 44]. NOC-NOC assumes a weaker freshness criterion, as for SNOW and NOCS [25, 30, 31], which allows returning stale snapshots. As shown by our evaluation, promising data freshness results can still be achieved in practice.

10 CONCLUSION

We have proposed the NOC-NOC design objective and established the related impossibility results. Examining existing transaction algorithms in the light of NOC-NOC, we have identified a significant gap in the design space. We have therefore designed two algorithms

that fulfill all six NOC-NOC criteria. Our evaluation shows their promising system performance and data freshness results.

Along with these two case studies, we have presented the dual view which, when coupled with version vectors, can be leveraged to design NOC-NOC-optimal transaction algorithms that provide other isolation guarantees, in addition to RA+ and TCCv.

We expect NOC-NOC to help transaction system developers rethink their designs and implementations by taking into account the optimization of writes, avoiding efforts on achieving the impossible, and guiding them to focus on what is actually possible. As discussed in Section 8, our next step is to explore the inherent trade-off between NOC-NOC and the overhead of local computations.

REFERENCES

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49.

[2] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *ICDCS 2016*. IEEE Computer Society, 405–414.

[3] Karolos Antoniadis, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel. 2020. The Impossibility of Fast Transactions. In *IPDPS'20*. IEEE, 1143–1154.

[4] Masoud Saeida Ardekani, Pierre Sutra, Nuno Preguiça, and Marc Shapiro. 2013. Non-Monotonic Snapshot Isolation. arXiv:1306.3906 [cs.DC]

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *SIGMETRICS'12*. ACM, 53–64.

[6] Anonymous Authors. July, 2023. Supplementary material for submission "NOC-NOC: Towards Performance-optimal Distributed Transactions. https://github.com/anonymous-sigmod24.

[7] Peter Bailis, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (2016), 15:1–15:45.

[8] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD'95*. ACM Press, 1–10.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[10] Manuel Bravo, Alexey Gotsman, Borja de Régil, and Hengfeng Wei. 2021. UniStore: A fault-tolerant marriage of causal and strong consistency. In *USENIX ATC 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 923–937.

[11] Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *PODC*. 7.

[12] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC'13*. USENIX Association, 49–60.

[13] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR'15 (LIPIcs)*, Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 58–71.

[14] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proc. VLDB Endow.* 14, 12 (2021), 3014–3027.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC'10*. ACM, 143–154.

[16] Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2019. Distributed Transactional Systems Cannot Be Fast. In *SPAA'19*. ACM, 369–380.

[17] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe? *Proc. VLDB Endow.* 11, 11 (2018), 1618–1632.

[18] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: scalable causal consistency using dependency matrices and physical clocks. In *SoCC 2013*. ACM, 11:1–11:14.

[19] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *SoCC 2014*. ACM, 4:1–4:13.

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *ATC'19*. USENIX Association, 1–14.

[21] ElectricSQL. Accessed in July, 2023. https://electric-sql.com/.

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

[23] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing consistency properties for fun and profit. In *PODC'11*. ACM, 197–206.

[24] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.

[25] Kishori M. Konwar, Wyatt Lloyd, Haonan Lu, and Nancy A. Lynch. 2021. SNOW Revisited: Understanding When Ideal READ Transactions Are Possible. In *IPDPS'21*. IEEE, 922–931.

[26] Butler Lampson and Howard E. Sturgis. 1979. Crash recovery in a distributed storage system. Xerox Palo Alto Research Center.

[27] Si Liu. 2022. All in One: Design, Verification, and Implementation of SNOW-Optimal Read Atomic Transactions. *ACM Trans. Softw. Eng. Methodol.* 31, 3 (2022).

[28] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*. ACM, 401–416.

[29] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI'13*. USENIX Association, 313–328.

[30] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *OSDI'16*. USENIX Association, 135–150.

[31] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *OSDI'20*. USENIX Association, 333–349.

[32] Microsoft. Accessed in July, 2023. Azure CosmosDB DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[33] MongoDB. Accessed in July, 2023. Read Isolation, Consistency, and Recency. https://docs.mongodb.com/manual/core/read-isolation-consistency-recency/.

[34] MySQL. Accessed in July, 2023. MySQL Cluster CGE. https://www.mysql.com/products/cluster/.

[35] Neo4j. Accessed in July, 2023. https://neo4j.com/.

[36] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (1979), 631–653.

[37] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. 2016. Causal Consistency: Beyond Memory. *SIGPLAN Not.* 51, 8, Article 26 (2016), 26:1–26:12 pages.

[38] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.* 6, 11 (2013), 1068–1079.

[39] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP'11*. ACM, 385–400.

[40] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2018. Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. In *DSN 2018*. IEEE Computer Society, 1–12.

[41] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2019. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *ICDCS 2019*. IEEE, 304–316.

[42] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2021. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 527–542.

[43] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *PDIS*. IEEE Computer Society, 140–149.

[44] Alejandro Z. Tomsic, Manuel Bravo, and Marc Shapiro. 2018. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware'18*. ACM, 120–133.

[45] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*. USENIX Association, 255–270.

[46] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3 (2021), 17:1–17:35.

[47] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. 2018. wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events. In *OSDI'18*. USENIX Association, 527–543.

A CORRECTNESS PROOF OF RA-NOC2

This section presents the proof that RA-NOC2 guarantees both RA and RYW, and thus RA+.

Read Atomicity. A system is said to guarantee RA if, for every read transaction T_r and write transaction T_w, T_r observes a state of the database before any of the changes of T_w are applied or after all changes of T_w are applied.

Lemma A.1. *RA-NOC2 guarantees RA in every execution.*

Proof. Assume T_r is a read transaction, its dual view is represented by GSV and LSV, and it returns value v. Assume T_w is the write transaction that wrote v at timestamp ts. Consider the following two cases:

(1) GSV is above ts: This means every partition involved in T_r has committed all changes made by T_w, as if T_w was still pending on one of the partitions involved, then the version vector value of that partition would be lower than ts, and thus GSV could not have advanced above it. Note that GSV is selected as the minimum of all the version vector values of the partition involved. If all the changes are committed, and the server returns the largest possible version below the GSV, then we know there are no fractured reads.
(2) GSV is below ts: This means T_r and T_w have been issued by the same client, as a client is only allowed to read beyond the GSV for its own writes. It also means that v is the value at LSV for the client. This means that the transaction T_w has been prepared on all partitions and that the client will, using LSV, always read the full transaction, not a fractured read, as it will fetch that version, or a later one, but never one below.

□

Read Your Writes. A system is said to guarantee RYW if for every T_r (read-only) and T_w (write-only) by the same client if T_r happens after T_w, then T_r returns a result which includes all changes made by T_w, i.e. either the version written by T_w or an even later one.

Lemma A.2. *RA-NOC2 guarantees RYW in every execution.*

Proof. Assume there are two transactions T_r (read-only) and T_w (write-only) by the same client. Assume T_w writes with timestamp ts. Assume T_r happens after T_w. Note that the LSV of the client will incorporate ts after T_w's prepare phase is over, thus by the time T_r starts for every key of T_w the LSV is $\geq ts$. Consider the following two cases based on the GSV during T_r:

(1) GSV is above LSV: T_r will check if there exists a version in between the GSV and the LSV, if so it returns it. This means that the version returned has been committed after ts, and thus is considered to be a state more advanced than that of T_w, which guarantees RYW. If it does not find a version between GSV and LSV it simply returns the version at LSV. This version is either that of T_w, or that of the latest write by the client which happened after T_w, in either case this satisfies RYW, as the latest write by the client on the key is returned.
(2) LSV is above GSV: This is the same as the second subcase in case 1, i.e. this returns the latest write by the client on the key since the LSV is advanced to that version every time a write transaction is done on the client side, which by definition satisfies RYW.

□

Combining Lemma A.1 and Lemma A.2, we prove that RA-NOC2 guarantees RA+.

B CORRECTNESS PROOF OF EIGER-NOC2

This section provides the proof of Eiger-NOC2's correctness: it satisfies causal consistency, read atomicity, and convergent conflict handling, thus guaranteeing TCCv.

Causal consistency. CC is characterized by three properties:

- P_1: if T_x and T_y are two operations by the same client and T_x happend before T_y in time then $T_x \rightarrow T_y$.
- P_2: if T_x is a write transaction, T_y a read, and T_y returns the version written by T_x then $T_x \rightarrow T_y$.
- P_3: if $T_x \rightarrow T_y$ and $T_y \rightarrow T_z$, then $T_x \rightarrow T_z$

Lemma B.1. *If a read-only transaction reads at the GSV, then it returns a snapshot of the database, including the write transaction that wrote the return values, and all earlier writes that said values depend on.*

Proof. Say a transaction T_y reads below GSV and returns a value v with commit time t_{com_x} and written by T_x. Let T_w be an arbitrary write transaction with commit time t_{com_w} such that $T_w \rightarrow T_x$, then we know $t_{com_w} < t_{com_x}$ because the system uses Lamport clocks which are advanced on every write transaction, and thus if T_x is causally after T_w then its commit time has to be above that of T_w. This means T_y returns a value which observes the results of T_x, plus all the write transactions with commit time below that of x, which we have shown to be all the writes causally ordered before x. Note that even if v is only prepared, i.e. it was read through the LSV, it is still ordered according to the future commit timestamp t_{com_x}, so we do not lose the generality of this proof when we allow to read prepared versions via the LSV. □

Lemma B.2. *Any execution of Eiger-NOC2 guarantees P_1.*

Proof. If c is a client and T_x and T_y two requests by it such that T_x happens before T_y then we have four possible cases, depending on whether the requests are read-only or write-only transactions.

(1) T_x and T_y are writes. The system is built using Lamport clocks, which are advanced during the prepare phase, this means that T_y will naturally have a timestamp more advanced than T_x, as all the clocks will have advanced by the time the prepare phase of T_x has ended and c can start a new transaction. Hence, $T_x \rightarrow T_y$.
(2) T_x is a write and T_y is a read. Let t_{com} be the commit time for T_x, if the GSV is $\geq t_{com}$ then T_y will be able to read a version which includes T_x as shown by Lemma B.1. Note that T_x is a write by the same client performing the read T_y, thus, if the GSV is below the commit time, Eiger-NOC2 will still return the result of T_x, or a further write by c which includes T_x, by using the LSV which fetches the latest at least fully prepared write on every key. Moreover, if the two requests have a disjoint set of keys, then P_1 still holds, because the GSV increases monotonically, and so does the LSV, and both

are updated at the end of the prepare phase, before c can start the next request, so T_y will always read from a state no older than T_x.

(3) T_x is a read and T_y is a write. T_x will naturally read a value $\leq$ the current Lamport time of the server. T_y will then write, and during the write the Lamport time will be advanced, thus making the commit timestamp for T_y larger than the largest possible read by T_x.

(4) T_x and T_y are both reads. The GSV advances monotonically, therefore for T_x it will be $\leq$ than the GSV for T_y. Similarly, the LSV also advances monotonically. This means the dual view of T_y contains the dual view of T_x, therefore T_y will never read an older state than T_x.

□

Lemma B.3. *Eiger-NOC2 guarantees P_2 in any execution.*

Proof. Assume T_x is a write transaction with commit time t_{com} and T_y a read transaction and assume T_y returns the value that was written by T_x. Then there are two possibilities:

(1) t_{com} is above the GSV of the read: Then T_y must be a write by the same client, which is returning the result of the most recent write transaction by it. This means P_2 holds, as we are in the same situation as case 2 in Lemma B.2.

(2) t_{com} is below the GSV of the read: by Lemma B.1 we know that T_y observers the results of T_x and all previous writes that it depends on.

□

Lemma B.4. *P_3 is guaranteed in any execution of Eiger-NOC2.*

Proof. Let T_x, T_y and T_z be three transactions and $T_x \rightarrow T_y$ and $T_y \rightarrow T_z$. Consider the following cases:

(1) T_y and T_z are writes. This means that all three transactions are by the same client, therefore if we apply Lemma B.2 on $T_x \rightarrow T_y$ and on $T_y \rightarrow T_z$ we must have that $T_x \rightarrow T_z$.

(2) T_y and T_z are reads. This means they have to both be by the same client. If T_x is also a read, we can apply case 4 of Lemma B.2 on $T_x \rightarrow T_y$ and $T_y \rightarrow T_z$. If it is a write, we apply Lemma B.3 on $T_x \rightarrow T_y$ and case 4 of Lemma B.2 on $T_y \rightarrow T_z$.

(3) T_y is a write and T_z is a read. Then T_x and T_y are from the same client. If T_z is also by the same client then we apply Lemma B.2 on $T_x \rightarrow T_y$, and $T_y \rightarrow T_z$. Else, we apply Lemma B.3 on $T_y \rightarrow T_z$ instead.

(4) T_y is a read and T_z is a write. Then T_y and T_z are from the same client. We can apply Lemma B.2 if T_x is a read, or B.3 if it is a write, on $T_x \rightarrow T_y$ and then apply Lemma B.2 on $T_y \rightarrow T_z$.

□

Together, Lemma B.2, Lemma B.3 and Lemma B.4 prove that Eiger-NOC2 guarantees causal consistency.

Read atomicity. A system is said to satisfy read atomicity RA if for every read transaction T_r and every write transaction T_w, T_r either observes a state of the system after the changes of T_w have been applied, or before.

Lemma B.5. *Eiger-NOC2 guarantees RA on every execution.*

Proof. Consider two transactions, T_r (read-only) and T_w (write-only with commit timestamp t_{com}). Assume T_r returns v written by T_w. Consider the following two cases:

(1) The GSV in T_r is above the commit timestamp t_{com}. Then we know the write transaction T_w is committed on all partitions, otherwise, the GSV would not have advanced up to that point, and all the changes of T_w have been applied.

(2) The GSV in T_r is below the commit timestamp t_{com}. Then for T_r to be able to read v, it means it is a read issued by the same client as T_w and was read using the LSV. This means all changes applied by T_w are at least prepared on all partitions, and v is the latest write issued by the client. This means, using the LSV, T_r can correctly fetch every part of the T_w transaction, and thus observe all its changes.

□

Convergence. We need to prove that our system guarantees convergent conflict handling: two transactions T_x and T_y that are not causally ordered, i.e. $T_x \nrightarrow T_y \wedge T_y \nrightarrow T_x$, are said to be in conflict. To guarantee convergence this conflict needs to be handled equally on all partitions.

Lemma B.6. *Given two write transactions T_x and T_y, such that $T_x \nrightarrow T_y \wedge T_y \nrightarrow T_x$, they are ordered the same way on all partitions.*

Proof. The final commit timestamp of a transaction is selected as the maximum of the Lamport clocks of the partitions involved after the prepare phase, and the Lamport clock is advanced every time. Therefore, the latest transaction between T_x and T_y to arrive at a partition will have a larger Lamport clock value on that partition, they will never have the same value. This means the latest transaction to arrive at the most advanced partition, will have the latest commit timestamp and thus be ordered after, i.e. latest-write-wins. And this ordering is equal on all partitions as the coordinator makes sure the commit timestamp is the same on every partition. □

Lemma B.7. *A client c will read monotonically increasing versions for any key K.*

Proof. This is guaranteed by the fact that both LSV and GSV are monotonically increasing. This means that if a version K_i is read at time t, at a future time t', no version below K_i can ever be returned because the dual view at this time includes that of time t. □

Lemma B.8. *Given two clients c_1 and c_2, they both view the same causal order of operations and thus guarantee convergence.*

Proof. By Lemma B.6 and by the proof of CC, we know that writes are ordered the same across partitions. By Lemma B.7 we know that both c_1 and c_2 return monotonically increasing versions. This means that the order of operations for the two clients is the same, as the order given by the Lamport clock is equal for all clients, and they all read based on that order. □

Combining all the above lemmas, we prove that Eiger-NOC2 guarantees TCCv.

C PSEUDOCODE OF RA-NOC

This section shows the pseudocode of RA-NOC, an RA+ algorithm that meets the NOCS design objective.

Algorithm 3 The RA-NOC Algorithm: Client-side Logic

```
1:  last[svr]: last safe stamp in server svr

2:  procedure GET_ALL(K : set of keys)
3:      ts_c ← min({last[svr] | svr storing k ∧ k ∈ K})
4:      parallel-for k ∈ K do
5:          rs[k], last[svr] ← GET(k, ts_c)
6:      return rs

7:  procedure PUT_ALL(W : set of ⟨key, value⟩)
8:      ts ← generate new timestamp
9:      parallel-for ⟨k, v⟩ ∈ W do
10:         ts_svr ← PREPARE(⟨k, v, ts⟩)
11:         last[svr] ← max(ts_svr, last[svr])
12:     parallel-for svr ∈ {svr|svr storing k ∧ k ∈ W.keySet} do
13:         ts_svr ← COMMIT(ts)
14:         last[svr] ← max(ts_svr, last[svr])
15:     while min(last) < ts do
16:         block
17:     return
```

Algorithm 4 The RA-NOC Algorithm: Partition-side Logic

```
1:  vers: multi-versioned DB ⟨key, value, timestamp⟩
2:  latest: highest committed timestamp
3:  pending: uncommitted write txns

4:  procedure PREPARE(ver)
5:      vers.add(ver)
6:      pending.add(ver.ts)
7:      return HST(pending,latest)

8:  procedure COMMIT(ts_com)
9:      latest ← max(ts_com, latest)
10:     pending.remove(ts_com)
11:     return HST(pending,latest)

12: procedure GET(k, ts_req)
13:     hst ← HST(pending,latest)
14:     for ver.below(ts_req) do
15:         if ver.ts ∉ pending then
16:             return ver.value, hst

17: function HST(s,ts)
18:     if s = ∅ then
19:         return ts
20:     else
21:         return min(s)
```

D COMPARISON WITH RA-NOC

This section presents a performance comparison between Eiger-NOC2 and RA-NOC. We follow the same experimental setup as in Section 5.2. Our experimental results show that RA-NOC2 significantly outperforms RA-NOC under various workloads. Moreover, RA-NOC2 achieves much better data freshness results.

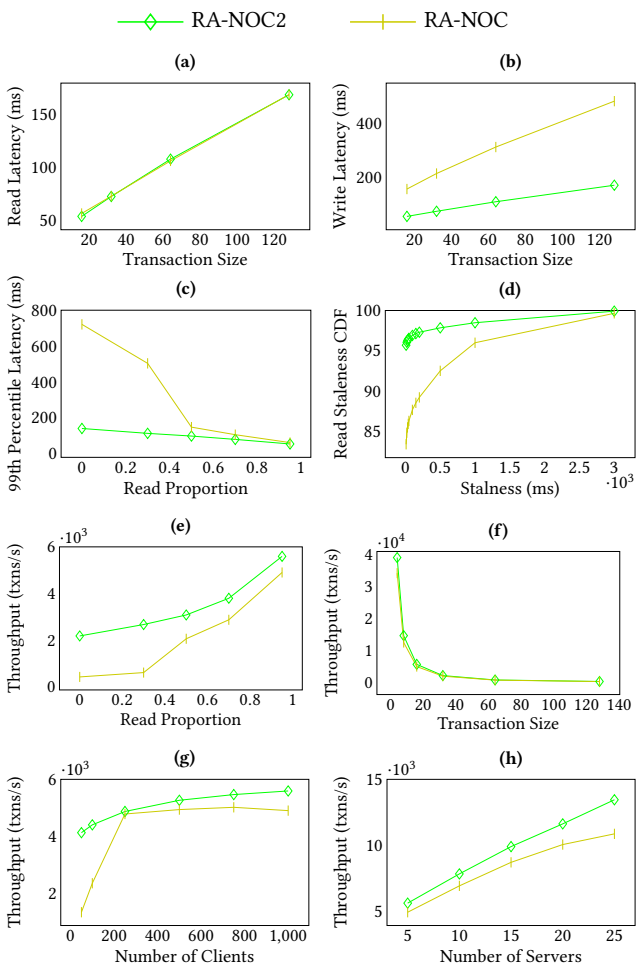


Figure 10: Performance and data freshness comparisons between RA-NOC and RA-NOC2.

E THE COMPLETE PSEUDOCODE OF EIGER-NOC2

This section gives the full pseudocode of Eiger-NOC2, including the partition-side write procedures.

Algorithm 5 The Eiger-NOC2 Algorithm: Client-side Logic

```
1:  last[svr]: last committed stamp in server svr
2:  latestWrite[key]: key → (txnid, t_own)
3:  gsv: global safe view

4:  procedure GET_ALL(K : set of keys)
5:      gsv ← min(last)
6:      parallel-for k ∈ K do
7:          txnid, t_own ← latestWrite[k]
8:          rs[k], last[svr] ← GET(k, gsv, t_own, txnid)
9:      return rs

10: procedure PUT_ALL(W : set of ⟨key, value⟩)
11:     txnid ← generate new transaction ID
12:     parallel-for ⟨k, v⟩ ∈ W do
13:         if k.server is coordinator then
14:             ts_svr, t_prep ← WRITE_COORD(k,v,gsv,txnid)
15:         else
16:             ts_svr, t_prep ← WRITE_COHORT(k,v,gsv,txnid)
17:         last[k.server] ← max(ts_svr, last[svr])
18:         t_own ← max(t_prep, t_own) // t_own initialized as -1
19:     for k ∈ W.keySet do
20:         if latestWrite[k].t_own < t_own then
21:             latestWrite[k] ← (txnid, t_own)
22:     return
```

Algorithm 6 The Eiger-NOC2 Algorithm: Partition-side Logic

```
1:  vers: multi-versioned DB ⟨k, v, t_prep, t_com⟩
2:  ts_svr: latest safe time
3:  pending: uncommitted write txns txnid → (t_pend)

4:  procedure GET(k, gsv, t_own, txnid)
5:      ver ← vers[k].at(gsv)
6:      if t_own ≥ ver.t_com then
7:          if pending.contains(txnid) then
8:              return vers[k].at(pending[txnid]), ts_svr
9:          else
10:             return vers[k].at(t_own), ts_svr
11:     return ver, ts_svr

12: procedure WRITE_COORD(k,v, gsv, txnid)
13:     ver, t_prep ← PREPARE(k, v, gsv, txnid)
14:     return ts_svr, t_prep
15:     asynchronously do
16:         wait for prep msgs from cohorts
17:         t_com ← max(prep_msg.t_prep, t_prep)
18:         send t_com message to all cohorts
19:         COMMIT(ver, t_com)

20: procedure WRITE_COHORT(k, v,gsv,txnid)
21:     ver, t_prep ← PREPARE(k, v, gsv, txnid)
22:     return ts_svr, t_prep
23:     asynchronously do
24:         send prep msg with t_prep to coord
25:         wait for t_com msg from coord
26:         COMMIT(ver, t_com)

27: procedure PREPARE(k, v,gsv, txnid)
28:     t_pend ← LamportClock.current()
29:     pending[txnid] ← t_pend
30:     LamportClock.advance()
31:     ver ← vers[k].new(v, gsv, t_pend, txnid)
32:     ver.is_pending ← true
33:     t_prep ← LamportClock.current()
34:     return ver, t_prep

35: procedure COMMIT(ver, t_com)
36:     ver.t_com ← t_com
37:     ver.is_pending ← false
38:     pending.remove(ver.txnid)
39:     if pending.isEmpty() then
40:         ts_svr ← LamportClock.current()
41:     else
42:         ts_svr ← min(pending[txnid].t_pend)
```

F COMPARISON WITH THE EIGER FAMILY

This section presents a comparison among Eiger-NOC2, Eiger-PORT, and Eiger. Our experimental results show that Eiger-NOC2 is particularly superior to Eiger under various workloads.

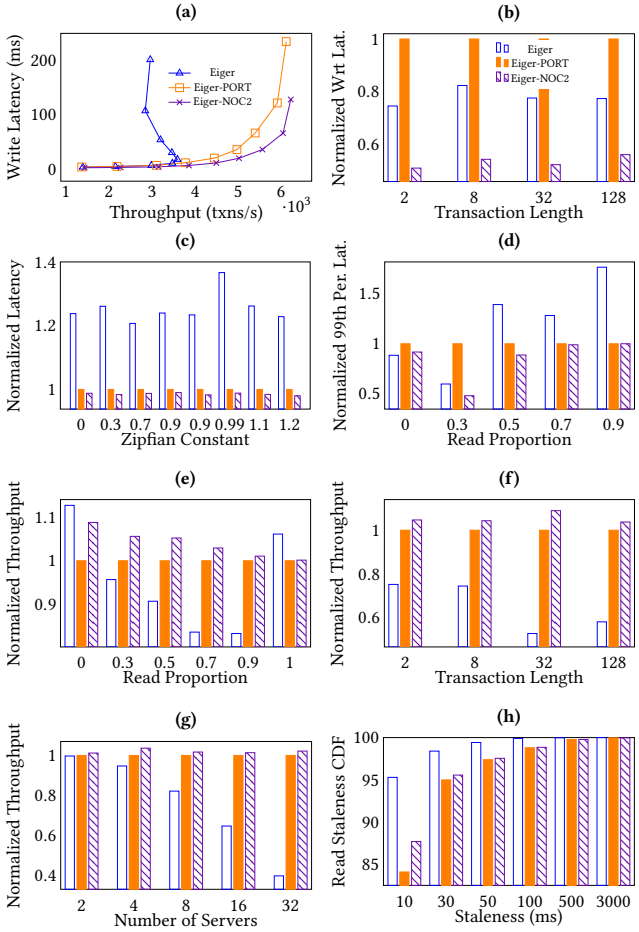


Figure 11: Performance and data freshness comparisons between Eiger-NOC2 and the Eiger-family algorithms.