Research Statement
Haonan Lu

Vision and Philosophy

My Ph.D. research focuses on advancing the state of the art in the understanding, design, and implementation of distributed storage systems. Today’s web services, e.g., Facebook, Google, Amazon, process large numbers of user requests and store huge amounts of data, more than a single server can handle. Thus, these large web services are built on distributed storage systems, which shard data across machines within and/or across datacenters. Storage systems are a research area with potentially large impact because it affects all end users of those web services. Instead of seeking better solutions as is typical, I focus my research on how to design the best possible systems. As a result, my research closes a line of research in an area by showing system designers what is the best they can possibly achieve, rather than providing arbitrarily better solutions than previous work.

To design the best practical systems, I focus on making the systems have high throughput, low latency, and strong consistency. Higher throughput can satisfy the same amount of user requests with fewer physical machines and effectively reduces per-operation cost, thus saving money and energy. Low latency is critical to good user experience and revenue, e.g., Amazon found every 100ms of latency cost them 1% in sales [4]. Stronger consistency ensures the correctness of the system and makes programming easier. These properties are the dimensions of the design space, in which I look for the best designs that achieve the most out of each property.

To make the best systems, I keep pushing new designs close to an ideal, i.e., the intersection of lowest latency, highest throughput, and strongest consistency, until I find an optimum in the design space where it cannot be pushed any further. I then verify the design is indeed optimal by mathematically proving that the ideal is impossible to achieve and there do not exist any designs closer to the ideal than the optimal design. Hence, my approach, i.e., finding and proving the optimum, discovers fundamental tradeoffs in the design space system designers have to make and provides them with insights on how to design the best possible systems.

As a first step towards best distributed storage systems, my Ph.D. research focuses on optimal designs for an important component of the storage system: read-only transactions, which consist only of read operations and do not modify data. I start with read-only transactions because they are implemented in most academic and production systems, hence better designs can bring improvements to a large fraction of existing systems. Moreover, read operations dominate real-world workloads. For instance, 99.8% of the operations to Facebook’s TAO system are reads [1]. Hence, making reads better can largely improve the experience of end users because the majority of user operations are reads.

Ph.D. Research

Existential Consistency

Large-scale web services often trade consistency for performance. Much work has been done to show the performance benefits of adopting weak consistency models. However, the flip side of this tradeoff was never clear. Existential Consistency [7] was the first work that quantitatively studied the user-visible consistency violations incurred by weak consistency models for a large-scale production system—Facebook’s TAO.

Existential Consistency includes a principled, offline analysis, where I wrote software that takes a trace of user requests and identifies consistency violations in the trace. It overcomes a slew of challenges in production systems at a global scale, e.g., clock skew, data privacy, deficient traces, etc. Existential Consistency also presents a real-time, online checker that monitors how well replicas converge after a new
update by actively sending probing messages. Existential Consistency shows that user-visible violations in TAO are rare—6 per million reads—and suggests that work on providing stronger consistency should include transactions to maximize its potential benefit.

**SNOW**

Read-only transactions are important because they ensure that reading data across machines returns a consistent snapshot of the system. For instance, loading a web page may incur hundreds of read requests on multiple servers, each of which stores a subset of the page. A read-only transaction guarantees that the combined results reflect a valid web page. Given that reads dominate real-world workloads and low latency is critical to good user experience, we want read-only transactions to be as fast as possible! On the other hand, we want them to also provide strong consistency, which guarantees the correctness of the system and makes programming easier.

Some systems prefer low latency by sacrificing consistency guarantees, e.g., Facebook’s TAO system, while some systems choose to pay latency penalties in favor of strong guarantees, e.g., Google’s Spanner system [2]. There are many systems in the middle adopting intermediate consistency models while having reasonably low latency, e.g., COPS [5]. I wanted to design read-only transaction algorithms that have the lowest possible latency, i.e., as fast as non-transactional reads. However, it was an open question whether it was possible to achieve the lowest latency, and whether the lowest latency was achievable by any systems under any consistency guarantees. SNOW [8] closes this question and sheds light on designs of latency-optimal read-only transactions instead of making incremental improvements.

SNOW proves an impossibility result—the SNOW Theorem—which states that it is impossible for any read-only transaction algorithm to achieve the following properties at the same time: strict serializability (S), non-blocking (N), one-round (O), and compatible with write transactions (W). N and O together represent the lowest possible latency because it is the fastest a read-only transaction can be, i.e., the client sends read requests in parallel to relevant servers, which immediately respond with consistent data. S and W properties together represent the strongest guarantee read-only transactions can provide.

The impossibility result saves developers from trying the impossible, e.g., attempting to design an algorithm that achieves all four properties. The SNOW Theorem is proven tight, i.e., any combination of three properties is possible. An algorithm is SNOW-optimal if it achieves any three of the properties because this is the most it can have, and latency-optimal if it satisfies properties N and O because this is the fastest it can be. Hence, SNOW-optimality can be used to examine existing systems and help design optimal algorithms. For instance, if the read-only transactions in a system are non-blocking and strictly serializable, but require two round trips, then the developers know confidently with SNOW that the read-only transactions can still be made faster without losing strict serializability. Developers can also gain insights on how to improve them, i.e., making the algorithm one-round.

I use the SNOW Theorem to build new versions of two previously state-of-the-art systems, COPS [5] and Rococo [9], by making their read-only transactions latency-optimal and SNOW-optimal respectively. The common design insight is making read-only transactions faster by shifting the overhead from reads to writes. The evaluations show both new systems achieve lower latency without sacrificing their consistency with slightly decreased throughput. SNOW is the first step towards performance-optimal read-only transactions by answering the question of how to make latency-optimal designs.

**NOCS**

NOCS is the second and last step towards performance-optimal read-only transactions by making latency-optimal designs also throughput-optimal. Throughput is important because higher throughput proportionally reduces the required size of the storage system—i.e., number of physical machines—which in turn saves money and energy. Again, instead of making intermediate solutions for improving throughput, I seek designs that achieve the highest possible throughput while maintaining the lowest possible latency.

NOCS defines three properties—one-round communication (O), non-blocking operations (N), and constant metadata (C)—to represent optimal performance because they are exactly the properties simple,
non-transactional read operations have. Simple read operations present a throughput upper bound for read-only transactions, which are defined to provide stronger semantics than simple reads, and thus naturally cannot have higher throughput than simple reads.

These properties also embrace the latency properties from SNOW. For instance, one-round communication requires one-round on-path messages—i.e., the O property in SNOW—and no messages incurred by read-only transactions but off the critical path of reads. The off-path messages do not affect the latency of reads but decrease throughput because they consume system resources. The non-blocking property affects both latency and throughput. One reason the new designs in SNOW decrease throughput is because the size of their metadata increases linearly \( w \cdot r \cdot t \), the number of concurrent operations.

I then prove the NOCS Theorem, which states that no read-only transaction algorithm can achieve both optimal performance and strict serializability—i.e., N, O, C, and S properties cannot coexist. Using the NOCS Theorem as a guide, I design performance-optimal read-only transactions for two systems, Eiger [6] and ScyllaDB [10]. The new system Eiger-NOCS demonstrates the improvements compared to Eiger’s existing read-only transaction algorithm, i.e., achieving lower latency and higher throughput. ScyllaDB does not support transactions, and thus I design a performance-optimal algorithm for it from scratch. The new system Scylla-NOCS demonstrates the overhead of optimal read-only transactions is minimal compared to Scylla’s non-transactional reads.

My dissertation work closes a line of research on improving the performance of read-only transactions in distributed storage systems. Instead of looking for better performance, I design systems in which the read-only transactions have the best possible performance, i.e., lowest latency and highest throughput.

**Future Directions**

**Towards Optimal Transactional Systems**

Making read-only transactions optimal is the first step towards optimal transactional data stores. Next, I will explore optimality for write transactions and general read-write transactions, which together with read-only transactions construct a complete picture of transactional storage systems.

Improving write transactions has a large impact on user experience and the entire system. Firstly, many applications contain many writes in their workloads, e.g., banking systems and online shopping systems. Secondly, even for read-dominant workloads, the performance of writes may have a disproportionate effect. For instance, one slow write may block all concurrent read operations on the same machine, which may result in a slowdown on the entire system especially when the blocked machine contains popular objects. Thirdly, writes have to be replicated for fault tolerance, and thus inefficient writes may have a cascading negative impact, which can be worse if writes are geo-replicated across continents.

There intuitively exists a tradeoff between the performance and consistency of write transactions because their concurrency control protocols, e.g., 2PL and OCC, result in complex implementations. To find optimal designs, I plan to apply the insights from SNOW and NOCS, i.e., I will drive the design of write transactions to perform as well as simple, non-transactional writes, and then aim for stronger consistency guarantees while keeping the best performance. If I find a situation where the best performance cannot be achieved under certain consistency guarantee, then it suggests a fundamental tradeoff between optimal performance and consistency of write transactions.

General transactions, which contain both write and read operations, are used in many applications because they provide the most powerful semantics. In NOCS, I consider throughput-optimal designs together with the latency-optimal properties from SNOW to make performance-optimal read-only transactions. I will apply a similar methodology to studying optimal general transactions by combining the results from optimal read-only and optimal write transactions, which are special cases of general transactions.
Highly Scalable Data Stores with Write Omission

Given the ever-rapid increase in user base, scalability is critical to today’s web services. Higher scalability enables the system to accommodate the increasing user requests more easily with lower cost, i.e., by adding fewer machines. Write operations are often the bottleneck of a system’s scalability because they require more work—i.e., to update system state—compared to reads, and they have to be replicated before responding to the user for fault tolerance. Both reasons make writes consume more system resources and take longer. Their longer duration makes writes more likely to conflict with concurrent operations, which may further cause performance degradation.

In Scylla-NOCS, I co-design the read-only transaction protocol with its underlying multi-versioning subsystem, which allows me to safely omit applying some writes if the transaction layer guarantees that these writes will never be observed by any users. Write omission saves system resources for committing and replicating those writes thus increasing throughput, and also eliminates the possibility of conflicting with other operations thus reducing the overhead of concurrency control, which may have a greater impact if writes are transactional. As a result, the evaluation of Scylla-NOCS even beats non-transactional read operations in the original ScyllaDB in terms of scalability when workloads are skewed because Scylla-NOCS saves a lot of work by not committing some writes.

I plan to explore the benefits of write omission and generalize it to both transactional and non-transactional systems. I want to understand how to co-design multiple layers to enable the most write omission, how aggressively writes can be omitted, and if/how the write omission approach can be applied to replication subsystems, i.e., selectively choosing writes to replicate.

A Hybrid Distributed Operating System for Disaggregated Hardware

Today’s distributed storage systems use monolithic servers as their basic operating unit. A monolithic server is a box that contains all the hardware components required to run an application, i.e., CPU, memory, and disk. The servers communicate with each other via network in a datacenter while hardware resources inside a server are constrained within the boundary of the box. Two major drawbacks are associated with this monolithic setting. Firstly, hardware resources are underutilized. For instance, consider a simplified example where a task running on a monolithic server is using 99% of the memory but 1% of the disk. Because the memory is the bottleneck, we cannot assign more tasks to the server, which ends up wasting 99% of the disk space. Secondly, the fate sharing of hardware components within a server causes dependent failures, i.e., if one component fails, then the entire server stops working.

To overcome these drawbacks, people have proposed a disaggregated setting, which breaks the boundary of a monolithic server and collocates hardware resources based on their type, e.g., a rack of CPUs, a rack of memory. Different types of hardware devices are connected via network. This disaggregated approach has been gradually adopted by industry, e.g., Facebook’s datacenters move disks out of monolithic servers while still keeping CPU and memory in the same box. The completely disaggregated setting also brings new, challenging questions. Firstly, how is latency kept low when running an application has to go through network to reach required hardware? Secondly, how can the disaggregated setting be made scalable? Existing work focuses on building rack-level systems; however, making systems scalable to the entire datacenter is more useful to the upper-level applications. Thirdly, how should short-lived tasks, e.g., popular Lambda serverless applications [3], be handled given that process initialization has high overhead for allocating resources? Fourthly, how should failures be tolerated when server boundary disappears? Server is the unit block of fault tolerance techniques, e.g., Paxos, in monolithic setting. I am interested in working on these challenging research problems.

I plan to adopt a hybrid approach to solving the challenges, i.e., finding a sweet spot in the design space between two extremes: monolithic server and completely disaggregated setting. In particular, I am interested in designing a distributed operating system, which allows short-lived tasks to run within a box to ensure low latency, and allows an application to dynamically reach out to hardware resources outside the box to better utilize resources. Compared to monolithic servers and complete disaggregation, this hybrid approach has the benefits from both worlds while mitigating their drawbacks.
References


