The SNOW Theorem and Latency-Optimal Read-Only Transactions

Haonan Lu, *University of Southern California;* **Christopher Hodsdon,** *University of Southern California;* **Khiem Ngo,** *University of Southern California;* **Shuai Mu,** *New York University;* **Wyatt Lloyd,** *University of Southern California*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lu

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

The SNOW Theorem and Latency-Optimal Read-Only Transactions

Haonan Lu[⋆], Christopher Hodsdon[⋆‡], Khiem Ngo[⋆], Shuai Mu[†], Wyatt Lloyd[⋆]
[⋆]*University of Southern California,* [†]*New York University*

Abstract

Scalable storage systems where data is sharded across many machines are now the norm for Web services as their data has grown beyond what a single machine can handle. Consistently reading data across different shards requires transactional isolation for the reads. Yet a Web service may read from its data store hundreds or thousands of times for a single page load and must minimize read latency to keep response times low. Examining the read-only transaction algorithms for many recent academic and industrial scalable storage systems suggests there is a tradeoff between their power—expressed as the consistency they provide and their compatibility with other types of transactions—and their latency.

We show that this tradeoff is fundamental by proving the SNOW Theorem, an impossibility result that states that no read-only transaction algorithm can provide both the lowest latency and the highest power. We then use the tight boundary from the theorem to guide the design of new read-only transaction algorithms for two scalable storage systems, COPS and Rococo. We implement our new algorithms and then evaluate them to demonstrate they provide lower latency for read-only transactions and to understand their impact on overall throughput.

1 Introduction

Scalable data stores are a fundamental building block of large-scale systems, such as modern Web services. Spreading data across machines—i.e., sharding—allows the system to scale its capacity and throughput, but also complicates how programs and users interact with the data. When all the data is on a single machine, consistently updating that machine is sufficient to ensure reads are consistent. When data is spread across machines, consistently updating the data store is no longer sufficient because reads to different shards will arrive at different times and thus see different views of the data store.

[‡]Work partially done as a student at Rutgers University-Camden.

Consistently viewing data thus requires transactional *isolation*, where reads to different shards either all observe a given update or none do. General transactions provide isolation, but are heavyweight and complex, especially for transactions that do not update data. Thus, it is common to have a special algorithm for *read-only transactions*, which are transactions the system knows will only read data. The importance of these read-only transaction algorithms has been recognized by many recent systems [5, 8, 11, 12, 14, 26, 27, 29, 31, 37, 38].

Read-only transaction algorithms ensure isolation, but often incur overhead relative to simple inconsistent reads of the same data. This overhead stems from extra rounds of communication to find a consistent view, extra metadata to determine if a view is consistent, and/or blocking operations until a consistent view is found. The overhead of these algorithms is important because many real-world workloads are dominated by reads and thus read performance determines the performance of the overall system. For instance, 99.8% of the operations for Facebook's distributed data store TAO are reads [10]. The latency of these reads is especially important because, as Facebook has reported, "a single user request may result in thousands of subqueries, with a critical path that is dozens of subqueries long" [4].

This breadth of reads and especially the depth of sequential reads for a single page load make their latency critical to the response times of the Web services. These response times are aggressively optimized because they affect user engagement and revenue [13, 24, 34]. Thus, one way to improve upon existing scalable storage systems is to decrease the latency of their read-only transactions. But instead of simply making them faster, we seek to make them *latency-optimal*, i.e., as fast as possible.

When examining existing systems we were able to derive latency-optimal read-only transaction algorithms for some, but not all of them. Investigating the cause of this dichotomy led us to discover a tradeoff between the latency and the power of read-only transactions.

We prove this tradeoff is fundamental with the SNOW

Theorem, which states it is impossible for a read-only transaction algorithm to provide all four desirable properties: **S**trict serializability, **N**on-blocking operations, **O**ne-response from each shard, and compatibility with conflicting **W**rite transactions. The power-related properties are strict serializability—which is the strongest form of consistency—and compatibility with conflicting write transactions—which indicates what other types of transactions are in the system. The latency-related properties are non-blocking operations—which ensures each shard immediately handles each read request—and one response—which ensures a single round of messages with the minimal amount of data.

The intuition of the proof is that when a transaction with writes commits there is a point at every server when the transaction becomes visible, but that the asynchronous nature of the network allows read requests to arrive before the transition on one server and after the transition on another. To cope with this possibility, a read-only transaction algorithm must either settle for consistency weaker than strict serializability (S), block some read requests to avoid the inconsistent interleaving (N), coordinate and/or retry the reads (O), or preclude the possibility of conflicting write transactions (W).

The SNOW Theorem is similar to the CAP Theorem [9, 18] in that it helps system designers avoid trying to achieve the impossible and identifies a fundamental choice they must make when designing their system. In addition, we make the SNOW Theorem even more useful by demonstrating what is possible. We show the four properties are tight by describing algorithms that provide each combination of them. We further tighten this boundary by moving beyond considering the properties as binary to instead viewing them as spectrums. For instance, we show that while strict serializability is impossible with the other three properties, an only-slightly weaker consistency model we call process-ordered serializability is possible. We call algorithms that touch the boundary of what is possible *SNOW-optimal*.

Using the lens of SNOW-optimality we can examine scalable data stores to determine if and what room for improvement in their read-only transactions exists. We find room for improvement in many systems, and focus in particular on two recent and quite different data stores, COPS [26] and Rococo [29]. COPS is scalable, geo-replicated, causally consistent, and has only read-only transactions and single-key write operations. In contrast, Rococo is scalable, designed for a single datacenter, strictly serializable, and has general transactions.

We present the design, implementation, and evaluation of novel read-only transaction algorithms for COPS and Rococo. We call the resulting systems COPS-SNOW and Rococo-SNOW. The key insight common to the systems is that to make reads as fast as possible we need to shift as much coordination overhead as possible into writes.

Our evaluation of COPS-SNOW shows that it almost always provides lower latency for read-only transactions and improves latency more as contention increases at the cost of lower overall throughput. Our evaluation of Rococo-SNOW shows that it always achieves lower latency for read-only transactions and has much higher throughput in the high-contention online transaction processing workloads Rococo is designed for, at the cost of slightly lower throughput under low contention.

The contributions of this paper include:

- The SNOW Theorem, which proves there is a fundamental tradeoff between the power and latency of read-only transaction algorithms. This paper also contributes algorithms that show the tightness of the SNOW Theorem and the precise boundary of what is possible, which we characterize as SNOW-optimality.
- The design and implementation of novel read-only transaction algorithms for both the COPS and Rococo scalable data stores that are latency-optimal and SNOW-optimal, respectively.
- Evaluations of COPS-SNOW and Rococo-SNOW that explore their effect on the latency and throughput of the systems under a variety of settings.

Section 2 presents necessary background and Section 3 explains the SNOW properties. Section 4 gives the statement and proof of the SNOW Theorem, shows its tightness, and explores SNOW-optimality. The designs of COPS-SNOW and Rococo-SNOW are presented in Section 5 and then evaluated in Section 6. Section 7 discusses related work and Section 8 concludes.

2 Background

Web services are typically built using two distinct tiers of machines: a frontend tier and a storage tier. The frontend tier is stateless and handles requests from users by executing application code that reads and writes data from the stateful storage tier. The Web service is typically replicated across multiple datacenters, but we restrict our discussion here to a single datacenter for simplicity.[1] The storage tier shards its data across many machines.

Figure 1 shows how a simple page is generated in a Web service. A frontend machine receives a request from a user and then runs the application logic to generate her page by reading data across many shards in the storage tier. All of the reads must complete before the page can be returned to the user. A typical page load issues hundreds or thousands of reads [4]. Many of these reads can be issued in a parallel batch like the reads of a and b. However, some reads are dependent on earlier reads, i.e.,

[1] Our results are magnified in cross datacenter settings.

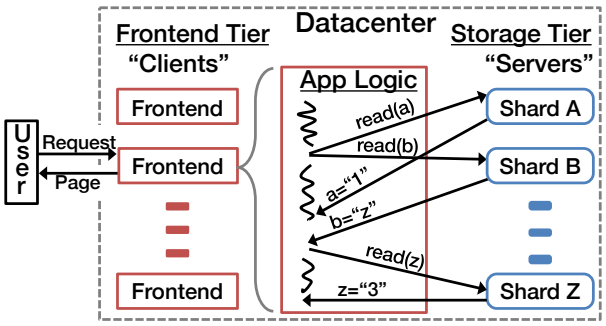


Figure 1: Typical Web service architecture with a frontend machine executing application logic to generate a page that reads data from the storage tier.

they read from keys that are returned by earlier reads. In Figure 1 the read of z depends on the read of b that returns z. Within a page load there are often chains of key dependences that are dozens of reads deep [4].

Because of the breadth and especially depth of these reads, providing low read latency is essential to enabling fast page load times. In the rest of this paper we focus on *one-shot* [20] read-only transactions that do not include key dependences that cross shards. One-shot read-only transactions can be issued in a single parallel batch, e.g., the reads of a and b. Following key dependences requires *multi-shot* read-only transactions. We focus on one-shot transactions because they are simpler to reason about and their results generalize: what is not possible for one-shot read-only transactions is also not possible for the more general multi-shot read-only transactions.

In this paper we consider read-only transactions generally, but with the current motivation of application logic on frontend machines consistently reading data from the storage tier to generate pages. To match general terminology on read-only transactions we call the frontend machines the *clients* and the storage tier the *servers*.

3 The SNOW Properties

This section introduces the SNOW properties and explains their importance for read-only transactions.

3.1 Strict Serializability

Strict serializability ensures there exists a total order over all the transactions in the system—i.e., transactions are *serializable*—and their results appear to have come from a single machine processing them one at a time [32]. This latter requirement ensures the total order respects the *real-time ordering* of transactions [19]. That is, if transaction t_2 begins in real time after transaction t_1 has completed, then t_2 will appear after t_1 in the total order. When two transactions are concurrent there is no real-time ordering between them. For instance, if t_4 begins after t_3 begins but before t_3 has finished, then they are concurrent and either could be ordered first in a legal total order. The total order requirement of strict serializability guarantees that transactions are fully isolated, i.e., a transaction does not observe partial effects of other transactions. Informally, the real-time ordering requirement of strict serializability guarantees a read-only transaction always returns the most recent values.

Strict serializability is the most desirable consistency model because it provides the strongest guarantees. It is easiest for programmers to write correct application logic on top of a strictly serializable system, and it eliminates the most user-visible anomalies [28] compared to other consistency models. Section 4.4 discusses weaker consistency models.

3.2 Non-Blocking Operations

We define *non-blocking operations* to require that each server can handle the operations within a read-only transaction without blocking for any external event. That is, a process involved in handling a read-only transaction never voluntarily relinquishes a processor. Blocking behaviors that are prohibited include waiting for a lock to be available, waiting for messages from other servers, waiting for messages from other clients, or waiting for a timeout to fire. In contrast, non-blocking behavior ensures a server can immediately process and respond to requests from clients. Non-blocking operations are desirable because they directly relate to the latency of the read-only transactions; they save at least the time that would be spent blocking.

3.3 One Response Per Read

We define *one response* per read to be the combination of one round-trip to each server and one version per read. The *one version* subproperty requires that servers send only one value for each read. The *one round-trip* subproperty requires the client to send at most one request to each server and the server to send at most one response back. (This allows for zero messages to and from some servers, for instance, if they do not store data being read.)

The one version subproperty aligns with the latency of read-only transactions. If a server sends multiple versions of a value, that much more time is spent serializing, transmitting, and deserializing the values. The one round-trip subproperty strongly aligns with the latency of read-only transactions. For instance, an algorithm that takes two round trips will take roughly twice as long in transmission and queuing. This subproperty also disallows algorithms that abort a transaction and then start

over, because starting over is another round trip. We explore multi-round algorithms further in Section 4.4.

The one response property is desirable because it leads to faster read-only transactions. We call algorithms that provide the one response and non-blocking properties *latency-optimal* because they are as fast as possible: a client sends a single request to each server, each server handles the request immediately, and the servers send back exactly the data the client wants to read.

3.4 Write Transactions that Conflict

We define the *write transactions* property as the ability of a read-only transaction algorithm to coexist with conflicting transactions that update data. This requires, first, that a data store allows transactions that update data. General transactions that read and write data satisfy this requirement, as do weaker write-only transactions that only write data. This property also requires that write transactions can *conflict* with read-only transactions, i.e., write transactions can update data spread across multiple servers concurrently with read-only transactions viewing that data. The ability to coexist with conflicting write transactions is desirable because write transactions make programming application logic much easier.

4 The SNOW Theorem

The SNOW Theorem is an impossibility result that states no read-only transaction algorithm can provide all of the SNOW properties. This section presents a proof of the SNOW Theorem, discusses the tightness of the theorem, defines SNOW-optimality, and discusses the spectrums of related properties.

4.1 Models, Definitions, and Assumptions

System Model. Our system model is similar to that used in FLP [16]. A distributed system is modeled by a set of N processes, where $N > 1$. Processes communicate by sending and receiving messages. A set of client processes (machines) issue requests to the server processes (machines), which store the data. System actions are modeled as each process going through a sequence of events, where an event is an atomic step of receiving a message, doing local computation, and/or producing a set of output messages.

Network Model. The SNOW Theorem holds for the asynchronous network [18] and the partially synchronous network [15] models. In an asynchronous network, there are no physical clocks and messages between processes can be arbitrarily delayed. In a partially synchronous network, the message delay is bounded and there is a bound on the drift rate between clocks at different processors, but either the rates are not known apriori or do not hold immediately. In the proof, we use an asynchronous network for simplicity. We then discuss the correctness of the SNOW Theorem under the partially synchronous network model.

Definitions. A transaction is a set of operations that read and/or update data. Clients group all operations that are sent to the same server into a single request. The *invocation time* of the transaction is the time when a client process sends each request in the transaction to the involved servers. The *response time* of the transaction is the time when the client has received all the responses from the servers.

Lamport's *happened-before* relation [21] is the (smallest) partial order such that "1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. 2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. 3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$."

We use the happened before relation to differentiate between two server behaviors for handling read-only transaction requests. Let r be the handling of a request from a read-only transaction, R, on a server. A write transaction, W, is *unknown* to r if the client that issued R could not know about W when it issued the request and the server handling the request could not know about R until the request arrived. More formally, W is unknown to r if $W_{inv} \not\rightarrow R_{inv}$ and $R_{inv} \not\rightarrow e'$, where e' is the event on server S that directly precedes r. The server handles r with the *default* behavior if W is unknown to r.

Assumptions. We assume a reliable network, reliable processors, and one-shot transactions. A reliable network eventually delivers every message sent. Reliable processors eventually receive and process every message sent. One-shot transactions [20] require at most one request per server process, i.e., they do not use the output of a request as part of the input of another request.[2] These assumptions are not necessary as the SNOW Theorem holds without them. Assuming them demonstrates the strength of the impossibility result. We also use these assumptions when characterizing what is possible (§4.3–4.4) and categorizing related work (§7).

We also assume there are at least two server processes and at least three client processes. These assumptions are necessary for our proof. SNOW is possible with a single server process or a single client process. It is an open question if SNOW is possible when the system has at least two server processes and exactly two client processes.

[2] Equivalently, there are no cross-server key- or value-dependencies.

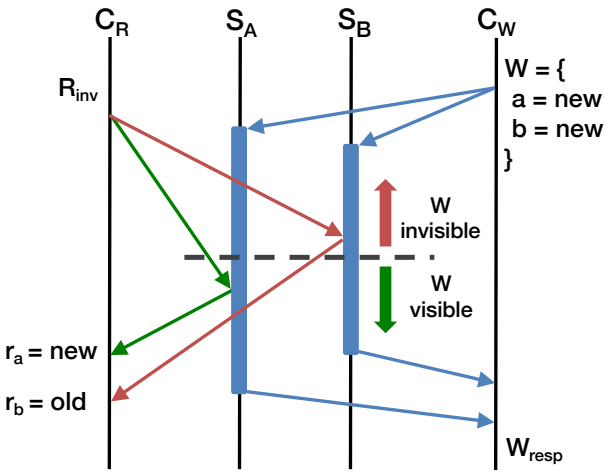


Figure 2: The asynchronous nature of the network allows one request in a read-only transaction to arrive before a conflicting write transaction is visible on one server, while another request arrives after the write transaction becomes visible at a different server. This requires read-only transactions to either settle for weaker consistency (S), block some read requests (N), coordinate and/or retry the reads (O), or preclude conflicting write transactions (W).

4.2 SNOW is Impossible

The SNOW Theorem is an impossibility result that states no read-only transaction algorithm can provide optimal latency and the highest power. Providing optimal latency requires providing the non-blocking (N) and one response (O) properties. Providing the highest power requires providing strict serializability (S) and being compatible with conflicting write transactions (W).

The intuition of the proof is that when a transaction with writes commits, there is a point at every server when the transaction becomes visible, i.e., newly arriving reads will see its effect. However, the asynchronous nature of the network allows read requests to arrive before the transition on one server and after the transition on another, as shown in Figure 2. Turning this intuition into a proof, however, turns out to be more complex. We prove the SNOW Theorem by contradiction, i.e., we assume there exists a read-only transaction with all of the SNOW properties and then eventually show this assumption leads to a contradiction. First, we show that the default behavior of servers must be to return a value, which we called the "default" value (Lemma 1). Second, we show that this default value must initially not expose an ongoing write transaction (Lemma 2). Third, we show that this default value must eventually expose that write transaction (Lemma 3). Fourth, we show that this default value must transition from not exposing the write transaction to exposing it, at some point on each server (Corollary 4). Finally, we prove the SNOW Theorem by constructing an execution where two requests from a read-only transaction take the default behavior at two different servers, with one request arriving before the transition and one arriving after the transition.

The SNOW Theorem. *No read-only transaction algorithm provides all of the SNOW properties.*

Proof. Assume to contradict that there exists a read-only transaction algorithm with all of the SNOW properties. Consider a distributed system with at least the two servers, S_A and S_B, and the three clients, C_R, $C_{R'}$, and C_W, that we assumed exist.

Let R be a read-only transaction issued by C_R that reads a from S_A and b from S_B. Let the first events that happen on S_A and S_B as part of the read-only transaction algorithm be r_a and r_b, respectively. Let W be a conflicting write transaction that writes $a = new$ and $b = new$. Let the values of a and b before W is applied be old.

Lemma 1. *The default behavior at servers returns values that are used by clients.*

Proof. This follows directly from the non-blocking and one response properties. ■

Lemma 2. *Servers initially return old by default.*

Proof. R and W can be concurrent by the conflicting write transaction property. r_a can occur at S_a before any event that happened after W_{inv} by the concurrency of R and W and the asynchronous network. r_a can be the first request in R to arrive at a server by the asynchronous network. Then, by definition, r_a is handled by default and returns a value by Lemma 1. S_a cannot know of W because $W_{inv} \not\rightarrow r_a$ and so must return old. ■

Lemma 3. *Servers eventually return new by default.*

Proof. Assume to contradict that servers never return new by default. Let R be invoked after W returns at C_W. R must return $a = new$ and $b = new$ by strict serializability. By assumption, S_a and S_b must have returned new by a non-default behavior. Let r_a be the first of the requests of R to be handled by a server, by the asynchronous network. Then the non-default behavior must have been triggered by the receipt at C_R of some message m_{see} that connects $W_{inv} \rightarrow R_{inv}$.

Consider the execution up until the point where the m_{see} message is in the network. By the asynchronous network we can deliver or delay messages arbitrarily. Delay all messages not explicitly mentioned and continue that execution by delivering m_{see}.[3] Then deliver all messages for R, which must still see $a = new$ and $b = new$

[3] This construction guards against write transaction algorithms that explicitly notify all clients of their existence before completing.

because they are indistinguishable from the original execution. Let R' be a read-only transaction issued by $C_{R'}$ that reads a from S_A and b from S_B that is invoked after R returns. Deliver no messages to $C_{R'}$ before it issues R', and deliver all of the messages in R'. Let r'_a be the first request in R' delivered by the asynchronous network. By definition r'_a is handled by default, and by assumption *old* must be returned. By strict serializability, *old* must also be returned for b to R'. Thus, R' reads $a = old$ and $b = old$ even though it is invoked after R returns having read $a = new$ and $b = new$. This violates strict serializability and is our contradiction. ■

Corollary 4. *There exists a transition at each server between defaulting to old and defaulting to new.*

Proof. This follows directly from Lemmas 2 and 3. ■

Proof of the SNOW Theorem. Let t_A and t_B be the first transitions from defaulting to old and defaulting to new at S_A and S_B, respectively, that exist by Corollary 4. Let t_A happen first, without loss of generality. Deliver no message to C_R before R is invoked. Deliver r_a immediately after t_A and before r_b. By definition, r_a is handled by default, and by being immediately after t_A it must return *new*. Next, immediately deliver r_b. By definition, r_b is handled by default. By being before t_B, r_b must return *old*. Thus, in this execution R returns $a = new$ and $b = old$. This violates strict serializability and is the contradiction we needed to prove the SNOW Theorem. ■

SNOW with Partial Synchrony. We have shown the correctness of the SNOW Theorem for the asynchronous network model. The theorem also holds for partially synchronous networks because transforming bounds on delay or clock drift into knowledge that can be used in an algorithm requires blocking in proportion to those bounds [15]. Lemma 2 still holds because r_a may still arrive before knowledge of W and would need to wait for that knowledge to arrive to be able to return *new*, which is not allowed because it is blocking. Lemma 3 also still holds because eliminating the possibility of C_R receiving m_{see}, then completing a read-only transaction, and then $C_{R'}$ completing a read-only transaction before receiving any messages would require waiting out bounds. That waiting would have to be on the read path of either C_R and/or $C_{R'}$, which is not allowed because it is blocking.

4.3 Tightness and SNOW-Optimality

We demonstrate the tightness of the SNOW Theorem by showing that every combination of three out of the four SNOW properties is possible. That is, there exists read-only transaction algorithms that satisfy S+O+W, N+O+W, S+N+W, and S+N+O.

Rococo-SNOW, one of the algorithms we will discuss later in this paper, is S+O+W. It is a blocking algorithm but is compatible with stronger transaction semantics, i.e., write-only transactions and general transactions. Algorithms that provide multi-object snapshots in the past are often N+O+W algorithms. For instance, Spanner [11]'s snapshot reads API that is serializable. MySQL Cluster [31] also uses a N+O+W algorithm while providing read committed consistency. These algorithms favor low latency over the isolation and/or recency of strict serializability. Eiger [27]'s read-only transaction algorithm is S+N+W, as it uses multiple rounds to make the write transaction known to all reads.

We designed a novel algorithm for COPS, COPS-DW, that satisfies S+N+O by making all simple write operations go through a distinguished writer. The distinguished writer totally orders the writes. In addition, when each write commits, the distinguished writer computes a consistent snapshot for concurrent or later read-only transactions to return. This algorithm is of theoretical interest only and is not practical because it serializes all writes. It remains open whether there exists practical S+N+O algorithms. These algorithms demonstrate the tightness of the SNOW Theorem, that is, the four SNOW properties are the minimal set of properties that make coexistence impossible.

Given that the SNOW Theorem is tight, we define a read-only transaction algorithm to be *SNOW-optimal* if its properties sit on the boundary of the SNOW Theorem, i.e., it achieves three out of the four SNOW properties. In the four different combinations of SNOW-optimality, S+N+O and N+O+W favor the performance of read-only transactions since non-blocking and one response lead to low latency. We call algorithms that satisfy N+O *latency-optimal*. In contrast, property combinations S+O+W and S+N+W lean towards the power of read-only transactions as they provide the strongest consistency guarantee and compatibility with conflicting write transactions.

4.4 Spectrums of Properties

To fully understand what we can learn from the SNOW Theorem, we further tighten the boundary on what is possible by moving beyond considering the one response and strict serializability properties as binary to viewing them as spectrums.

If the one response property is sacrificed, then how many rounds of messages are sufficient for the rest of the properties to hold?

By examining the systems we have found, existing read-only transaction algorithms range from at most three rounds of messages (Eiger) to an unbounded number of rounds of messages. It is currently open if there exist S+N+W algorithms with at most two rounds.

If the strict serializability property is sacrificed, then what is the next strongest consistency model an algorithm can achieve if the other three properties hold?

Process-ordered serializability is a consistency model slightly weaker than strict serializability that is effectively the combination of serializability and sequential consistency [22]. It requires that there exists a legal total ordering over all operations in the system, provides transactional isolation, and guarantees that the legal total order agrees with each process's ordering of its own operations. It is weaker than strict serializability in that it does not necessarily return the most recent values across processes. We designed a novel algorithm we call Eiger-PS for the Eiger data store. Eiger-PS satisfies N+O+W, while providing process-ordered serializability.

The key idea of the read-only transaction algorithm of Eiger-PS is for each client to maintain a serializable view of the system and to only move to a new view when it is certain that the new view contains that client's most recent write. We accomplish this by having each client maintain a *global safe time*, GST, which is the latest logical time at which no server is holding a pending write transaction. The GST is maintained by each client periodically requesting from every server their *local safe time*, which is the latest time on the server with no in-progress write transactions. The client then only reads at its GST, which provides serializability in Eiger [27]. To achieve process-ordered serializability, each client must see its most recent write. Reading at the GST will not necessarily guarantee that the client sees its most recent write as the client may not have updated its GST since the commit of the last write transaction. Thus, we have each write transaction wait to return until the writing client's GST exceeds the logical commit time of the write transaction. This does not provide strict serializability because a client may not see the most recent write committed by another client.

Because process-ordered serializability is possible, all of the consistency models that are weaker than process-ordered serializability are also able to coexist with the other three properties. For instance, some weaker consistency models include serializability, causal consistency, snapshot isolation, parallel snapshot isolation, and read committed. That is, with all of these weaker forms of consistency, a read-only transaction algorithm can provide N+O+W.

5 Read-Only Transaction Designs

This section explores how to use SNOW-optimality as a lens to examine existing algorithms, our common insight in deriving SNOW-optimal algorithms, and the designs of COPS-SNOW and Rococo-SNOW that integrate new read-only transaction algorithms.

5.1 Exploring Improvements with SNOW

SNOW-optimality is a powerful lens with which to examine the design of read-only transaction algorithms. If an algorithm is already SNOW-optimal, then we cannot improve it without making a different choice in the trade-off between latency and power. If an algorithm is not SNOW-optimal, however, we know that it is possible to improve it without making a different design choice.

Any algorithm that is not SNOW-optimal has at most two of the SNOW properties. We improve upon such algorithms by keeping the SNOW properties they provide and adding at least one of the latency-related properties. We do not add strict serializability or compatibility with conflicting write transactions because doing so would change the base system into something new.

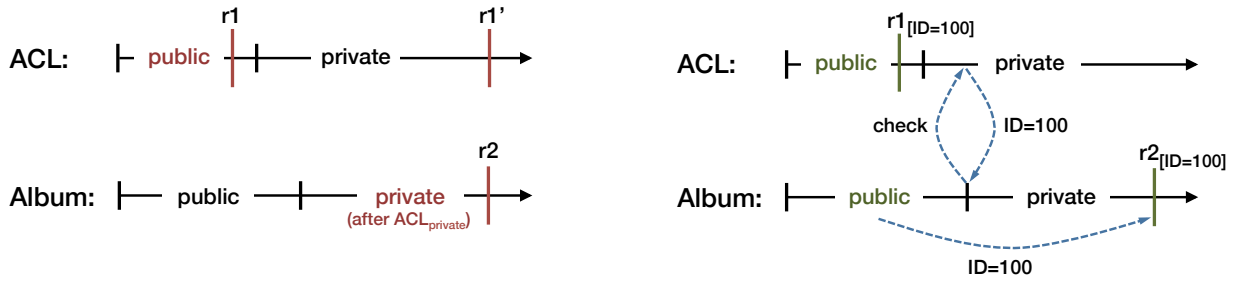
The COPS distributed data store has a non-blocking algorithm for its read-only transactions and we designed a new non-blocking and one response algorithm. The new COPS-SNOW algorithm is latency-optimal, but not SNOW-optimal because it is neither strictly serializable nor compatible with write transactions. The Rococo distributed data store has a read-only transaction algorithm that is strictly serializable and compatible with conflicting writes. We designed a new algorithm that adds the one response property. The new Rococo-SNOW algorithm is SNOW-optimal but not latency-optimal.

In addition to helping us discover systems whose algorithms we can improve, the SNOW Theorem also helps us avoid trying to improve systems that we cannot. Some of the distributed data stores we examined were already SNOW-optimal and so we knew it would be impossible to improve them. For instance, Spanner [11] is SNOW-optimal because it has a strictly serializable, one round read-only transaction algorithm that is compatible with conflicting write transactions. Section 7 discusses more systems that are already SNOW-optimal.

5.2 Common Insight for Optimal Reads

After identifying if and how we can improve upon the read-only transaction algorithm in existing systems, we need to design algorithms that realize that improvement. We have found one common insight in our new algorithms that we think will be useful in deriving other SNOW-optimal algorithms. This key insight is to make reads cheaper by making writes more expensive.

Instead of blocking reads, block writes. Instead of requiring extra rounds of communication for reads, require them for writes. Shifting the burden to writes will always improve the individual performance of reads. But for the read-heavy workloads that are common for Web services, such a design can also improve overall performance because it diminishes a minority of the workload to improve the majority of it.



(a) COPS read-only transaction **(b) COPS-SNOW read-only transaction**

Figure 3: Ensuring causal consistency for a private photo that is updated after setting the ACL to private. COPS takes two rounds of requests (r_1,r_2 then $r_{1'}$) while COPS-SNOW takes only one round to ensure consistency.

5.3 COPS-SNOW Design

This subsection describes the COPS system and our new read-only transaction algorithm for it.

COPS Overview. COPS is a scalable, geo-replicated storage system where each replica (datacenter) contains a full copy of the data sharded across many machines. COPS has single-key write operations and does not have write transactions. Each datacenter accepts writes locally and then replicates them to remote datacenters with metadata that indicates their causal dependencies. Remote datacenters check that the write's causal dependencies are satisfied before applying the write. This ensures that the data spread across many shards in each replica is always causally consistent [3, 21]. COPS has read-only transactions that are handled entirely locally in a replica and provide a causally consistent view of the data store. For the rest of our discussion of COPS we focus on its operations within a single datacenter, but our results apply equally in the geo-replicated setting.

The original read-only transaction algorithm in COPS is causally consistent, non-blocking, two rounds, and is not compatible with write transactions. COPS read-only transactions begin when application logic invokes a read-only transaction that includes the full list of keys the client wants to read. The client then sends out a first round of read requests to each shard that has data in the transaction. Servers respond with their current value for the data along with the causal dependencies of each value. After the first round the client checks to see if all of the returned values are mutually consistent. Each causal dependency that is returned with a read is a constraint on other values in the system, e.g., b_1 depends on a_1 means that if a client observes b_1 it must also observe a_1 or an even later version of a (if it reads a). If all of the dependencies of all the values returned in the first round are satisfied, then COPS returns the values to the application logic after a single round. If, however, not all of the dependencies are satisfied, then COPS issues a second round of read requests for each value that does not satisfy other values' dependencies. COPS requests the specific versions of keys that are depended upon, e.g., if a_0 and b_1 are returned in the first round then a_1 will be requested in the second round. Requesting these specific versions guarantees COPS will complete in two rounds because these versions satisfy current dependencies. In addition, these specific versions do not introduce any new dependencies because, by the definition of causality, their dependencies are a subset of the dependencies of values that depend upon them, e.g., a_1's dependencies are a subset of b_1's.

COPS-SNOW Algorithm. We improve COPS with a new latency-optimal read-only transaction algorithm. Our COPS-SNOW algorithm keeps the power properties of the current algorithm: it provides causal consistency and is not compatible with conflicting write transactions. It is latency-optimal because it keeps the non-blocking property of the current algorithm and adds the one response property. Following our common insight we shift the complexity from the reads to the writes in COPS. More specifically, we shift the consistency check and second round fetch of consistent values from the read-only transaction algorithm into the write algorithm.

In COPS a second-round read is needed if and only if one part of the read-only transaction r_a does not see a write w_{a1} and another part of the read-only transaction r_b does see a write w_{b1} that is causally after w_{a1}. Figure 3a shows this in action with COPS using the canonical access control list (ACL) and photo example where an album is switched to private (w_{a1}) and then a private photo is added (w_{b1}). In this example, COPS will send a new read $r_{1'}$ that will see write w_{a1} and return the consistent set of the private ACL and Album.

Our new algorithm flips this responsibility by having a write check if any of its causal dependencies have not been observed by an ongoing read-only transaction. If

```
Client Side
function read_only_txn(<keys>):
  trans_id = generate_uuid()
  vals, deps = []
  for k in keys # in parallel
    vals[k], deps = read_txn(k, trans_id)
  # update causal dependencies
  return vals

function write(key, val):
  old_deps = get_deps()
  new_dep = write(key, val, old_deps)
  # update causal dependencies
  return

Server Side
function read_txn(key, trans_id):
  if trans_id in old_rdrs[key]
    time = old_rdrs[key][trans_id]
    return val = read_at_time(key, time)
  curr_rdrs[key].append(trans_id,
                    logical_time.now())
  return read(key)

function write(key, val, deps):
  for d in deps # in parallel
    old_rs = check_dep(d)
    old_rdrs[key].append(old_rs)
  old_rdrs[key].append(curr_rdrs[key])
  curr_rdrs[key].clear()
  write(key, val)
  # calculate causal dep for this write
  return new_dep

function check_dep(dep)
  # normal causal dependency check
  return old_rdrs[dep.key]
```

Figure 4: Pseudocode for COPS-SNOW.

any of those causal dependencies were not observed by a read-only transaction then this write should not be observed by it either, so the write updates metadata encoding that. Figure 3b shows our new algorithm in action on the same example. COPS-SNOW returns the consistent set of the public ACL and Album.

Figure 4 shows the pseudocode for COPS-SNOW. On the client side writes are the same as in COPS, and read-only transactions are similar but simpler because they return the values from the first and only round. On the server side there are five high-level changes relative to COPS: two changes to reads, two to writes, and one to dependency checks.

The first change to reads is that they check to see if their enclosing transactions are listed in the old readers data structure (`old_rdrs`) and if so return an older, consistent value. The second change to reads is that they are recorded as observing the current value in the current readers data structure (`curr_rdrs`). This enables writes that overwrite the value to record which read-only transactions did not see them.

The first change to writes is that they do dependency checks to see if any of their causal dependencies overwrote values that a read-only transaction observed. If so, the write records in the old readers data structure that those read-only transactions should see older values to be consistent. The second change to writes is that they record any read-only transactions that observed the value they overwrote by copying the current readers data structure into the old readers data structure. The change to dependency checks is that they return the set of read-only transactions that did not see a causally dependent update. The combination of changes to writes enables this. Adding reads of the overwritten values captures reads that did not observe this write. Adding reads that did not see this write's causal dependencies—which also did dependency checks that added their causal dependencies, and so on—captures the transitive closure of this write's dependencies.

For clarity the pseudocode excludes logic related to updating causal dependencies; grouping reads to keys that are stored on the same server; updating Lamport clocks; and storing, reading, and garbage-collecting old versions. All of this logic is similar to what COPS does, and is identical to what Eiger does.

5.4 Rococo-SNOW Design

This subsection describes the Rococo system and our new read-only transaction algorithm for it.

Rococo Overview. Rococo is a strictly serializable, distributed data store with general transactions [29]. Rococo was designed primarily for the single datacenter setting we consider here. Rococo introduced a new concurrent control algorithm that outperforms traditional concurrency control algorithms under high contention workloads by reordering conflicting transactions instead of aborting them. Rococo requires the transactions that it executes to be chopped into pieces and analyzed for safety before the system is deployed. Each transaction is chopped into pieces that execute on shards as stored procedures. For instance, to increment keys *a* and *b* that are stored in different shards, Rococo would have a piece for *a* that invokes the increment stored procedure server-side and a separate piece for *b* that invokes the increment stored procedure server-side. Rococo analyzes the pieces of transactions to ensure that if they conflict at run time it will be able to safely reorder them.

Rococo's general transactions operate in three phases run by a coordinator. The first phase distributes pieces of the transaction to the appropriate shards and determines all directly conflicting transactions. The second phase ensures all shards have the same metadata about directly conflicting transactions. The third phase, which can often be skipped, ensures all shards have the same metadata about transitively (but not directly) conflicting transactions. After the second or third phase each shard deterministically orders all conflicting transactions and then executes them in that order.

The original read-only transaction algorithm in Rococo is strictly serializable, blocking, multi-round, and compatible with conflicting write transactions. It takes two rounds in the best case and an infinite number of rounds in the worst case. In the first round the coordinator sends read requests to each involved shard. Those read requests block until after the execution of all conflicting transactions that started at that shard before this read arrived. The second round is identical, and Rococo considers the read-only transaction successful only if both rounds read the same values. If not, Rococo will continue issuing another round of reads until two consecutive rounds return the same results. This algorithm ensures strict serializability for the reads because it ensures they are totally ordered relative to all conflicting transactions. Waiting for all conflicting transactions to execute at a shard before returning in the first round ensures those transactions will have at least started at all other involved shards before the second-round read arrives. Thus, if a read-only transaction is not fully ordered before or after a write transaction, it will see different results and continue trying.

Rococo-SNOW Algorithm. We improve Rococo with a new SNOW-optimal read-only transaction algorithm. Our Rococo-SNOW algorithm keeps the power properties of the current algorithm: it provides strict serializability and is compatible with conflicting write transactions. It also adds the one response property to these, which makes it SNOW-optimal. It is not latency-optimal because it blocks, which we know is unavoidable. Following our common insight we shift the complexity from reads into the commit algorithm of Rococo.

Due to space limitations, we only briefly describe our new algorithm and omit its pseudocode. It is conceptually similar to the COPS-SNOW algorithm in that it tracks whenever a value is read by a read-only transaction and then propagates the knowledge of that to all other servers where that ordering is important. In Rococo the second round (and additional rounds after that) are necessary to protect against the case where one part of a read-only transaction does not see a write transaction but another part does. Our new algorithm ensures this case never occurs by blocking each piece of a read-only transaction until all conflicting write transactions have executed at that shard. Rococo's commit algorithm ensures that each piece of a transaction has knowledge of the transitive closure of all conflicting transactions. We piggyback the knowledge of read-only transaction pieces that did not see any of the transitive closure of conflicting transactions into that commit algorithm.

If a different piece of the read-only transaction did not see a conflicting write transaction, then this shard will know about that through the commit algorithm before it unblocks this piece of the read-only transaction. Thus, the shard will know whether to return an old state or the most recent state when it executes the read-only transaction piece. When the coordinator receives replies from all involved shards, it knows the results are consistent and thus returns them to the application logic.

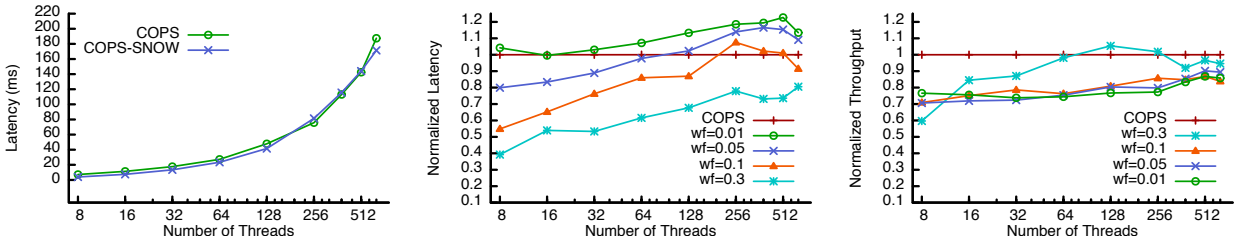
6 Evaluation

We experimentally evaluate COPS-SNOW and Rococo-SNOW to understand how their latency and throughput compare to the original COPS and Rococo under a variety of settings. The evaluation shows that both COPS-SNOW and Rococo-SNOW achieve lower latency for read-only transactions. COPS-SNOW achieves this at the cost of lower system throughput. Rococo-SNOW has slightly lower throughput than Rococo under low contention but actually achieves much higher throughput in the high-contention settings Rococo was designed for.

6.1 COPS-SNOW

Implementation. We implemented COPS-SNOW as a modification to Eiger [27], the successor to COPS. Eiger provides the same level of consistency guarantees as COPS, adds support for write-only transactions, supports a richer data model, and has a read-only transaction algorithm based on logical time instead of causal dependencies. We disable Eiger's write-only transactions to change it to a COPS-like mode where it has an at most two-round read-only transaction algorithm. This allows Eiger to propagate and store far fewer dependencies than COPS and makes its read-only transaction algorithm far more efficient. For this reason, we implement on top of Eiger: we are comparing to the state-of-the-art read-only transaction algorithm for causally-consistent systems without compatibility with write transactions. We refer to this baseline as COPS throughout the evaluation. This implementation is available publicly on GitHub.[4]

[4] https://github.com/USC-NSL/COPS-SNOW



(a) Latency with 0.1 write frac **(b) Normalized median latency** **(c) Normalized throughput**

Figure 5: Latency and throughput for COPS-SNOW and COPS for varying fractions of writes in the workload. The latency graphs show the median latency for read-only transactions. Throughput graphs show the overall throughput of the cluster.

Testbed, Bottleneck Resource, and Trials. We tried to match our experimental setup to that of Eiger's as much as possible. We ran all the experiments on the PRObE Nome testbed [17]. (Some of Eiger's experiments were run on Nome's predecessor Kodiak.) Each Nome machine has four Quad-Core AMD Opteron 2.2 GHz CPUs, 32 GB RAM, and two network interfaces: one 1 Gbps Ethernet and one 20 Gbps Infiniband. All COPS-SNOW experiments were run on a 20 Gbps Infiniband network, which matches the network configuration Eiger used. Due to inefficient thread scheduling within Cassandra, upon which Eiger is based, one instance cannot saturate all 16 cores on a physical machine. To make a fair baseline for comparison, we run two instances on each node so we can saturate one of the machine's resources. For all experiments the bottleneck is network interrupt processing. We ran 15 trials for each experiment and report the median. Each trial lasted at least 90 seconds with the first and last quarter excluded to avoid artifacts due to warm up, cool down, and imperfectly synchronized clients.

Configuration and Workloads. We evaluate COPS-SNOW using two logical datacenters that are physically co-located in the testbed with eight server machines each. We use 16 client machines to load the servers in one of the logical datacenters. (Our throughput disadvantage would decrease as client load shifted to be more evenly distributed across the datacenters.) We use the dynamic workload generator from Eiger with Zipfian traffic generation using these parameters:

Parameter	Default	Range
Value Size (B)	128	
Cols/Key	5	
Keys/Operation	5	5 – 32
Write Fraction	0.1	0.01 – 0.5
Zipfian Constant	0.8	0.7 – 0.99

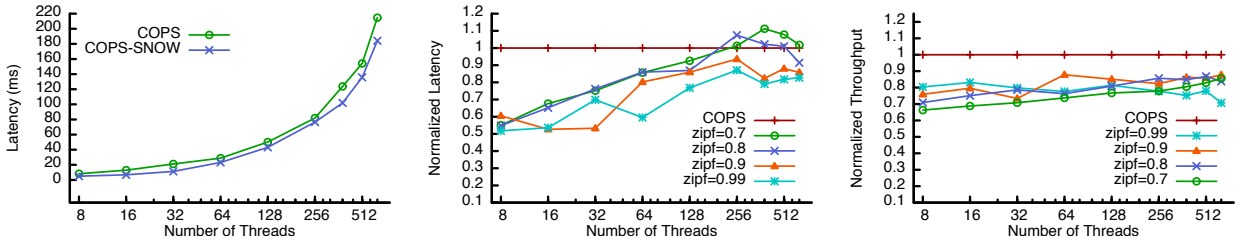
The default parameters match the defaults in Eiger's evaluation and we choose 0.8 as the default Zipfian constant because it provides moderate skew. For a write of 5 – 32 keys we send out 5 – 32 parallel, unrelated individual write operations. We explore how the throughput of COPS-SNOW compares to COPS under a variety of settings shown by the ranges of parameters we explore.

Performance with Varying Write Fraction. Figure 5 shows the latency and throughput of COPS-SNOW and COPS as we increase the number of closed-loop client threads on each client machine. Figure 5a shows the latency with a write fraction of 0.1. In this setting, there are enough writes to require COPS to sometimes go to the second round of its algorithm and COPS-SNOW has a slight latency advantage.

Figure 5b shows the median latency of COPS-SNOW normalized against that of COPS for a variety of write fractions. When the write fraction is very low at 0.01, there are so few writes that all of COPS's read-only transactions take only one round and have similar latency to COPS-SNOW. When the number of closed-loop client threads is high, sometimes the latency of COPS-SNOW is actually higher than that of COPS because the systems are overloaded. This overload is outside the model we considered in this paper, which we believe is reasonable because overload is outside of the normal range of system operations. Exploring the affect of overload on latency is an interesting avenue of future work.

The larger result from Figure 5b is that the latency improvement of COPS-SNOW increases as the write fraction increases. It is substantial when the write fraction is close to 0.3. The rest of the experiments use the default 0.1 fraction. If they used a higher write fraction their latency results would be more pronounced and if they used a smaller write fraction their latency results would be less pronounced.

Figure 5c shows the throughput of the cluster in the same settings. Here we see that COPS-SNOW is trading



(a) Latency with Zipf=0.9 **(b) Normalized median latency** **(c) Normalized throughput**

Figure 6: Latency and throughput for COPS-SNOW and COPS for varying levels of skew in the workload. Latency graphs show the median latency for read-only transactions. Throughput graphs show the overall throughput of the cluster.

away throughput for lower latency for read-only transactions. This loss in throughput comes from the extra messages in the write algorithm in COPS-SNOW.

Performance with Varying Keys per Operation. Due to space constraints, we omit the figure showing the throughput and latency of COPS-SNOW and COPS for a varying number of operations in each transaction. Our results show that COPS-SNOW has a latency advantage that increases as read-only transactions increase in size because the read-only transactions in COPS are more likely to go to the second round. It also shows that the throughput of COPS-SNOW becomes worse relative to COPS as read-only transactions become larger because each write has more causal dependencies to check.

Performance with Varying Levels of Skew. Figure 6 shows the latency and throughput of COPS-SNOW and COPS for varying levels of skew in the workload. Figure 6a shows the latency of read-only transactions when the skew is moderate with a Zipfian constant of 0.9. COPS-SNOW has a latency advantage over COPS because there is moderate contention in the workload and COPS sometimes needs a second round for reads. Figure 6b shows that COPS-SNOW has an increasing latency advantage as the workload becomes more skewed as long as the systems are not overloaded. Figure 6c shows that the throughput disadvantage of COPS-SNOW decreases slightly as the workload becomes more skewed and the extra RPCs in the write algorithm of COPS-SNOW are offset by the extra RPCs in the second round of read-only transaction in COPS.

6.2 Rococo-SNOW

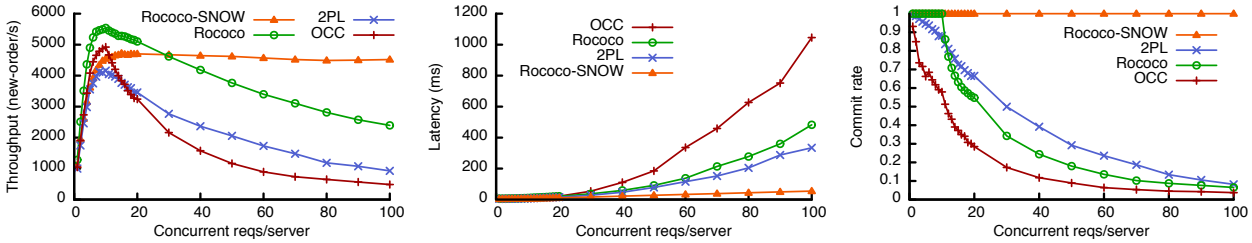
Implementation. We implemented Rococo-SNOW as a modification to Rococo's code base. We converted Rococo from a single-version to a multi-version system to support our read-only transactions and replaced its read-only transaction logic with our new algorithm. This implementation is available publicly on GitHub.[5]

Testbed, Bottleneck Resource, and Trials. We tried to match our experimental setup to that of Rococo's as much as possible. We ran all the experiment on the PRObE Nome testbed. (All of Rococo's experiments were run on Nome's predecessor Kodiak, which has been decommissioned.) The machines' specifications are the same as they were for COPS-SNOW with two exceptions. First, we use the Ethernet network interface instead of the Infiniband interface to match the setup from Rococo's evaluation (the network is never a bottleneck). Second, Rococo is single-threaded and used only one core in its evaluation so we run one Rococo process per machine, which only uses one of the cores. This core is always the bottleneck. The rest of our experiment settings were identical to what were used in Rococo, e.g., each trial lasted at least 60 seconds with first and last quarter excluded to avoid artifacts due to warm up, cool down, and imperfectly synchronized clients.

Configuration and Workload. We evaluate Rococo-SNOW using 8 server machines and 16 client machines. We evaluate using Rococo's district-sharded TPC-C with all parameters matching Rococo's evaluation [29].

TPC-C Throughput and Read-only Transactions. Figure 7 shows the performance of Rococo-SNOW, Rococo, 2PL, and OCC as load and contention are increasing by increasing the number of concurrent requests per server in the system. The throughput for the (read-write) new order transaction is shown in Figure 7a. The TPC-C benchmark requires a specific mix of its five transaction types, so this throughput is proportional to the throughput of each type of transaction. Our results for Rococo,

[5] https://github.com/USC-NSL/Rococo-SNOW



(a) New order throughput **(b) Stock level latency** **(c) Stock level commit rate**

Figure 7: Rococo-SNOW performance with Rococo's TPC-C benchmark. The throughput of new order transactions is shown, which is proportional to the throughput of all transactions. The median latency and commit rate of the read-only stock level transactions are shown.

2PL, and OCC, match what was observed in Rococo's evaluation. We see that Rococo-SNOW provides lower peak throughput than Rococo. This is because with so few requests per server there is low contention and Rococo's read-only transactions rarely "abort" (i.e., must continue to another round), while Rococo-SNOW makes the write algorithm more complex. Once contention increases to a moderate level with 30 requests/server, the throughput of Rococo-SNOW matches that of Rococo and then starts to exceed it. In the high-contention workloads that Rococo was designed for, Rococo-SNOW actually has much higher throughput. This is because Rococo-SNOW's read-only transactions always succeed after a single blocking round, while Rococo's read-only transactions often have to retry many times when contention is high.

Figure 7b shows the latency of the read-only stock level transactions, which shows that Rococo-SNOW provides much lower latency than Rococo. Figure 7c shows the "commit" rate of stock level transactions. The low commit rate for these read-only transactions makes them the bottleneck for Rococo under high contention, even though all of Rococo's read-write transactions have a commit rate of 100% (not shown).

7 Related Work

This section reviews existing read-only transactions and discusses other impossibility results.

Existing Read-Only Transactions. Figure 8 categorizes many recent systems with read-only transaction algorithms according to the SNOW properties. Some like Yesquel [1], MySQL Cluster [31], and Spanner [11]'s snapshot reads API are SNOW-optimal and latency-optimal. To be optimal in both the systems must give up one of the power related properties, and all of the three systems give up strict serializability. Spanner's snapshot reads API provides a serializable (but potentially stale) snaphot over the data. Yesquel provides snapshot isolation, which is slightly weaker. MySQL Cluster provides the yet weaker read-committed consistency model.

Many systems are SNOW-optimal but not latency-optimal. These systems made a different design choice and give up a latency related property to be as powerful as possible. One system, Eiger [27], has a bound on the number of rounds needed for read-only transactions. Other non-blocking SNOW-optimal systems have an unbounded number of rounds. These include DrTM [37], RIFL [23], and Sinfonia [2]. The unbounded number of rounds typically comes from algorithms that can abort, e.g., those based on optimistic concurrency control. Rococo-SNOW is a different flavor of SNOW-optimal because it is blocking, as is the algorithm of Spanner-RO [11], which is Spanner's strictly serializable read-only transaction API.

Finally, many systems are neither SNOW-optimal nor latency-optimal. This suggests there is room for improvement in the latency of their read-only transaction algorithms without making a fundamentally different design choice. COPS [26] and Rococo [29] fall into this category, which is the primary reason we developed new algorithms for them. Walter [35], Orbe [14], ChainReaction [5], Calvin [36], RAMP [8], Granola [12], TAPIR [38], and Janus [30] also fall in this category and are strong candidates for improvement.

Impossibility Results. Our work is inspired by other impossibility results. The FLP result proves that in a deterministic asynchronous system distributed processes cannot always achieve consensus if even one process can be faulty [16]. FLP is a different type of impossibility result than SNOW because it states a liveness property cannot always be satisfied: it is impossible to guarantee a good thing (consensus) will always eventually happen. In practice, however, consensus despite multiple faulty processes happens regularly. The SNOW theorem, on the other hand, states a safety property cannot always be

System	S	N	O	W
SNOW-optimal and latency-optimal				
Spanner-Snap [11]*	Ser	✓	✓	✓
Yesquel [1]	SI	✓	✓	✓
MySQL Cluster [31]*	RC	✓	✓	✓
SNOW-optimal				
Eiger [27]*	✓	✓	≤ 3	✓
DrTM [37]*	✓	✓	≥ 1	✓
RIFL [23]	✓	✓	≥ 2	✓
Sinfonia [2]	✓	✓	≥ 2	✓
Spanner-RO [11]*	✓	×	✓	✓
Rococo-SNOW*	✓	×	✓	✓
Latency-optimal				
COPS-SNOW*	Causal	✓	✓	×
Neither SNOW-optimal nor latency-optimal				
Janus [30]	✓	×	≤ 2	✓
Calvin [36]	✓	×	2	✓
Rococo [29]*	✓	×	≥ 1	✓
TAPIR [38]*	Ser	×	✓	✓
Granola-Independent [12]*	Ser	✓	≥ 2	✓
Granola-Coordinated [12]*	Ser	✓	≥ 2	✓
Walter [35]	PSI	✓	≤ 2	✓
COPS [26]*	Causal	✓	≤ 2	×
Orbe [14]*	Causal	×	2	×
ChainReaction [5]*	Causal	×	≥ 2	×
RAMP-F [8]*	RA	✓	≤ 2	✓
RAMP-H [8]*	RA	✓	≤ 2	✓
RAMP-S [8]*	RA	✓	2	✓

Figure 8: Categorization of read-only transactions along the SNOW properties. Astericks denote algorithms that are specialized for read-only transactions.

satisfied: it is impossible to guarantee a bad thing (violating strict serializability) will never happen. Any system that can violate a safety property is not safe, and thus cannot be used in practice.

The CAP Theorem proves it is impossible for a distributed data store to always provide consistency (strict serializability) and availability under network partitions [9, 18]. Lipton and Sandberg [25] first discovered and Attiya and Welch [7] later refined a result that shows it is impossible to achieve sequential consistency and low latency in a replicated system. The CAP Theorem and Lipton/Sandberg result are similar to SNOW in that they point to a fundamental design decision for system builders where they must choose some properties at the expense of losing others.

A recent line of work has investigated read-only transaction for transactional memory (TM). Attiya et al. [6] proved that it is impossible to have strictly serializable TM implementations that ensure read-only transactions are *invisible*—i.e., reads do not update memory—and *wait-free*—always terminate regardless of concurrent transactions. Peluso at el. [33] further refined this result with a TM implementation that has wait-free read-only transactions but with a relaxed consistency model. This work explores the possibilities of read-only transactions in a different setting from ours. The concerns of the different setting are different, TM is interested in efficient hardware implementation while we are more interested in a finer granularity of performance properties, e.g., one response.

8 Conclusion

Read-only transactions are a fundamental building block for large-scale applications such as modern Web services. The SNOW Theorem proves that there is a fundamental tradeoff between the power and latency of read-only transactions by showing that it is impossible for an algorithm to provide strict serializability, nonblocking operations, one response per read, and compatibility with write transactions. The resulting notion of SNOW-optimality along with latency-optimality are powerful lenses for examining existing systems and determining if the latency of their read-only transactions can be improved. Using those lenses we designed and implemented COPS-SNOW—a new latency-optimal algorithm—and Rococo-SNOW—a new SNOW-optimal algorithm. Our evaluation demonstrates that both algorithms provide lower latency for read-only transactions.

Acknowledgments

We are grateful to Theano Stavrinos, Minlan Yu, the OSDI program committee, and our shepherd, Dan Ports, for their feedback that improved this work. Our evaluation at scale was made possible by the PRObE testbed, which is supported by NSF awards CNS-1042537 and CNS-1042543. The administrative team for the PRObE testbed went above and beyond in their help to us. This work was supported by NSF grants CNS-1464438, CNS-1514422, and AFOSR grant FA9550-15-1-0302. Part of Christopher Hodsdon's work on this project was done while he was an undergraduate student at Rutgers University-Camden and supported by NSF awards 1218620 and 1433220.

References

[1] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Yesquel: scalable SQL storage for Web applications. In *Proc. SOSP* (Oct 2015).

[2] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm forbuilding scalable distributed systems. In *Proc. SOSP* (Oct 2007).

[3] AHAMAD, M., NEIGER, G., KOHLI, P., BURNS, J., AND HUTTO, P. Causal memory: Definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995).

[4] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to adopting stronger consistency at scale. In *Proc. HotOS* (May 2015).

[5] ALMEIDA, S., LEITAO, J., AND RODRIGUES, L. ChainReaction: a causal+ consistent datastore based on chain replication. In *Proc. Eurosys* (Apr 2013).

[6] ATTIYA, H., HILLEL, E., AND MILANI, A. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proc. SPAA* (Aug 2009).

[7] ATTIYA, H., AND WELCH, J. L. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst. 12*, 2 (1994).

[8] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Scalable atomic visibility with RAMP transactions. In *Proc. SIGMOD* (Jun 2014).

[9] BREWER, E. A. Towards robust distributed systems. In *Proc. Principles of Distributed Computing* (Jul 2000).

[10] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. Tao: Facebook's distributed data store for the social graph. In *Proc. ATC* (Jun 2013).

[11] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., ANDSANJAY GHEMAWAT, J. F., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ANDSEBASTIAN KANTHAK, W. H., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., ANDDAVID NAGLE, D. M., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., ANDCHRISTOPHER TAYLOR, M. S., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proc. OSDI* (Oct 2012).

[12] COWLING, J., AND LISKOV, B. Granola: Low-overhead distributed transaction coordination. In *Proc. ATC* (Jun 2012).

[13] DIXON, P. Shopzilla site redesign: We get what we measure. Velocity Conference Talk, 2009.

[14] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proc. SoCC* (Oct 2013).

[15] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*, 2 (1988), 288–323.

[16] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. In *Proc. Principles of Database Systems* (Mar 1983).

[17] GIBSON, G., GRIDER, G., JACOBSON, A., AND LLOYD, W. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:* (June 2013).

[18] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM SIGACT News* (Jun 2002).

[19] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Program-ming Languages and Systems 12*, 3 (1990), 463–492.

[20] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P., MADDEN, S., STONEBRAKER, M., ZHANG, Y., ET AL. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1496–1499.

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

[22] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* (1979).

[23] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITAY, S., AND OUSTERHOUT, J. Implementing linearizability at large scale and low latency. In *Proc. SOSP* (Oct 2015).

[24] LINDEN, G. Make data useful. Stanford CS345 Talk, 2006.

[25] LIPTON, R. J., AND SANDBERG, J. S. PRAM: A scalable shared memory. Tech. Rep. TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[26] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. SOSP* (Oct 2011).

[27] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI* (Apr 2013).

[28] LU, H., VEERARAGHAVAN, K., AJOUX, P., HUNT, J., SONG, Y. J., TOBAGUS, W., KUMAR, S., AND LLOYD, W. Existential consistency: Measuring and understanding consistency at facebook. In *Proc. SOSP* (Oct 2015).

[29] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *Proc. OSDI* (Oct 2014).

[30] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating concurrency control and consensus for commits under conflicts. In *Proc. OSDI* (Nov 2016).

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

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

[33] PELUSO, S., PALMIERI, R., ROMANO, P., RAVINDRAN, B., AND QUAGLIA, F. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proc. PODC* (Jul 2015).

[34] SCHURMAN, E., AND BRUTLAG, J. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. Velocity Conference Talk, 2009.

[35] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proc. SOSP* (Oct 2011).

[36] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proc. SIGMOD* (May 2012).

[37] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proc. SOSP* (Oct 2015).

[38] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactionswith inconsistent replication. In *Proc. SOSP* (Oct 2015).