Performance-Optimal Read-Only Transactions

Haonan Lu, *Princeton University;* Siddhartha Sen, *Microsoft Research;*
Wyatt Lloyd, *Princeton University*

https://www.usenix.org/conference/osdi20/presentation/lu

This paper is included in the Proceedings of the
14th USENIX Symposium on Operating Systems
Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the
14th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by USENIX

Performance-Optimal Read-Only Transactions

Haonan Lu[⋆], Siddhartha Sen[†], Wyatt Lloyd[⋆]
[⋆]*Princeton University,* [†]*Microsoft Research*

Abstract

Read-only transactions are critical for consistently reading data spread across a distributed storage system but have worse performance than simple, non-transactional reads. We identify three properties of simple reads that are necessary for read-only transactions to be performance-optimal, i.e., come as close as possible to simple reads. We demonstrate a fundamental tradeoff in the design of read-only transactions by proving that performance optimality is impossible to achieve with strict serializability, the strongest consistency.

Guided by this result, we present PORT, a performance-optimal design with the strongest consistency to date. Central to PORT are version clocks, a specialized logical clock that concisely captures the necessary ordering constraints. We show the generality of PORT with two applications. Scylla-PORT provides process-ordered serializability with simple writes and shows performance comparable to its non-transactional base system. Eiger-PORT provides causal consistency with write transactions and significantly improves the performance of its transactional base system.

1 Introduction

Large-scale web services are built on distributed storage systems. Sharding data across machines enables distributed storage systems to scale capacity and throughput. Sharding, however, complicates building correct applications because read requests sent to different shards may arrive at different times and thus return an inconsistent view of the data.

Consistently interacting with data in a distributed storage system thus requires transactional *isolation*, which unifies the view of data across shards. While general transactions provide isolation for reading and writing across shards, this paper focuses on *read-only transactions* that only read data. Read-only transactions are prevalent: they are used in systems without general transactions [4, 14, 31, 32, 34] and, even for systems with general transactions, they are often implemented with a specialized algorithm [10, 11, 34, 37, 38, 39, 51]. Read-only transactions are practically important because reads dominate real-world workloads: Facebook reported 99.8% reads for TAO [8] and Google reported three orders of magnitude more reads than general transactions for the ads workload (F1) that runs on Spanner [10]. They are also theoretically important because they provide a lower bound for other classes of transactions: anything impossible for read-only transactions is also impossible for any class of transactions that includes reads.

The dominance of reads in real-world workloads makes their performance the primary determinant of end-user latency and overall system throughput. Unfortunately, read-only transactions perform worse than simple, non-transactional reads due to the coordination required to present a consistent view across shards. Whether a view is consistent is determined by a system's *consistency model*: stronger consistency provides an abstraction closer to a single-threaded environment, greatly simplifying application code [33]. Thus, ideal read-only transactions would provide the strongest consistency *and* have optimal performance.

What is the "optimal" performance? Although recent work has studied optimality through the lens of latency [34], it did not consider throughput, which adds a fundamentally new dimension to this question. In this paper, we formalize the notion of optimality for read-only transactions and use it to explore the tradeoff between their consistency and performance. We posit that optimality should be defined by the algorithmic properties of simple reads that comprise a read-only transaction. *Simple reads* do not provide transactional isolation and thus capture the minimum work required to read data in a distributed storage system: **O**ne round of **N**on-blocking communication with a **C**onstant amount of metadata. As we elaborate in §3, these algorithmic properties (N, O, and C) precisely capture the additional coordination incurred by read-only transactions to present a consistent view. Thus, we define *performance-optimal* read-only transactions to be those with the same NOC properties as simple reads.

Our main theoretical result is that performance optimality is impossible in a system that provides **S**trict serializability—the strongest type of consistency. Specifically, our NOCS Theorem states that no read-only transaction algorithm can be performance optimal and provide strict serializability. This result holds even in systems that only support non-transactional writes, and thus applies to systems with and without more general types of transactions. It shows there is a fundamental choice in the design of distributed storage systems: they can either provide the strongest consistency or the best performance for read-only transactions, not both.

Guided by our impossibility result, we present the PORT design, which enables performance-optimal read-only transactions with the strongest consistency to date: process-ordered serializability. Previous performance-optimal transactions only provided relatively weak consistency (§5.1). PORT provides performance-optimal read-only transactions without harming either the latency or throughput of writes. The main mechanism enabling our design is a new specialized logical clock, called *version clocks*, that concisely capture the ordering constraints imposed by process-ordered serializability on read and write operations. PORT uses version clocks to tightly co-design its components. Version clock values index its multi-versioning framework, control what read-only transactions see, and control where writes are applied. They also enable optimizations that avoid the work of applying some concurrent writes (*write omission*) and limit the staleness of reads (*data freshness*).

We use the PORT design with the write omission and data freshness optimizations to build a new storage system, Scylla-PORT, that adds performance-optimal read-only transactions to ScyllaDB [47] while providing process-ordered serializability. As a single-versioned, non-transactional system, ScyllaDB provides a clean slate for implementing PORT and allows us to quantify the overhead of our performance-optimal read-only transactions relative to simple reads. ScyllaDB's simple reads are a challenging baseline as the system is aggressively engineered for high performance, including core-level sharding and custom lock-free data structures. Our evaluation shows that PORT's read-only transactions introduce low overhead, achieving throughput and latency within 3% of ScyllaDB on most of the workloads we test, and within 8% in the worst case. Our evaluation also compares PORT to a variant of OCC that is optimized for read-only transactions. PORT significantly outperforms OCC with at least double the throughput and at most half the latency because Scylla-PORT always finishes in one round while OCC's best case is two rounds.

We also applied PORT with data freshness optimizations to Eiger [32] to make its read-only transactions performance optimal while preserving the system's causal consistency and write transactions. Eiger is a challenging baseline because it can complete read-only transactions in a single round. Our evaluation shows that Eiger-PORT significantly improves performance with throughput up to 3× higher and latency up to 60% lower than Eiger. These improvements do come with some staleness relative to strongly consistent systems, but our data freshness optimizations keep the staleness low.

In summary, this work makes the following contributions:

- A fundamental understanding of the tradeoff between performance and consistency for read-only transactions. This includes a precise definition of performance optimality (§3) and the NOCS Theorem that proves optimality is impossible with strict serializability (§4).
- The PORT design that achieves performance-optimal read-only transactions with the strongest consistency to date by leveraging version clocks, a new type of logical clock that concisely captures the necessary ordering constraints (§6).
- The implementation and evaluation of two new systems based on the PORT design. Scylla-PORT is a clean-slate application of PORT to a non-transactional system, ScyllaDB (§7). Eiger-PORT makes the read-only transaction algorithm of Eiger performance optimal (§8, §9).

2 Background

Web service architecture. Web services are typically built using two tiers of machines: a stateless frontend tier and a stateful storage tier. The frontends handle end user requests by executing application logic that generates sub-requests to read or write data in the storage tier. We refer to the frontends as *clients* and the storage machines as *servers*, as is common. Web services are often replicated across multiple datacenters. For simplicity, we focus on a single datacenter setting, but our results also apply to multi-datacenter settings.

Read-only transactions. Read-only transactions provide a consistent, unified view of data spread across servers in a storage tier. They consist of one or more logical rounds of simple read requests issued in parallel to the servers, which collectively return a view satisfying the consistency model of the system. *One-shot transactions* [23] know the data locations of all reads prior to the transaction start. In contrast, *multi-shot transactions* may include key dependencies, where the data read in one shot determines what data to read in later shots. We study one-shot transactions for simplicity, because they are common, and because what is impossible for them is also necessarily impossible for multi-shot transactions. The NOCS Theorem thus also applies to multi-shot transactions. The PORT design for read-only transactions can be easily extended to support multi-shot transactions.

3 Performance-Optimal Read Transactions

This section explains the challenges of reasoning about performance, the rationale of our approach, and the set of algorithmic properties that define optimal performance.

3.1 Reasoning About Performance

The key challenges to reasoning about performance are identifying the fundamental overhead of read-only transactions and modeling it in a way that connects with practical designs.

Capturing the fundamental overhead. As a layer built upon simple reads, the performance of a read-only transaction is impacted by both the engineering factors in executing simple reads and the algorithmic properties of coordinating simple reads to find a consistent view. Engineering factors, such as load balancing, batching, and networking, equally affect simple reads and the read-only transactions built on them. In contrast, the algorithmic properties, such as rounds of communication, only affect read-only transactions. For instance, a read-only transaction protocol that requires multiple round trips incurs overhead due to those extra rounds of messages, while the read requests in each round are engineered the same as simple reads.

Thus, this work focuses on the algorithmic properties that capture the fundamental overhead of read-only transactions. These properties capture the additional overhead to coordinate a consistent view and are orthogonal to underlying engineering factors. More specifically, we answer the question, "given a system, how low can we make the performance overhead of read-only transactions relative to the system's simple reads?"

Being useful in practice. Our goal is to model optimal performance in a way that is both *theoretically insightful* and *practically useful*. Theoretical insights help clarify fundamental tradeoffs between performance and guarantees. Practically useful guidance helps us design better systems. Our NOCS Theorem (§4.1) and properties yield theoretical insights that lead to a better design, PORT (§6), that achieves better performance in practice. This shows that our modeling is practically useful (§5).

3.2 Approach Overview

To reason about optimal performance in a practically useful way, we examine the mechanisms used in existing systems to coordinate a consistent view across shards. These coordination mechanisms include blocking, extra messages, and metadata. Some systems *block* read operations until a consistent view is ready—e.g., systems that use two-phase locking. Almost all systems use *extra messages* to determine a consistent view, such as multiple round trips on the critical path of reads—e.g., OCC [24]—or approaches that asynchronously coordinate a consistent view—e.g., COPS-SNOW [34], GentleRain [15], Cure [3]. Finally, all systems we are aware of use *metadata* to help compute a consistent view for read-only transactions to return—e.g., timestamps, transaction ids. Figure 9 in Section 10 shows representative systems that use these mechanisms.

These coordination mechanisms cause read-only transactions to have worse performance than simple reads, as they consume additional system resources. Therefore, we define performance-optimal read-only transactions to be those that require the least amount of each coordination mechanism, making their performance closest to that of simple reads.

3.3 NOC: Optimal Performance

We now explain the NOC properties, which we use to define optimal performance for read-only transactions.

N: Non-blocking. A read-only transaction algorithm is *non-blocking* if servers process each read request without waiting for any external event, such as a lock to become available, a message to arrive, or a timer to expire.

Blocking for a read request increases the latency of the read-only transaction: the more time spent blocking, the longer the transaction takes to complete. It also decreases throughput due to the overhead of context switches. In practice, blocking can incur more serious performance issues, e.g., CPU underutilization and deadlocks, which are increasingly pronounced in modern services [44, 52].

O: One-round communication. A read-only transaction algorithm has *one-round communication* if it uses exactly one parallel round of on-path messages and does not have any off-path messages. This matches the messages of simple reads: the client sends a single request to each server holding relevant data, and each server sends a single response back. It excludes algorithms that use extra messages, such as those that require multiple rounds of on-path communication, e.g., to abort/retry. It also disallows coordinating through *off-path messages*, i.e., messages that are necessary for the read-only transactions but lie off the critical path of reads.

A message is an off-path message for read-only transactions if its removal affects *only* the correctness of read-only transactions. For example, COPS-SNOW [34] adds extra messages to writes. These messages are used for read-only transactions to find a consistent snapshot and are not necessary for processing writes. Because only the correctness of read-only transactions is affected if these messages are removed, they are off-path messages.

Additional rounds of on-path messages increase the latency of read-only transactions. Both extra on-path and off-path messages decrease system throughput because transmitting and processing them consume network and CPU resources that could otherwise be used to service requests.

C: Constant metadata. Metadata is the information required by a read-only transaction algorithm to coordinate consistent values. It is information a server needs to find the specific version of the data that will produce a consistent cross-shard view across reads in the same transaction. Examples of metadata include timestamps [2, 10], transaction ids [34, 41], and identifiers of participating servers [5].

A read-only transaction algorithm has *constant metadata* if the amount of metadata required to process each of its read requests is constant, i.e., it does not increase with the size of the system, the size of the transaction, or the number of concurrent operations. An example of constant metadata is one timestamp per read request for snapshot reads in Spanner [10]. An example of non-constant metadata is COPS-SNOW [34], which requires information about *many* concurrent read-only transactions to process each read request.

Transmitting and/or processing extra metadata consumes more resources, increasing latency and decreasing throughput. Its negative impact on performance has been reported in recent work [13, 14, 15]. We use Big-O notation, i.e., "constant," to capture the algorithmic complexity of metadata required for coordination. In practice, system designers should aim for as low a constant as possible. We realize this in our PORT design, which uses a single integer per read request.

Performance optimality. We deem an algorithm *performance optimal* if it satisfies the N+O+C properties because they capture the least coordination overhead and thus enable performance as close as possible to simple reads.

4 The NOCS Theorem

An ideal system would have performance-optimal read-only transactions that provide the strongest consistency. Our NOCS Theorem proves this ideal is impossible.

S: Strict serializability. Strict serializability is the strongest form of consistency, equivalent to linearizability [22] with the addition of transactional isolation. It requires that there exists a legal total order of transactions that respects the real-time order between transactions [42]. A *legal total order* ensures that the results of transactions are equivalent to a single entity processing them one by one. The *real-time order* ensures that if transaction T_2 starts after transaction T_1 ends, then T_1 must appear before T_2 in the total order. If T_1 and T_2 have overlapping lifetimes, then they are concurrent and can be placed in either order. Strict serializability gives application programmers the powerful abstraction of programming in a single-threaded, transactionally isolated environment.

4.1 NOCS is Impossible

Our main result is that performance-optimal read-only transactions (N+O+C) cannot provide strict serializability (S). This section present a condensed version of the proof. The full proof appears in our accompanying technical report [35].

The NOCS Theorem. *No read-only transaction algorithm satisfies all NOCS properties.*

System model. We model a distributed system as a set of processes that communicate by sending and receiving messages. This model is similar to that used in FLP [17]. A set of client processes (clients) issue requests to server processes (servers) that store the data. Processes are modeled as deterministic automata: in each atomic step, they may receive a message, perform deterministic local computation, and send one or more messages to other processes.

A transaction (operation) starts when a client sends the request messages to servers and ends when the client receives the last necessary server response. Two transactions (operations) are concurrent if their lifetimes overlap, i.e., neither begins after the other ends. If concurrent transactions (operations) access the same data item, then they conflict.

Assumptions. We make the following assumptions:

(A-0) There are ≥ 2 servers and ≥ 2 clients. Otherwise, optimal performance and strict serializability are trivial. All reads and writes eventually complete.

(A-1) The network and processors are reliable. Every message is eventually delivered and processed by the destination process. Processes are correct and never crash. By proving our impossibility result under these favorable conditions, it will necessarily hold when the system can fail.

(A-2) The network is either asynchronous [20], i.e., messages can be arbitrarily delayed, or partially synchronous [16], i.e., physical clocks ensure bounded delays.

Proof intuition. Due to network asynchrony, it is always possible for a read-only transaction to conflict with write operations and other concurrent read-only transactions. These requests occupy *an unstable region* in the system's history, where conflicts are possible and a total order has not yet been established. In contrast, the *stable region* is the part of history that precedes the unstable region, where all writes have committed and system states are finalized. Reading in the stable region is easy as there are no conflicting writes. However, we show that the real-time order requirement of S requires read-only transactions that are N+O to interact with the most recent writes in the unstable region (Lemma 1). Doing this while ensuring a legal total order requires transferring metadata between the servers (Lemma 2), either proactively through read requests or through the write protocol. By extending this construction, we show that processing a set of read-only transactions requires metadata that is asymptotically larger than the total size of the transactions, regardless of how the metadata is transferred (Lemma 3). This violates C, proving the theorem.

Proof. Suppose the system has two servers, S_1 and S_2, and multiple clients. Let `ALG` be any read-only transaction algorithm that satisfies N+O+S. Let $R = \{r_1, r_2\}$ be a read-only transaction that executes `ALG`, issued by client C_R. Let w_1 and w_2 be simple write requests issued by client $C_w \neq C_R$, where $w_1 \to w_2$ in real-time, i.e., w_2 is sent after the response for w_1 is received. We place no restrictions on the write protocol (beyond assumption A-0). Consider the execution e_1:

$S_1: \quad r_1, w_1$
$S_2: \quad\quad\quad w_2, r_2$

Suppose there is no metadata in the system, i.e., no information for coordinating consistent values between requests.

Lemma 1. *Without metadata, a read-only transaction that is N+O+S must observe any write that precedes it at a server.*

Proof Summary. Without metadata, S_2 cannot distinguish between an execution where w_2 and R are concurrent and one with $w_2 \to R$ in real-time. The latter requires $r_2 \in R$ to observe w_2 to satisfy S's real-time order. ■

Lemma 2. *Processing e_1 while satisfying N+O+S requires dependency $R \to w_1$ to be transferred from S_1 to S_2.*

Proof Summary. Lemma 1 states that, without metadata, r_2 must observe w_2, implying $w_2 \to R$. But r_1 must be processed before w_1 to satisfy N+O, implying $R \to w_1$. Since $w_1 \to w_2$ by construction, this creates a cycle, violating the legal total order of S. Using basic two-party communication complexity, we show that legalizing the total order requires transferring $R \to w_1$ from S_1 to S_2. ■

We now extend e_1 with more read-only transactions, servers, and write requests, and apply the structure above to force more dependency metadata to transfer between servers. We then quantify this metadata and show that it violates C.

Proof of the NOCS Theorem. Suppose the system has $M^2 + 1$ servers $S_1, S_2, \ldots, S_{M^2+1}$. Let $R_1, R_2, \ldots, R_N$ be N read-only transactions that execute `ALG`, where each R_i sends a read request to S_1 and $M - 1$ other servers, such that every server other than S_1 receives N/M read requests. (In practice $M^2 \ll N$, but our construction works for any $N, M \geq 1$.) The specific mapping of read requests to servers is unimportant; we lay them out sequentially by transaction index below. Let $r_{i,j}$ be a read request of R_i assigned to S_j. We assign one read request from each of R_1 to $R_{N/M}$ to S_2, one read request from each of $R_{N/M+1}$ to $R_{2N/M}$ to S_3, and so on, restarting at R_1 after reaching R_N. Let $w_1, w_2, \ldots, w_{M^2+1}$ be M^2+1 simple writes issued to each server by a distinct client C_w that does not issue any read-only transactions. Suppose w_1 precedes all other writes, i.e., $w_1 \rightarrow w_j$ for $j = 2, \ldots, M^2+1$, and all read-only transactions are concurrent with all writes. Consider the execution e_*:

$$
\begin{array}{ll}
S_1: & r_{1,1}, \ldots, r_{N,1}, w_1 \\
S_2: & w_2, r_{1,2}, \ldots, r_{N/M,2} \\
S_3: & w_3, r_{N/M+1,3}, \ldots, r_{2N/M,3} \\
\vdots & \\
S_{M+1}: & w_{M+1}, r_{N-N/M+1,M+1}, \ldots, r_{N,M+1} \\
S_{M+2}: & w_{M+2}, r_{1,M+2}, \ldots, r_{N/M,M+1} \\
\vdots & \\
S_{M^2+1}: & w_{M^2+1}, r_{N-N/M+1,M^2+1}, \ldots, r_{N,M^2+1}
\end{array}
$$

By decomposing this execution into layers, we can inductively quantify the metadata required to process it. Let e_1 be the execution fragment containing all write requests and only the read requests of R_1. Let e_i contain the requests of e_{i-1} plus all read requests of R_i, for $i = 2, \ldots, N$. Thus $e_N = e_*$.

Lemma 3. *Processing e_k while satisfying N+O+S requires $\Omega(kM^2)$ metadata, for $k = 1, \ldots, N$.*

Proof Summary. The proof is by induction. For the base case of e_1, Lemma 2 requires us to transfer $R_1 \rightarrow w_1$ from S_1 to all $M-1$ servers targeted by R_1. We show that the write protocol cannot efficiently transfer this metadata, since it does not know which servers R_1 targets, and hence must send $R_1 \rightarrow w_1$ to all M^2 servers, or $\Omega(M^2)$ metadata. Alternatively, $r_{1,1}$ can convey the list of target servers, but due to asynchrony, a different execution could cause a different target server S_j to play the role of S_1, making it impossible to know which $r_{1,j}$ will appear before a write. Thus, every $r_{1,j}$ must include the list of M servers, requiring $\Omega(M * M) = \Omega(M^2)$ metadata. In the inductive step, we show that e_k cannot rely on previous metadata transferred in e_{k-1}, and thus requires an additional $\Omega(M^2)$ metadata. ∎

Completion of the proof. By Lemma 3, $e_* = e_N$ requires $\Omega(NM^2)$ metadata. Since $R_1, \ldots, R_N$ issue NM read requests total, the amortized metadata required per read request is $\Omega(\frac{NM^2}{NM}) = \Omega(M)$, which is not constant, violating C. ∎

4.2 The Broad Scope of NOCS

We prove NOCS is impossible in the specific setting of one-shot read-only transactions in failure-free systems. When it comes to an impossibility result, the more restricted the setting it is proved in, the stronger the result, because any setting that is more general is also subject to the impossibility result (the general setting *includes* the restricted setting as a special case). Thus, the NOCS Theorem also applies to more general settings, such as those with read-write transactions, multi-shot transactions, and/or failures.

4.3 NOCS Is Tight

While all properties are impossible to achieve together, we find that NOCS is "tight" in the sense that any combination of three properties is possible. Spanner's [10] read-only transactions are one-round, use constant metadata, but block reads in order to return strictly serializable results (O+C+S). Many systems use multiple non-blocking round trips to coordinate strongly consistent results (N+C+S), e.g., DrTM [49], RIFL [29]. To the best of our knowledge, no existing system provides strict serializability in one round of non-blocking communication (N+O+S). We present the design of such a system, PORT-SEQ, and a proof of its correctness in our technical report [35]. The design uses a centralized write sequencer to totally order writes, and requires a linear amount of metadata for read-only transactions. We are aware of two systems that have performance-optimal read-only transactions (N+O+C): MySQL Cluster [39] and the snapshot read API of Spanner. These systems provide weak consistency, however, as we discuss below.

5 NOCS Connects Theory with Practice

This section discusses the value of the NOCS Theorem in understanding the design space and in guiding system designs.

5.1 Theoretical Insights

Proving the impossible. NOCS is philosophically similar to other impossibility results like CAP and SNOW, in that it helps system designers avoid attempting the impossible and instead identifies a fundamental choice they must make: their system can either have performance-optimal read-only transactions or provide strict serializability, but not both.

Identifying the possible. The crux of NOCS's impossibility is that the real-time requirement of strict serializability forces read-only transactions to confront conflicting requests (Lemma 1). This suggests optimal performance could be possible with even *slightly* relaxed consistency models that do not require real-time ordering, and thus can avoid the unstable region. In particular, the second strongest consistency model we are aware of—process-ordered serializability [34]—does not require real-time ordering.

Yet, there is a large gap in the current design space. The only two existing systems whose read-only transactions are performance optimal provide weak consistency. MySQL Cluster's read-committed consistency does not isolate transactions. Spanner's snapshot read API can be used to get performance optimality, but it does not ensure clients see their own recent writes when used in this way (§10). Between these weak guarantees and strict serializability are many stronger consistency models, such as read-atomic [5], causal consistency [31], and process-ordered serializability [34]. We bridge this gap by presenting the PORT design that provides performance-optimal read-only transactions and the strongest consistency to date: PORT provides process-ordered serializability in systems with only simple writes (§6), and it provides causal consistency in systems with write transactions (§8). (We conjecture causal consistency is the upper bound for performance-optimal read-only transactions when transactional writes are present.)

5.2 Guiding System Designs

NOCS is also useful in guiding system designs. First, to make a design performance-optimal, it must satisfy the NOC properties: each transaction must succeed using a single round of non-blocking messages with constant metadata. Therefore, the NOC properties indicate we must avoid validation-based and stabilization-based techniques to satisfy O, avoid techniques based on distributed lock management to satisfy N, and ensure the complexity of processing a read does not depend on the level of contention—i.e., the number of conflicting reads and/or writes—to satisfy C. Second, the NOCS Theorem suggests a path towards designing NOC protocols by avoiding how it derives its impossibility: read-only transactions should always execute on system states outside the unstable region. These implications of the NOC properties and the NOCS proof significantly reduced the design space of algorithm we needed to explore and led us to two high-level techniques for PORT: explicit ordering control and multi-versioning.

Explicit ordering control. There are two methods for ensuring reads avoid the unstable region by explicitly controlling the ordering of concurrent operations. First, reads can request versions of the data that lie before the unstable region begins, which orders a read-only transaction before ongoing writes. Second, servers can reorder operations when a read requests data in the unstable region.

Explicitly controlling ordering is not compatible with strict serializability because the real-time requirement forces a specific ordering of operations (Lemma 1) that cannot be communicated in a performance-optimal system (Lemma 3). Consistency models without the real-time requirement, however, might be compatible with an explicitly controlled ordering while satisfying NOC. PORT confirms this, by using versions clocks to capture this explicit ordering. PORT uses both types of explicit control on top of multi-versioning to provide its consistency guarantees and optimal performance.

Multi-versioning. Enabling reads to control what version of data they request requires multi-versioning on servers. Multi-versioning introduces storage overhead to temporarily keep additional version around, but this overhead is minor as storage is inexpensive and extra versions are not kept long. It also introduces some processing overhead to look up the correct version of data to return, reflected by our C property.

The need for multi-versioning to support efficient reads is not new. The existing performance-optimal systems, Spanner and MySQL Cluster, are multi-versioned. In fact, all existing systems whose read-only transactions are guaranteed to terminate—i.e., have a bounded number of retries and/or bounded blocking—are multi-versioned (Table 9). On the other hand, multi-versioning alone does not ensure optimal performance: most MVCC protocols require either extra on-path messages to query a timestamp oracle [6, 43], off-path messages to compute stable snapshots [3, 15], or blocking reads if the client-provided timestamp in MVTSO-based protocols points to the future [30, 45]. PORT's novelty is in how it uses version clocks to explicitly control ordering by manipulating the multi-versioning framework in order to achieve optimal performance.

6 PORT Design

PORT is a new system design that enables performance-optimal read-only transactions with process-ordered serializability, the strongest consistency to date.

Process-ordered serializability. Process-ordered serializability guarantees there exists a legal total order of transactions that respects the ordering of transactions within each process [34]. It is equivalent to sequential consistency [27] with the addition of transactional isolation. It preserves all the properties of strict serializability (§4) except for the real-time order across processes (clients). That is, it preserves the real-time order within each process, i.e., process order, and a total order across processes, but a client may not see the most recent updates of other clients. *Process order* ensures that each client interacts with the system monotonically, e.g., sees her own recent writes. *Total order* ensures that concurrent transactions are observed by all clients in the same order.

6.1 Version Clocks

This section describes *version clocks* (§6.1), a new specialized logical clock that tightly couples all the components of PORT (§6.2). Version clocks also allow us to avoid the work of applying some writes (*write omission*, §6.3) and limit the staleness of reads (*data freshness*, §6.4).

Version clocks are designed in the context of distributed storage systems and have two features: they ensure process order by concisely capturing the ordering constraints between requests and enable optimal performance by reading at the most recent snapshot in the stable region.

Enforcing process order. Version clocks take advantage of two observations. First, process order is a per-client order, and thus can be explicitly controlled by clients. Second, read and write requests have different semantics, i.e., writes modify system state while reads do not. Therefore, they should be treated differently: it is unnecessary to enforce an order among the read requests that observe the same system state.

Capturing the stable frontier. Version clocks follow the practical guidance of the NOCS Theorem (§5.2) to avoid the

```
1 Client Side
2 versionstamp = 0 # clock value
3 view[]           # max known versionstamp per server
4
5 # Sending requests
6 function get_vs_read():
7   versionstamp = tick(min{view[]}) # stable frontier
8   return versionstamp
9
10 function get_vs_write():
11   versionstamp++
12   return versionstamp
13
14 # Receiving a response msg from server svr
15 function recv_response(maxVS):
16   view[svr] = max{view[svr], maxVS}
17   if msg.for_write is true
18     versionstamp = tick(maxVS)
19   return
20
21 function tick(vs):
22   return max{vs, versionstamp}
23
24 Server Side
25 maxVS = 0 # max seeen versionstamp
26 # ... return maxVS when sending response msg
```

Figure 1: Pseudocode for version clocks.

```
1 Client Side
2 function read_only_txn(<keys>):
3   vs = VersionClock.get_vs_read()
4   for k in keys # in parallel
5     vals[k], maxVS = read(k, vs)
6     VersionClock.recv_response(maxVS)
7   return vals   # replies to end user
8
9 function write(key, val):
10   vs = VersionClock.get_vs_write()
11   maxVS = write(key, val, vs)
12   VersionClock.recv_response(maxVS)
13   return        # replies to end user
14
15 Server Side
16 vers[keys][] # multi-versioned storage
17 function read(key, vs):
18   if vers[key][vs] exists
19     return vers[key][vs], VersionClock.maxVS
20   else # return nearest version to not block
21     near_vs = find_nearest_earlier(ver)
22     # ensure future writes have higher vs
23     vers[key].max_r_vs = max(vers[key].max_r_vs, vs)
24     return vers[key][near_vs], VersionClock.maxVS
25
26 function write(key, val, vs):
27   if vs <= vers[key].max_w_vs
28     return VersionClock.maxVS # omit write
29   if vers[key].max_r_vs >= vs
30     vs = max_r_vs + 1 # commit after promoted versions
31   vers[key][vs] = val
32   vers[key].max_w_vs = vs
33   if vs > VersionClock.maxVS
34     VersionClock.maxVS = vs
35   return VersionClock.maxVS
```

Figure 2: Pseudocode for PORT.

unstable region by capturing the stable frontier. The stable frontier is the most recent snapshot in which all writes are in the stable region. Each server tracks the final versionstamp of its most recent write. A version clock tracks the minimum of such versionstamps across all servers the client has contacted, which is exactly the stable frontier the client knows. Version clocks direct read messages to the stable frontier when possible. PORT takes care of the cases when reads have to confront conflicting requests beyond the stable frontier. "Promotion" is used in systems with simple writes to advance the stable frontier beyond the versionstamp of an incoming read to ensure a total order. "Per-client ordering" is used in systems with write transactions to logically move a client's own writes before the stable frontier so the client can always safely read at the stable frontier (§8.2). Both techniques enforce the necessary order between concurrent reads and writes without blocking either reads or writes.

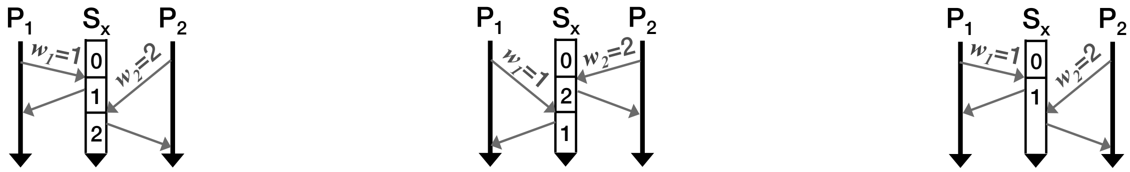
Clock structure. Figure 1 shows the pseudocode of version clocks. *versionstamp* stores the current clock value (line 2), which is embedded in every read/write message to explicitly control their ordering. When versionstamps are the same for two operations of the same type, the server orders them arbitrarily. When versionstamps for a read and a write are the same, the server orders the read after the write. A server responds with the highest versionstamp it has seen (line 26). A client uses *view* to track the highest versionstamps of the servers it has contacted (line 3) and uses them to find the stable frontier (line 7) before sending a read message (lines 6–8). *view* is updated upon receiving a response (line 16). If the response is for a write message, then the clock is advanced so that future read messages will have greater versionstamps than the write (lines 17–18), ensuring read-your-writes. Because versionstamps increase monotonically and reads have non-smaller versionstamps than earlier writes, version clocks preserve process ordering.

6.2 Basic PORT Design

The basic PORT design includes a multi-versioning framework, a read-only transaction algorithm, and a write algorithm. We co-design these components tightly by leveraging version clocks. Figure 2 shows PORT's pseudocode.

Client library. The read-only transaction and write algorithms are executed by a client library. For each read-only transaction or write, the client obtains a versionstamp from its version clock and embeds it in the request message(s). This per-client versionstamp decides which system version on the servers the operation must read (or write) to ensure the client's process order (lines 3, 10). The server-side logic ensures a total order on top of the process order on each client to guarantee process-ordered serializability.

Multi-versioning framework. Servers store written values in a multi-versioning framework (line 16). Since PORT uses version clocks to track the ordering between operations, it is natural and efficient to index the historical values of each data item with versionstamps. In this way, the multi-versioning framework and transaction layer are nicely coupled via versionstamps. We omit a detailed discussion of garbage collection, which uses standard mechanisms similar to those used to provide at-most-once semantics.



(a) orders w_1 before w_2 by arrival. **(b) orders w_2 before w_1 by arrival.** **(c) orders w_2 before w_1 by omission.**

Figure 3: Space-time diagrams showing three executions of writes w_1 and w_2 that are concurrent and conflicting. The value underneath S_X indicates the value stored by the server. Process-ordered serializability allows w_1, w_2 to be ordered either way. This enables us to omit w_2 in (c) because it is equivalent to the ordering in (b), i.e., (w_2, w_1).

Read-only transactions. To process a read request, a server executes it against the system version specified by its versionstamp. Executing a read is thus equivalent to returning the value indexed at versionstamp. If the server has the requested version, then the read is inside the stable region and it returns the version directly (lines 18, 19). Otherwise, it uses promotion to ensure a total order between the read and any concurrent writes at the specified versionstamp, without blocking either the read or write (lines 20–24).

Promotion logically copies the value of the nearest earlier version to all empty positions between that version and the one requested by versionstamp. Logical versions are used as placeholders to ensure a total order: once a version has been read by any client, no earlier versions can be modified to ensure different clients observe them in the same order. For example, if a read request has $vs = 4$ and the data item has committed values at $vs = 1, 2$, the version at $vs = 2$ is the nearest earlier version and is promoted to positions $3, 4$. A conflicting write at $vs = 3, 4$ will be "bumped up" to $vs = 5$ when it arrives. We implement promotion with a single variable (line 23) that marks earlier positions as immutable.

Writes. When receiving a write request, a server finds the position specified by the write's versionstamp in the multi-versioning framework. If the position is empty, then the write is applied at the versionstamp (line 31). If the position has been marked immutable by read promotion, the server finds the next available position to write the version at (lines 29–31). The write protocol also includes a mechanism for safely skipping concurrent writes (lines 27–28), discussed next.

6.3 Write Omission

Write omission is a special conflict resolution mechanism that skips an incoming write if it is concurrent with an already applied write. Omitting a write is desirable because it saves the computation needed to apply it, reduces the number of stored versions, and saves the work of replicating it.

Write omission is safe. Consistency models in general, and process-ordered serializability specifically, allow conflicting writes to be ordered either way. For instance, if two processes concurrently issue w_1 : $\texttt{write}(\texttt{x} = 1)$ and w_2 : $\texttt{write}(\texttt{x} = 2)$, then they can be ordered as either (w_1, w_2) or (w_2, w_1). Typically, systems apply writes in the order that they arrive, e.g., w_1 then w_2. But if instead we use the opposite order, then this is equivalent to omitting w_2, as shown in Figure 3: skipping the later write is equivalent to ordering it before the earlier write and immediately overwriting it with the latter. Write omission does not affect the total order requirement: all clients observe concurrent writes in the same order, because omitted writes are never seen by any client.

Knowing a write is concurrent. Version clocks enable PORT to identify when writes are concurrent, allowing a later concurrent write to be omitted. PORT omits an incoming write if its versionstamp, vs_{omit}, is less than or equal to the highest committed versionstamp of the data item, $vs_{highest}$ (lines 27–29). The write with the highest committed versionstamp cannot have happened-before [26] the omitted write because $vs_{highest} \geq vs_{omit}$. More specifically, version clocks guarantee the invariant: if write x happens-before write y, then $vs_x < vs_y$. The omitted write cannot have happened-before the write with the highest committed versionstamp because it has not happened yet. Therefore, the two writes are concurrent, and it is safe to omit the incoming write.

Omitting a write is equivalent to applying it immediately before the write with the highest versionstamp. A client's future reads must observe the "higher" write if its own write was overwritten in this way. Therefore, the server returns the versionstamp of its highest applied write to the client (line 29), which uses it to update its versionstamp as normal.

6.4 Keeping Reads Fresh

To avoid the unstable region, we must sometimes return values staler than what strict serializability would return (§5.2). PORT limits data staleness in two ways, neither of which incurs extra messages, blocking, or non-constant metadata. That is, they do not forfeit optimal performance (NOC).

Reducing staleness with version clocks. Instead of naively returning versions far behind the stable frontier, version clocks try to track the stable frontier precisely. They use *view* to track the most recent versionstamp on each server a client has contacted, so a client's version clock never ticks slower than the servers it is aware of. This significantly improves the freshness of data requested by read-only transactions.

Reducing staleness via co-location. Many storage systems co-locate "end users" on the same client machine [12, 18, 40], i.e., each client (machine) has many sessions (threads), one per end user. We leverage co-location to help user sessions keep each other fresh by sharing one version clock among them on the same client, which ensures no user session is staler than the freshest session it is co-located with.

6.5 Correctness and Generality

The only technique PORT relies on is version clocks, which can easily be added to systems with existing physical/logical clocks, or implemented from scratch. We demonstrate both by applying PORT to a system without transactions (shown by Scylla-PORT) and a system with existing sub-optimal read-only transactions (shown by Eiger-PORT). We present a proof of correctness for PORT in our technical report [35].

Failures. PORT can tolerate server failures using typical techniques such as state machine replication [46]. To tolerate client—i.e., frontend—failures, clients can send versionstamps back to end-user machines that then include the versionstamp in subsequent requests to the application (e.g., via cookies). This ensures process ordering is maintained even if an end user's later requests go to a different frontend due to load-balancing or frontend failure.

7 PORT Implementation and Evaluation

This section discusses Scylla-PORT, the implementation of PORT on a clean slate base system.

7.1 Implementation

We build PORT on ScyllaDB [47], a clean slate, non-transactional base system that supports only simple reads and simple writes. ScyllaDB is a production system that serves as a drop-in replacement for Cassandra [25] and provides an order-of-magnitude better performance. It is well-engineered and aggressively-optimized for performance, including a new implementation in C++14, core-level sharding that avoids cross-core locking and context switches, and customized lock-free data structures.

Rationale and takeaways. We chose to implement PORT on ScyllaDB for three reasons. First, it stresses the efficiency of PORT: as a highly efficient baseline system, it is sensitive to any additional overheads, and thus amplifies any performance cost introduced by PORT. Second, ScyllaDB is single-versioned. The negligible performance overhead shown in our evaluation includes the cost of making it multi-versioned (§5.2), which shows the efficiency of co-designing the multi-versioning framework and the transaction layer enabled by version clocks. Third, PORT is compatible with all the customized engineering decisions of ScyllaDB, which demonstrates the generality of the design of PORT.

7.2 Evaluation Overview

We evaluate Scylla-PORT against ScyllaDB (the clean slate, non-transactional base system) and Scylla-OCC (an implementation of OCC atop ScyllaDB). We compare their throughput, latency, scalability, and quantify data staleness.

Scylla-OCC. We implemented a variant of OCC optimized for read-only transactions, similar to Rococo's read-only transaction algorithm [37]. It includes an initial round of optimistic reads and then a validation round. If the values read in the optimistic round match the values in the validation round the transaction succeeds. If not, the read-only transaction is aborted and retried. This variant has strictly better performance than traditional distributed OCC because it avoids the need for distributed commit: its best case is two rounds compared to traditional distributed OCC's best case of three rounds (read, validate/prepare, commit).

Code. We implemented our server-side logic in ScyllaDB's codebase (release 2.1-RC3) in C++14 and our client-side logic in the Java Thrift client of the YCSB benchmark (release 0.10.0) [9]. Version clocks are implemented on both servers and clients. Scylla-PORT adds ~1,300 LOC.

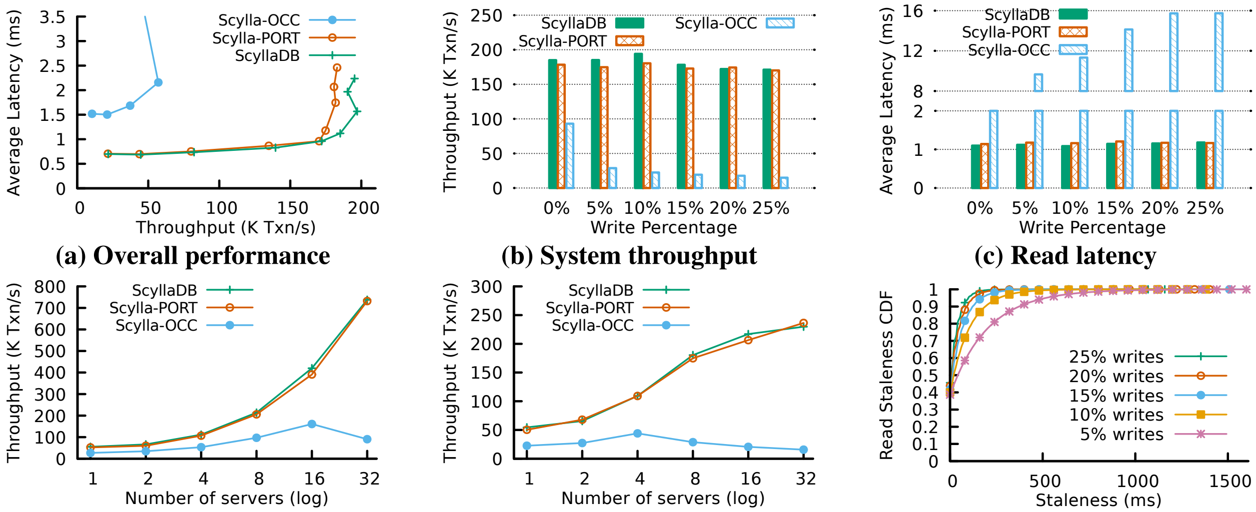
Experimental setting. We run experiments on Emulab [50]. Each machine has two 2.4 GHz 8-Core Xeon CPUs, 64 GB RAM, and a 10 Gbps network interface. We use a single datacenter setting. All experiments, except for scalability tests, use 8 servers loaded by 8 client machines. The scalability tests use up to 64 machines. Each client issues 10 million requests in each experiment, which takes 5–10 minutes to complete, sufficiently long to minimize warm-up and cool-down effects and provide stable results. Experiments are CPU-bound on servers.

Configuration and workloads. We use YCSB's standard workloads B (read-heavy, 95% reads) and C (read-only) with customized read-to-write ratios of up to 25% writes. We use YCSB's default parameters: 1 million records, 10 fields per record, 100 B values per field, and Zipf constant of 0.99. Each request (a read-only transaction or a group of simple writes) accesses 5 records and all fields in each record.

Results summary. Transactional overhead is generally evident with read-write conflicts and under skewed workloads, so we focus our evaluation in such scenarios to amplify Scylla-PORT's cost. Our results show that Scylla-PORT can almost match its performance to that of non-transactional ScyllaDB: 1–3% overhead in throughput and latency in most settings and less than 8% even in the worst case. Scylla-PORT outperforms OCC by an order-of-magnitude in such contended scenarios due to OCC's retries, and outperforms OCC under low contention (OCC's best case) by at least two times. Scylla-PORT scales as well as ScyllaDB and scales better under contention. More than 40% of its reads return fresh values.

7.3 Throughput and Latency

Figure 4a shows the overall performance of the systems as we gradually increase the system load by using more closed-loop client threads. Scylla-PORT has similar performance to the baseline ScyllaDB. Their largest difference before Scyl-



(a) Overall performance **(b) System throughput** **(c) Read latency**

(d) Scalability (uniform) **(e) Scalability (Zipf=0.99)** **(f) Scylla-PORT data staleness**

Figure 4: The performance of Scylla-PORT closely matches non-transactional ScyllaDB and is significantly better than OCC, Scylla-PORT scales even better than ScyllaDB with skewed workloads, and half of its reads return fresh data.

laDB becomes overloaded is evident with 32 client threads: 5.6% in throughput and 5% in latency. All later experiments report throughput and latency at this operating point, i.e., with 32 client threads. OCC initially has latency that is twice that of ScyllaDB and Scylla-PORT because it takes at least two rounds to complete instead of one. As load increases, OCC's latency increases quickly and its throughput decreases slightly because contention forces it to retry.

Varying write percentage. Figure 4b and 4c show the throughput and latency as we vary the read-to-write ratio. Scylla-PORT's throughput is within 4% of ScyllaDB's for five of the experiments and within 7% for the remaining one. Similarly, its latency is within 2% (20 µs) of ScyllaDB's for two of the experiments and within 7% (107 µs) for the other four. As the write percentage increases, the overhead disappears because of write omission: doing slightly more work during reads is offset by doing less work during writes. When there are only reads, Scylla-PORT has double the throughput and half the latency of OCC because OCC's read-only transactions require at least two rounds. With writes, OCC's performance drops quickly due to retries.

7.4 Scalability

Figure 4d compares the scalability of the three systems under a uniform workload as we increase the number of servers while increasing the number of clients to keep the servers CPU-bound. Scylla-PORT scales as well as ScyllaDB; the differences in throughput are negligible. Interestingly, Scylla-PORT outperforms ScyllaDB under a skewed workload, as shown in Figure 4e. ScyllaDB stops scaling at 16 servers because the server holding the hottest keys becomes the bottleneck, and adding more servers does not help. (We have confirmed this finding with ScyllaDB's developers.) Scylla-PORT scales better than ScyllaDB under skewed workloads because it can avoid the work of some writes to the hottest keys due to write omission. Since write omission only applies to conflicting writes, this rarely occurs under a uniform workload. OCC initially shows a similar scaling pattern starting from its lower throughput. OCC's scaling stops, however, as more concurrent clients accessing the same keys lead to higher contention and thus more retries.

7.5 Data Staleness

Figure 4f shows the staleness of Scylla-PORT under a skewed workload with varying write percentages. Staleness is measured relative to strict serializability, which always has a staleness of 0: it is the amount of time since a newer version has been committed. For example, if v_0, v_1 are consecutive versions, v_0 is returned at 0:05, and v_1 committed at 0:00, then the staleness of v_0 is 5 seconds.

Scylla-PORT returns the most recent data ~40% of the time, and 90% of reads return values no staler than 500 ms. Scylla-PORT returns fresher data as the write percentage increases because version clocks advance versionstamps more frequently when there are more writes. Scylla-PORT leverages version clocks to precisely capture the stable frontier, but does not utilize client co-location. Sharing one clock among co-located user sessions would further decrease staleness, but also decreases the rate at which write omission can be used. We leave investigating this tradeoff to future work.

7.6 Low Contention Evaluation

We focused here on high contention workloads because those are where any differences between Scylla-PORT and ScyllaDB would appear. Scylla-OCC did poorly in this setting as is expected because OCC is better suited to low contention settings. We present the results of evaluating the three systems under low contention in our accompanying technical report [35]. Even in that setting, Scylla-PORT significantly outperforms Scylla-OCC with at least double the throughput and at most half the latency because Scylla-PORT always finishes in one round while OCC's best case is two rounds.

8 Improving an Existing System

This section adapts PORT to improve Eiger, an existing system that has both read-only and write transactions.

8.1 Eiger Overview and Rationale

Eiger is a geo-replicated, causally consistent system that has read-only transactions and write transactions. Each machine implements a Lamport clock and attaches a Lamport timestamp to each committed write that is guaranteed to be larger than any earlier write it causally depends on. Eiger's write transaction protocol is a variant of two-phase commit [21, 28] that always commits. Eiger's read-only transaction protocol takes between one and three non-blocking rounds of communication. If there are no concurrent write transactions, it completes in a single round. Otherwise, it requires a second round of messages to a subset of the servers, followed by a third round if the concurrent write transactions are still pending when the second-round requests arrive. In the third round, each read request needs to query the states of all write transactions it conflicts with, and thus the required metadata increases linearly with respect to the number of conflicting write transactions.

Rationale. We choose Eiger as a base system because of its guarantees and the efficiency of its read-only transactions. First, it provides causal consistency, not strict serializability, so it may be possible to add performance-optimal read-only transactions to it. Second, it includes write transactions, which present a new challenge for the PORT design. Third, it is the only system with write transactions and causal (or stronger) consistency that completes read-only transactions in a bounded number of non-blocking rounds of communication (Figure 9). Finally, its read-only transactions often complete in a single non-blocking round, making them a more difficult baseline than other algorithms such as OCC.

8.2 Eiger-PORT

Eiger's read-only transactions are non-blocking, require up to three rounds of on-path communication, and use linear-sized metadata in the third round. We make them performance-optimal by making them always finish in one round using only constant metadata. The major challenge is to ensure write isolation, i.e., return a system state that is either before all updates in a write transaction or after.

More specifically, when a read-only transaction must read beyond the stable frontier, e.g., to ensure read-your-writes, PORT reorders the read-only transaction and the conflicting writes without blocking by using "promotion" (§6.2). However, promotion does not work for Eiger because it cannot ensure that all writes in the same write transaction are promoted at the same time since they can be on different servers. Our solution, *per-client ordering*, enables clients to observe conflicting writes in different orders, as allowed by causal consistency. Specifically, it pulls back any of a client's recent writes that are beyond the stable frontier. This allows the client to read at the stable frontier while also always seeing their own writes. Figures 5, 6, and 7 show the pseudocode, written in a way that favors clarity over efficiency.

```
1  Client Side
2  lst_map[][]   # maps server to its local safe time
3  gst           # global safe time
4
5  function read_only_txn(<keys>):
6    gst = get_read_ts(min{lst_map.valueSet()})
7    for k in keys  # messages in parallel
8      vals[k], lst = read(k, gst, cl_id)
9      lst_map[k.server] = lst # lst is monotonic
10   return vals
11
12 function write_txn(<keys, vals>):
13   for k, v in <keys, vals>  # in parallel
14     if k.server is coord # the coordinator
15       lst = write_coord(k, v, cl_id, gst)
16     else # a cohort
17       lst = write_cohort(k, v, cl_id, gst)
18     lst_map[k.server] = lst # lst is monotonic
19   return
20
21 function get_read_ts(ts):
22   return max{ts, gst}
```

Figure 5: Client-side pseudocode for Eiger-PORT.

```
Server Side (Read-Only Txn)
lst              # local safe time, updated upon writes

function read(k, rts, cl_id):
  ver = DS[k].at(rts) # vers are sorted by commit_t
  for v in DS[k].newer_than(ver.commit_t)
    # ensure read-your-writes, from newer ver to old
    if v.cl_id == cl_id
      return v.val, lst
  if ver.cl_id != cl_id
    return ver.val, lst
  else # ensure write isolation
    v = find_isolated(ver)
    return v.val, lst

function find_isolated(ver):
  # iterate from newer version to old
  while v in DS[k].newer_than(ver.gst)
    and v in DS[k].older_than(ver.commit_t)
    if v.cl_id != ver.cl_id
      return v
    else
      return find_isolated(v)
  return ver
```

Figure 6: Read-only transaction logic for Eiger-PORT.

Client-side logic. Figure 5 shows the client-side logic. Each client maintains two variables (lines 2, 3). *lst_map* tracks the local safe time, *lst*, of each server. Global safe time, *gst*, is the minimum *lst* across all servers (line 6) and advances monotonically. *gst* is used as the read timestamp for each

```
1  Server Side (Write Txn)
2  lst            # local safe time
3  pending_wtxns   # uncommitted write txns
4  DS[][]  # multi-versioned k-v data store
5
6  function write_coord(k, v, cl_id, gst):  # coordinator
7    # PREPARE
8    ver, prepared_t = prepare_write(k, v, cl_id, gst)
9    # ... get yes-vote-msgs from all cohorts
10   # COMMIT
11   commit_t = max{yes-vote-msgs.prepared_t, prepared_t}
12   commit-msg = {"commit", commit_t}
13   # ... send commit-msg to all cohorts
14   commit_write(ver, commit_t)
15   return lst
16
17 function write_cohort(k, v, cl_id, gst):  # cohort
18   # PREPARE
19   ver, prepared_t = prepare_write(k, v, cl_id, gst)
20   yes-vote-msg = {"yes", prepared_t}
21   # ... send yes-vote-msg to coordinator
22   # ... wait for commit-msg
23   # COMMIT
24   commit_t = commit-msg.commit_t
25   commit_write(ver, commit_t)
26   return lst
27
28 function prepare_write(k, v, cl_id, gst):
29   pending_t = LamportClock.current()
30   pending_wtxns.append(pending_t)
31   LamportClock.advance()
32   ver = DS[k].create_new_ver(v, cl_id, gst, pending_t)
33   ver.is_pending = true
34   return ver, LamportClock.current()
35
36 function commit_write(ver, commit_t):
37   ver.commit_t = commit_t
38   ver.is_pending = false
39   pending_wtxns.remove(ver.pending_t)
40   if pending_wtxns is empty
41     lst = LamportClock.current()
42   else
43     lst = pending_wtxns.head() # min of pending_wtxns
44   return
```

Figure 7: Write transaction logic for Eiger-PORT.

read-only transaction. Both *lst* and *gst* are Lamport timestamps as used in Eiger. A client sends all read requests in a read-only transaction in parallel. Each read request includes the key, the read timestamp *gst*, and the unique identifier of this client (line 8). The server responds with the requested value and *lst* on that server. A client issues a write transaction by sending the write requests in parallel (lines 12–19). One server is randomly chosen as the coordinator (line 14) for 2PC with the others as cohorts. Each write request contains the key, the value, the client ID, and the client's current *gst* (lines 15, 17). *gst* specifies the stable frontier this write transaction causally depends on. The client updates *lst_map* after each read/write request (lines 9, 18).

Write transactions. Figure 7 shows the server-side logic of write transactions. When a server receives a write request, it records the current Lamport time (line 29) and creates a new pending version (lines 8, 19, 32, 33). *pending_wtxns* tracks ongoing write transactions by keeping an ordered list of *pending_time*s. The running minimum of *pending_wtxns* is the *lst* on this server, i.e., no pending writes exist before *lst*. Because Lamport clocks advance monotonically, insertion, removal, and fetching the minimum of *pending_wtxns* have a cost of $O(1)$. At the end of the "prepare" phase of 2PC, each cohort sends a yes-vote message to the coordinator, which includes the *prepared_time* of this pending write transaction. *prepared_time* is guaranteed to be greater than *pending_time* by clock ticking (line 31).

To commit a write transaction, the coordinator calculates the commit time by taking the maximum across all *prepared_time*s (line 11) and then sends a commit message to the cohorts and commits its local pending version (lines 13, 14). When a cohort receives the commit message, it commits its local pending version (lines 25, 38) with the commit time (lines 24, 37). It then removes this write transaction's *pending_time* from *pending_wtxns* and updates *lst* (lines 39–43). The server returns its *lst* to the client upon commit. Eiger-PORT made minimum changes to Eiger's write transactions, i.e., the management of *pending_wtxns*.

Read-only transactions. Figure 6 shows the server-side logic of read-only transactions. When a server receives a read request, it finds the version at the read timestamp, *rts* (line 5), and checks if the same client has made a recent write later than *rts*. It returns the most recent write by the same client to ensure read-your-writes (lines 6–9). If the version at *rts* was written by the same client, then we need to ensure write isolation by checking whether there exist any versions between the version's *gst*, which is the snapshot time the version depends on, and the version's *commit_t* (lines 18, 19). If there exists such a version written by a different client, then that version is returned to satisfy write isolation (lines 20, 21). We need to do this recursively, but our implementation uses a loop instead for better performance. To ensure write isolation (lines 16–24), we go through the multi-versioned data store once, which has the same cost as finding a particular version by timestamp in other algorithms, e.g., MVCC.

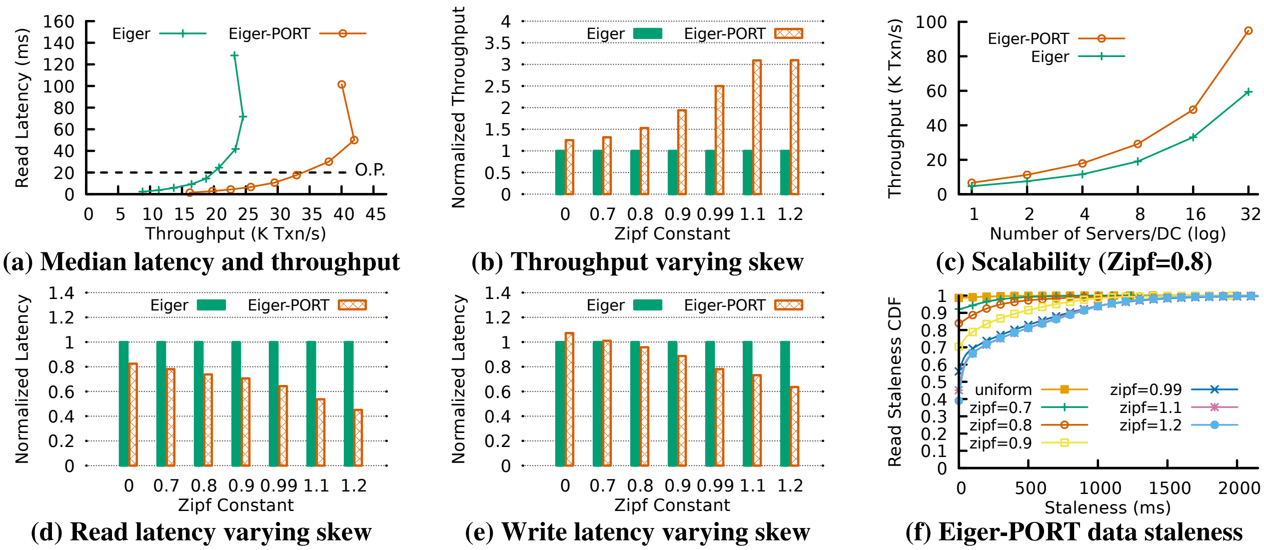
Correctness. We show the correctness of Eiger-PORT by proving that any execution in Eiger-PORT satisfies the causal ("happened before") relation [26] and write isolation for write transactions. We present the full proof in the technical report [35].

9 Eiger-PORT Evaluation

We evaluate Eiger-PORT against Eiger, showing its throughput and latency improvement as well as its data staleness.

Implementation. We implemented Eiger-PORT as a modification to Eiger's code base, which is built on top of Cassandra [25] and written in Java. Eiger-PORT adds ~1000 LOC.

Experimental setting. We try to match Eiger's original experimental setup. We run all experiments on Emulab [50], similar to the now-decommissioned PRObE testbed [19] Eiger used. Each machine has one 2.4 GHz Quad-Core Xeon CPU, 12 GB RAM, and a 1 Gbps network interface. We run 5 trials for each data point, each lasting 65 seconds, and report the median. We exclude the first and last 15 seconds to



(a) Median latency and throughput (b) Throughput varying skew (c) Scalability (Zipf=0.8)

(d) Read latency varying skew (e) Write latency varying skew (f) Eiger-PORT data staleness

Figure 8: Throughput, latency, scalability, and staleness of Eiger-PORT: up to 3× throughput improvement and 60% latency reduction compared to Eiger, better scalability, and low data staleness. All latencies are median latencies.

avoid artifacts due to warm-up, cool-down, and imperfectly synchronized clients. All experiments are CPU-bound.

Configuration and workloads. We use the same setting as Eiger: two logical datacenters co-located in the testbed. Each datacenter has eight server machines, and uses eight client machines to load the servers. The second datacenter is used as a replica, which applies updates replicated from the first datacenter. We use the dynamic workload generator from Eiger with the same default values: 1 million keys, 128-byte values, 5 columns per key, 5 keys per operation, and a write percentage of 10% unless otherwise specified. We also use a Zipf traffic generator with a default value of 0.8.

9.1 Performance Improvement

Results summary. Eiger-PORT significantly improves the performance of Eiger under different workloads, without degrading write performance: 2× and 3× throughput improvement under mild and high skew, respectively, and 20%–60% latency reduction. The performance improvement comes from Eiger-PORT's fewer on-path messages and less metadata to process. The improvement is larger in contended workloads because Eiger is more likely to require more than one round and more metadata in the third round when there are more conflicting write transactions.

Throughput improvement. Figure 8a shows the median read latency and system throughput as we double the number of closed-loop client threads loading the system (from 2 to 512). It shows that Eiger-PORT performs strictly better than Eiger: it achieves higher throughput with the same latency and lower latency with the same throughput. We run all other experiments in Figure 8 with 32 threads, representing an operating point with reasonably low latency (< 20 ms), i.e., at line "O.P." in Figure 8a. The improvements are more pronounced at higher loads. Figure 8b shows normalized throughput with different skew; the improvement stops increasing after Zipf value 1.1, where a single server becomes the bottleneck. Figure 8c shows Eiger-PORT scales better than Eiger due to fewer messages in the system.

Latency improvement. Figure 8d shows the normalized median read latency as we vary skew. Eiger-PORT achieves 20% lower latency under uniform workloads and up to 60% lower latency under contended workloads. Figure 8e shows that Eiger-PORT achieves lower write latency even though we did not intentionally improve writes. The lower latency comes from less queuing delay for writes because reads are faster and there are fewer messages in the system. This demonstrates that PORT can make read-only transactions performance-optimal without making writes more costly.

9.2 Data Staleness

Figure 8f quantifies the read staleness in Eiger-PORT. Staleness is measured relative to strict serializability as in Scylla-PORT's evaluation. Even with high skew, over 40% of Eiger-PORT's read-only transactions return up-to-date values, and over 90% of reads experience less than 1 s staleness. Eiger-PORT tends to return staler data than Scylla-PORT because the stable frontier moves more slowly in Eiger/Eiger-PORT: write transactions take longer to commit than simple writes.

10 Related Work

This section examines existing read-only transactions with the NOCS Theorem, reviews impossibility results, and discusses the move from latency to performance optimality.

Bridging the gap in the design space. We use the NOCS Theorem as a lens to better understand existing systems and show a set of representative systems in Figure 9. We find

System	N	O	C	S	W
		Performance-optimal			
Scylla-PORT *	✓	✓	✓	POS	×
Eiger-PORT *	✓	✓	✓	Causal	✓
Spanner-Snap [10]*	✓	✓	✓	SR	✓
MySQL Cluster [39]*	✓	✓	✓	RC	✓
		One fewer performance property for stronger guarantees			
Spanner-RO [10]*	×	✓	✓	✓	✓
DrTM [49]*	✓	≥ 1	✓	✓	✓
RIFL [29]	✓	≥ 2	✓	✓	✓
Sinfonia [1]	✓	≥ 2	✓	✓	✓
		Candidates for improvement in performance and/or guarantees			
TAPIR [51]*	×	✓	✓	Ser	✓
Pileus-Strong [48]	×	2	✓	✓	✓
Rococo-SNOW [34]*	×	✓	Linear	✓	✓
COPS-SNOW [34]*	✓	Off-path	Linear	Causal	×
COPS [31]*	✓	≤ 2	Linear	Causal	×
RAMP-F|H [5]*	✓	≤ 2	Linear	RA	✓
RAMP-S [5]*	✓	2	✓	RA	✓
Eiger [32]*	✓	≤ 3	Linear	Causal	✓
Janus [38]	×	≤ 2	Linear	✓	✓
Callinicos [41]	×	2	Linear	✓	✓
Occult [36]	✓	≥ 1	✓	PC-PSI	✓
Rococo [37]*	×	≥ 2	✓	✓	✓
Contrarian [13]*	✓	2	✓	Causal	×
GentleRain [15]*	×	≤ 2 + off-path	✓	Causal	×
Cure [3]	×	Off-path	✓	Causal	✓
MVTSO [30, 45]	×	✓	✓	Ser	✓

Figure 9: A review of existing systems through the lens of NOCS. Asterisks denote specialized read-only transaction algorithms. ***W*** **denotes write transactions.**

a large gap in the design space. The only existing systems that have performance-optimal read-only transactions provide weak consistency (§4.3). MySQL Cluster [39] provides read-committed, which does not isolate transactions. Spanner's snapshot reads API [10] cannot always guarantee non-blocking read-your-writes. Suppose a client updates key k in a read-write transaction with commit timestamp ts, and then immediately performs a read-only transaction involving a set S of keys that includes k. To ensure read-your-writes, the client must use a timestamp greater than or equal to ts for its read-only transaction. But doing so may block since other keys in S may be involved in a read-write transaction that is in the midst of two-phase-commit with a commit timestamp less than ts. That is, Spanner must use its externally consistent read-only transaction API, which may block reads in such cases to ensure read-your-writes.

We bridge this gap in the design space with PORT, the first design that provides performance-optimal read-only transactions and the strongest consistency to date.

Other read-only transactions. Some systems choose to trade one performance property for stronger guarantees [1, 10, 29, 49] but still reside on the "tight boundary" of the NOCS Theorem. Many systems neither are performance-optimal nor provide the strongest possible guarantees [3, 5, 13, 15, 31, 32, 34, 36], and thus could potentially be improved by our PORT design.

Impossibility results. Our NOCS Theorem is philosophically similar to other impossibility results, e.g., FLP [17], CAP [7, 20], and SNOW [34], in that it saves system designers' effort from trying the impossible. The most relevant result is the SNOW Theorem, which we discuss next.

The move from latency to performance. SNOW [34] showed tradeoffs in the design space of read-only transactions with a focus only on latency. It proved optimal latency is impossible if the system is strictly serializable and has write transactions. This work aims for a more complete understanding of the tradeoffs in the design of read-only transactions by considering latency and throughput. The move from latency to performance has two takeaways.

First, optimal latency neither translates to nor forfeits optimal throughput. The former is shown by the two systems built with SNOW, which provided lower latency at the cost of lowering throughput. The latter is shown by our new designs that achieve both optimal latency and optimal throughput. What really matters is a complete understanding of the tradeoff between performance and consistency and its insights for designs—the major contributions of this work.

Second, higher demand for performance, e.g., the move from latency only to both latency and throughput, suggests higher difficulty in providing stronger guarantees. Optimal latency is possible in strictly serializable systems without write transactions, but optimal performance is not.

11 Conclusion

Distributed storage systems are a fundamental building block of large-scale web services. They rely on read-only transactions to provide consistent views of sharded data. Our NOCS Theorem proves that read-only transactions cannot have optimal performance in strictly serializable systems. We presented PORT, a performance-optimal read-only transaction design that provides the strongest consistency to date. We applied PORT to design Scylla-PORT and Eiger-PORT. Scylla-PORT has minimal performance overhead compared to its non-transactional baseline. Eiger-PORT significantly improves the performance of its transactional base system.

Acknowledgments

We would like to thank our shepherd, Jinyang Li, for her invaluable feedback that improved this work. We thank the anonymous reviewers for their careful reading of our paper and their many insightful comments and suggestions. We are also grateful to Christopher Hodsdon, Theano Stavrinos, and Jeffrey Helt for their feedback on earlier stages of this work. Our evaluation at scale was made possible by the Emulab testbed. This work was supported by NSF award CNS-1824130 as well as a gift from Microsoft Research.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2007.

[2] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: scalable SQL storage for Web applications. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, Jun 2016.

[4] S. Almeida, J. Leitao, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on chain replication. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, Apr 2013.

[5] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *ACM Special Interest Group on Management of Data (SIGMOD)*, Jun 2014.

[6] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB journal*, 23(6):987–1011, 2014.

[7] E. A. Brewer. Towards robust distributed systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, Jul 2000.

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, Jun 2013.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, Jun 2010.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. F. andSanjay Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2012.

[11] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (ATC)*, Jun 2012.

[12] Developer Blog. Twemproxy: A fast, lightweight proxy for memcached. `https://blog.twitter.com/developer/en_us/a/2012/twemproxy.html`, 2012.

[13] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. Causal consistency and latency optimality: friend or foe? In *International Conference on Very Large Data Bases (VLDB)*, Aug 2018.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, Oct 2013.

[15] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, Nov 2014.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[18] H. Fugal, A. Likhtarov, R. Nishtala, R. McElroy, A. Grynenko, and V. Venkataramani. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. `https://engineering.fb.com/core-data/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments/`, 2014.

[19] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, June 2013.

[20] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[21] J. N. Gray. Notes on database systems. IBM Research Report RJ2188 (Feb.1978), 1978.

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

[23] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. In *Proceedings of the VLDB Endowment (PVLDB)*, Aug 2008.

[24] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[25] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 1979.

[28] B. Lampson and H. Sturgis. Crash recovery in a distributed storage system. Xerox Palo Alto Research Center, 1979.

[29] C. Lee, S. J. Park, A. Kejriwal, S. Matsushitay, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[30] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR)*, Jan 2015.

[31] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2011.

[32] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2013.

[33] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[34] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[35] H. Lu, S. Sen, and W. Lloyd. Performance-optimal read-only transactions (extended version). Technical Report TR-005-20, Princeton University, Computer Science Department, 2020.

[36] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar 2017.

[37] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2014.

[38] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[39] MySQL. MySQL :: MySQL Cluster CGE. `https://www.mysql.com/products/cluster/`, 2016.

[40] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2013.

[41] R. Padilha, E. Fynn, R. Soulé, and F. Pedone. Callinicos: Robust transactional storage for distributed data structures. In *USENIX Annual Technical Conference (ATC)*, Jun 2016.

[42] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[43] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2010.

[44] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2018.

[45] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems (TOCS)*, 1(1):3–23, 1983.

[46] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Surveys*, 22(4), Dec. 1990.

[47] ScyllaDB. ScyllaDB :: Scylla Is Next Generation NoSQL. `http://www.scylladb.com/`, 2018.

[48] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating System Principles (SOSP)*, Nov 2013.

[49] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[50] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec 2002.

[51] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[52] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wPerf: generic Off-CPU analysis to identify bottleneck waiting events. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2018.

A Artifact Appendix

A.1 Abstract

This appendix presents the steps for installing Eiger-PORT and running experiments that compare the performance of Eiger-PORT and its base system, Eiger. Eiger-PORT is implemented as a modification to Eiger's code base, which is built on top of Cassandra and written in Java. The experiments evaluate latency, throughput, and scalability. The results are expected to show that Eiger-PORT outperforms Eiger in all experiments and the performance advantages become more significant under more skewed workloads. Eiger-PORT's better performance comes from its performance-optimal read-only transactions.

A.2 Artifact check-list

- **Hardware:** 2.4 GHz Quad-Core Xeon CPU, 12 GB RAM, 1 Gbps network interface
- **Metrics:** latency, throughput, scalability
- **Expected experiment run time:** 10–20 hours
- **Public link:** `http://github.com/princeton-sns/Eiger-PORT.git`

A.3 Description

A.3.1 How to access

The code base of Eiger-PORT is publicly accessible on Github at `http://github.com/princeton-sns/Eiger-PORT.git`. It includes a README file that provides step-by-step instructions on how to set up the environment and run experiments.

A.4 Installation

Please clone the code repository under a clean directory on a machine. The scripts in the package will work seamlessly if the repository is cloned under */local*. The required dependencies can be installed by simply running the bash file *install-dependencies.bash*. Apache Ant is used to build the source code. Both the system files and the stress tool need to be compiled. Please see the README file in the repository for more details.

A.5 Experiment workflow

Running experiments as described in the paper requires setting up two clusters with each having 8 servers and 8 clients. One cluster is the active cluster for processing transactions and the other cluster is used as a replica, which passively receives replicated writes from the active cluster. One extra machine is needed for the control node. Therefore, to create an 8-server-8-client environment, 33 machines are needed in total (2 clusters, 16 machines in each, and 1 control node).

When the experiment topology is determined, the configuration files under the directory *vicci_dcl_config* need to be modified accordingly. All the scripts used to run experiments are under the directory *eval-scripts*. Experiments can be launched by executing *latency_throughput.bash*. The experimental parameters, such as Zipfian constant and read-to-write ratio, are specified in the file *dynamic_defaults*. For details, please see the README file.

A.6 Evaluation and expected result

The results of each experiment are stored under the directory *experiments/dynamic*. Throughput numbers are shown in the file *combined.graph*. A set of latency processing scripts are provided under the directory *data_proc_scripts*. Eiger-PORT is expected to have ~2*X* higher throughput and ~50% latency compared to Eiger. The performance advantages of Eiger-PORT are expected to become more significant under more skewed workloads.

A.7 AE Methodology

Submission, reviewing and badging methodology:

- `https://www.usenix.org/conference/osdi20/call-for-artifacts`