

# BigBFT: Scaling BFT without Compromising Fault Tolerance via State Sharding

Guangda Sun

sung@comp.nus.edu.sg  
National University of Singapore

Jialin Li

lijl@comp.nus.edu.sg  
National University of Singapore

## Abstract

Scalability remains a major challenge for Byzantine fault tolerance (BFT) systems, whose throughput is often limited by sequential transaction processing at each node. Prior work has attempted to address this challenge by full sharding or by replacing total ordering with serializable concurrent execution, but these approaches introduce coordination overhead, complex protocols, or weakened security and isolation guarantees. In this work, we propose BigBFT, a BFT transactional store that scales by sharding the state but not the execution. BigBFT keeps a single consensus protocol to order transactions globally, while multiple node shards replicate state partitions and process state access in parallel. In this design, the consensus protocol circumvents complicated concurrency control in prior works. Unlike Basil and sharded blockchains, BigBFT scales without weakening fault tolerance or absence of Byzantine behavior. A preliminary prototype outperforms a fully replicated BFT store, and we expect the advantage to grow as the design is fully realized.

## CCS Concepts

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Information systems** → *Distributed storage*.

## Keywords

Byzantine fault tolerance, state sharding, transactional storage, distributed systems

## ACM Reference Format:

Guangda Sun and Jialin Li. 2026. BigBFT: Scaling BFT without Compromising Fault Tolerance via State Sharding. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '26)*, September 17–18, 2026, Bangkok, Thailand. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3838177.3841729>



This work is licensed under a Creative Commons Attribution 4.0 International License.

APSys '26, Bangkok, Thailand

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2912-6/26/09

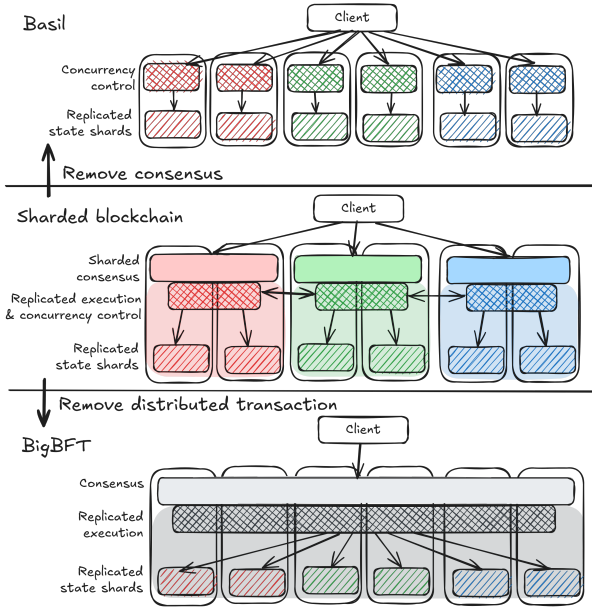
<https://doi.org/10.1145/3838177.3841729>

## 1 Introduction

Byzantine fault tolerance (BFT) systems [2, 3, 12, 13, 28, 30, 34] provide consistent shared state across mutually untrusted parties. The ability to tolerate Byzantine faults makes BFT systems a fundamental building block for applications with strong fault tolerance requirements, such as distributed ledgers, supply chain management, and collaborative machine learning. The seminal approach in BFT systems is the replicated state machine (RSM) [26], which maintains a total order of all operations through a consensus protocol [3] and applies them deterministically to the shared state. However, the RSM approach faces a scalability bottleneck, as the system performance is bounded by the processing capacity of a single node. To further scale performance, a BFT system needs to distribute load across participating nodes; at the same time, it must sustain high performance in the presence of Byzantine faults [5] and various workload conditions.

Recent work has attempted to address this scalability challenge by sharding the state and operations [1, 9, 14, 16]. Sharding, however, introduces cross-shard communication and coordination overhead and weakens security or isolation guarantees. To further improve scalability, transactional BFT stores propose to concurrently process non-conflicting operations and serialize them via concurrency control instead of totally ordered replication. Basil [30] applies a client-coordinated optimistic concurrency control protocol and achieved significantly higher throughput ceiling and lower latency than sequential processing. However, Basil suffers from high replication factors ( $5f + 1$ ) and cryptographic overhead, and its performance relies on unrealistic assumptions such as the absence of Byzantine behavior. A single Byzantine client can reliably degrade system performance by more than 20%, which largely defeats the purpose of BFT [5]. Basil's fragile performance, together with the complexity of its protocol, leads us to revisit the traditional RSM approach: Can a system still benefit from a consistent total order while executing transactions in parallel?

In this work, we propose a novel approach to scaling BFT systems by sharding the state but not the execution. We observe that recent protocols [2, 7, 12, 28] can linearly scale the throughput of BFT atomic broadcast with the number of nodes while maintaining low latency. This shows that totally ordering transactions is no longer the performance



**Figure 1: Comparison of different approaches to scaling BFT systems. The shaded areas are replicated.**

bottleneck. Moreover, previous work has shown that state access accounts for more than 80% of total execution time [19]. Together, we hypothesize that substantial performance gains are feasible when state access is sharded and parallelized, while consensus and execution are not. Based on these insights, we propose sharding the state data and balancing state access load across node shards, while relying on a BFT consensus protocol to totally order transactions globally. Our design enables better scalability by decoupling state management from transaction execution, while introducing zero additional coordination overhead. By retaining totally ordered execution, the approach avoids the fault tolerance and workload dependency compromises in previous solutions.

We illustrate the different scaling approaches in figure 1. Basil eliminates global consensus and relies on clients to coordinate distributed transactions, so its node shards must tolerate arbitrarily many Byzantine clients, which leads to performance and security compromises. Instead, we show that BFT systems can be scaled by removing distributed transaction coordination while preserving the consensus protocol and replicated state partitions. Our approach provides better fault tolerance, as the global BFT consensus isolates the performance impact of Byzantine clients and servers. Our main novelty is that the replicated state shards are concurrently accessed by all the servers without costly distributed transaction coordination. The state shards are only accessed internally by the servers. Because a majority of servers are correct and replicate the same state access sequence, they can

be treated as a single *logical* client submitting transactions per block execution. Consequently, the shards can remain a simple data plane without handling concurrent transactions or Byzantine clients.

Following this philosophy, we design BigBFT, a robust, high-performance BFT transactional store that leverages total ordering and data sharding. BigBFT replicates each state partition with dedicated node shards, each of which can tolerate  $f$  Byzantine faults among a total of  $3f + 1$  nodes. This scheme is identical to that of Basil and prior sharded blockchains, preserving the same state maintenance complexity. However, in BigBFT, nodes from all shards participate in the *same* consensus protocol, which orders all transactions before they are executed by *all* nodes. BigBFT can employ any modern BFT consensus protocol to atomically broadcast transaction blocks. The BigBFT nodes then execute blocks similarly to Basil’s clients: they first fetch state data from the corresponding shards, then execute transactions and buffer updates locally, and finally submit state updates. The nodes wait for  $f + 1$  matching state query responses to ensure the integrity and freshness of the fetched data, while the shards wait for  $2sf + 1$  matching updates before applying them to the state, where  $s$  is the number of shards. BigBFT’s design lies between Basil and sharded blockchains in terms of modularity. Compared with Basil’s monolithic approach, BigBFT is more modular because it only modifies the state management layer of a blockchain to address the most critical scalability bottleneck; compared with sharded blockchains, BigBFT modifies the blockchain core directly, thereby avoiding the performance and expressiveness pitfalls of full sharding.

We present the full design of BigBFT in this paper. We plan to implement BigBFT and evaluate its performance in comparison with Basil and sharded blockchains, and we expect BigBFT to achieve comparable performance to Basil in the ideal case and significantly outperform the other systems in the presence of Byzantine faults or skewed state accesses.

## 2 Motivation

The goal of this work is to design a BFT transactional store that can robustly scale throughput with the number of nodes while maintaining low latency. In this section, we first define the problem and then review prior scaling approaches and their limitations.

### 2.1 Problem Definition

BFT transactional stores provide mutable shared key-value state across mutually untrusted parties. They support transactions that dynamically read and write multiple keys, where the accessed keys are not known until execution. Performance-wise, BigBFT favors transactions with predetermined access

key sets. BFT stores ensure strict serializability, so transactions appear to execute sequentially in a way that is consistent with their real-time order from the clients' perspective [15]. BiGBFT provides a blockchain-style interface, where clients submit transactions that call on-chain smart contracts encapsulating multiple read and write operations, while it also supports Basil-style client-executed transactions by integrating multi-version values into the state.

BiGBFT can be scaled out by adding more node shards, where each shard is a fault-tolerant group of nodes that replicates a partition of the state data. Each shard can tolerate  $f$  Byzantine faults among a total of  $3f + 1$  nodes. BiGBFT can tolerate arbitrary Byzantine clients, while transactions are only guaranteed to be committed on correct clients. BiGBFT can ensure strict serializability in asynchronous networks, while its liveness depends on the synchrony assumptions of the underlying consensus protocol. It can guarantee liveness under asynchronous networks with asynchronous consensus protocols such as Bullshark [28].

## 2.2 Previous Scalable BFT Store

In traditional distributed storage systems, state data is partitioned, and transactions that access different partitions are processed in parallel by different node shards [4, 6, 11, 27, 33]. However, in blockchain systems, every node must participate in state machine replication, maintain the full state, and process all transactions in a consistent sequential order [26], even when the transactions are independent and could be processed concurrently. To similarly realize parallel transaction execution, sharded blockchains run multiple consensus instances in parallel, where each instance is operated by a different group of nodes, called a committee, and is responsible for a different state partition. Transactions that access different partitions are processed in parallel by different committees, while cross-shard transactions are handled by dedicated distributed transaction protocols. While sharding can parallelize transaction processing across committees, each committee still processes transactions sequentially, and the number of concurrent transactions that can be processed is limited by the number of shards even when all transactions are independent. Moreover, as the number of shards increases, cross-shard transactions become more common and dominate performance, and the distributed transaction protocols introduce significant overhead and complexity.

Basil [30] proposed strengthening traditional distributed storage systems with Byzantine fault tolerance and theoretically allowing for unlimited parallelism in transaction processing as the traditional systems. However, it relies on complicated client-coordinated optimistic concurrency control protocols to resolve transaction dependencies and ensure

serializable execution, incurring various performance penalties. Basil requires a  $5f + 1$  replication factor to prevent Byzantine nodes from arbitrarily aborting transactions. To scale out Basil with  $f = 1$ , it needs at least 6 nodes compared to 4 nodes in BiGBFT, increasing the cost by 50%. Basil's throughput is also highly limited by its expensive cryptographic overhead. Basil must sign a reply message for each transaction with public key cryptography, which not only means 10s of thousands of signatures per second to sign for each node, but also assumes powerful clients that can verify these signatures. Basil offers batched reply optimization, but only up to 16 replies to avoid severe network amplification to clients, while introducing additional accumulated authentication overhead. On the other hand, BiGBFT does not need to sign client replies with public key cryptography, and all publicly verifiable messages in BiGBFT operate at block level encapsulating arbitrary number of transactions and do not incur wasted network bandwidth.

In addition to being suboptimal, Basil's performance also requires strong assumptions of the absence of Byzantine nodes and clients, and low contention workloads. It is impractical for a BFT system to assume there is no Byzantine behavior occurring [5]. Moreover, Basil finds it difficult to support sophisticated database primitives such as range reads with its client-driven transaction execution, and its follow-up work [31] adapts server-side execution for this purpose, a similar approach to BiGBFT at a higher layer. In contrast, BiGBFT can always execute arbitrary business logic on the server side, inherently as in blockchains, which makes it easier to support database operations.

## 3 Design

BiGBFT is a scalable key-value store that shards the key space into multiple partitions, stores each partition on a separate node shard, and tolerates Byzantine faults within each shard. BiGBFT can be dynamically reconfigured to add or remove node shards. We omit the details of sharding scheme, reconfiguration, and state migration for simplicity, since many existing solutions are applicable given the consensus protocol employed by BiGBFT. BiGBFT provides an interface similar to blockchains. Clients submit transactions that call on-chain smart contracts, similar to stored procedures, executing which may read and write multiple keys. BiGBFT guarantees ACID properties. All transactions submitted by correct clients will eventually be committed with atomic and serializable execution, and all reads return the values of the most recent writes to the corresponding keys.

### 3.1 Design Overview

BiGBFT consists of three primary components: consensus, stateless transaction execution, and sharded state storage.

Every node in BiGBFT is multiplexed across all three components. The consensus component employs a BFT consensus protocol to accept transactions from clients and to order them globally. The consensus protocol in BiGBFT is not sharded: nodes across all shards participate in the same protocol. When shards join or leave BiGBFT, the consensus protocol is reconfigured to include additional or fewer nodes. When there are  $s$  shards, each with a disjoint set of  $3f + 1$  nodes where at most  $f$  are Byzantine, the consensus protocol is configured to tolerate  $sf$  Byzantine faults among a total of  $s(3f + 1)$  nodes, which is feasible because common consensus protocols can tolerate  $sf$  Byzantine faults with at least  $3sf + 1$  nodes. BiGBFT can integrate any BFT consensus protocol as the consensus component; those with scalable performance, such as DAG-based protocols [28] and Autobahn [12], are preferred to avoid introducing scalability bottlenecks. When using these protocols, clients can submit transactions to any node of any shard and commit transactions after receiving  $sf + 1$  matching confirmations from the nodes, ensuring at least one confirmation from a correct node.

The consensus component sequentially delivers blocks of transactions in a consistent order to the execution component, which executes the transactions and confirms to the clients after the writes are durably stored in the state storage component. BiGBFT adopts fully replicated execution: every node executes every transaction regardless of its shard, achieved through stateless execution. When executing a block, a node fetches the read values from the state storage component and buffers the writes locally. Although general transactions are supported, BiGBFT favors transactions with read sets known in advance, which enables batched fetching of all transactions in the block and improves efficiency. The node then posts the writes back to the state storage component and confirms to the clients after the post completes.

### 3.2 State Storage and Access

The state storage component exposes fetch and post interfaces to the execution component. Like the node shards in Basil and sharded blockchains, in BiGBFT the state storage component is sharded, and each shard is responsible for a disjoint partition of the key space. Reads and writes are consistently directed to the corresponding shards via consistent hashing [17]. Each fetch and post operation is divided into multiple network requests based on the accessed keys, and requests are sent to the corresponding shards in parallel. This parallelism is key to enabling concurrent state access in BiGBFT.

The state storage component handles fetch and post requests on a per-block basis. For fetch requests, it simply returns the current values of the requested keys. For post requests, it waits for  $2sf + 1$  matching requests from distinct

nodes to ensure a quorum of correct nodes have posted, then applies the writes to the state, acknowledges all requests, and begins responding to fetch requests for the following blocks. Because the consensus component is at the front, the state storage component in BiGBFT is much simpler than Basil's shard replicas, since it effectively serves only one client: the execution component of the majority of correct nodes. Therefore, it is designed as a simple data plane that does not need to handle concurrent or faulty clients.

In parallel, to fetch values from a state storage shard, the execution component sends fetch requests to all nodes in the shard and waits for  $f + 1$  matching responses to ensure at least one is from a correct node, thereby ensuring the integrity and freshness of the fetched values. To post values to a state storage shard, the execution component sends post requests to all nodes in the shard and waits for  $2f + 1$  matching acknowledgments, ensuring that at least  $f + 1$  correct nodes have applied the writes and are ready to serve fetch requests for the following blocks, which is necessary for the liveness of subsequent fetches.

### 3.3 Correctness and Liveness Proof

We now sketch the correctness and liveness of BiGBFT and defer the formal proof to the full version of the paper. The atomicity and durability of transactions are guaranteed by the state machine replication mechanism, so we focus primarily on consistency, serializable isolation, and liveness. The key insight is that the fetch and post operations served by the state storage component are always equivalent to having a full state copy on each node, as in normal BFT state machine replication, so the execution results are also equivalent to normal BFT systems, which are known to guarantee the desired properties. Specifically, fetch and post operations always complete, and fetch operations always return the values of keys as they appear in the most recent posts containing those keys.

As described earlier, both the execution and state storage components wait for  $f + 1$  and  $2f + 1$  quorums of matching messages, respectively. For each fetch request, the  $f + 1$  responding nodes must overlap with the  $2f + 1$  nodes that acknowledged the post request from which the fetched value originates, guaranteeing at least one correct node in the intersection and thus ensuring the correctness of the fetched value. For liveness, post requests require only  $2f + 1$  acknowledgments, so  $f$  Byzantine nodes cannot prevent them from completing. Moreover, at least  $f + 1$  of those  $2f + 1$  acknowledgments must come from correct nodes, which will eventually respond to subsequent fetch requests for values in the posted updates, ensuring liveness of fetch operations.

We omit the discussion on garbage collection in this short paper. Correct nodes that are slower than  $2sf + 1$  others may

attempt to fetch a block after the state storage has already overwritten the fetch values with subsequent post updates. In this case, the state storage component can respond with a certificate indicating that these nodes can safely skip executing the block, since at least  $sf + 1$  correct nodes must have executed the block and confirmed to the clients, ensuring the transactions committed.

### 3.4 Optimizations

*Caching.* When state access happens locally, caching is widely deployed in multiple tiers of the system. In BigBFT, while the state storage component can cache hot values locally, the execution component still pays the network overhead to access them. Thus, the execution component is designed to cache recently accessed key-value pairs locally. For workloads with highly skewed state access patterns, this cache can significantly reduce the number of keys to fetch for each block, bypassing the network for those accesses and achieving better performance.

*Pipelining.* In the design described above, only one of the execution and state storage components is active at a time: the execution component is idle while waiting for post acknowledgments, and the state storage component is idle while waiting for execution. To further improve performance, we propose pipelining the execution and state storage components, while the consensus component is already pipelined to both. While executing the current block, the execution component can concurrently post the writes of the previous block and fetch values for the next block. At the same time, the state storage component serves fetch requests for the next block after the previous block's updates are posted instead of the current block's. With this change, the state storage component responds to fetch requests with a fixed one-block staleness, so the execution component must maintain intermediate states between consecutive blocks to ensure execution correctness.

*Trusted state access.* Although logically separated, the three components are co-located on the same nodes, and components deployed on the same node share the same trusted domain. We can leverage this trusted state storage component to remove quorum collection for a node's own shard. When the execution component needs to fetch values from its own shard, it can directly access the local state storage component without sending network requests or waiting for multiple matching responses. Similarly, when the state storage component receives post requests from the execution component on the same node, it can directly apply the writes without waiting for multiple matching requests.

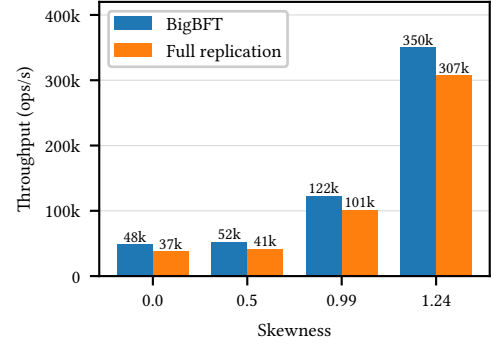


Figure 2: Throughput of BigBFT and the baseline under different workload skews.

## 4 Preliminary Results

We implement a prototype of BigBFT in Rust, with partial synchrony version Bullshark [29] as the consensus component and RocksDB [25] as storage. Here we present preliminary results of BigBFT comparing to a fully replicated baseline BFT store. We plan to conduct more comprehensive evaluations in the full version of the paper, including comparisons with Basil and sharded blockchains, and measuring more performance metrics such as latency and scalability with more shards and nodes.

In our preliminary evaluation, we deploy BigBFT and the baseline on 100 nodes. The state is a 100 GB key-value store with 100 million keys. In the baseline, each node stores the full state, while in BigBFT the state is sharded into 100 partitions, each replicated by a node shard of 7 nodes that tolerates up to 2 Byzantine nodes. BigBFT assumes disjoint node shards, so deploying this configuration as intended would require 700 machines. Constrained by our testbed, we instead colocate the 100 shards on the same 100 nodes, with each node belonging to seven shards. This is not the intended deployment model, but it exercises the same state access paths and quorum sizes, so we expect it to be broadly performance-equivalent for the fault-free workloads measured here.

In the evaluation, we run a read-only YCSB workload to stress BigBFT under a high network fetch load, which is the worst case for BigBFT. We also test different workload skews to understand the impact of caching, ranging from uniform ( $\alpha = 0$ ) and highly skewed ( $\alpha = 1.24$ ). The results are shown in figure 2. Unsurprisingly, BigBFT constantly outperforms the baseline in all scenarios with and without caching and under both uniform and skewed workloads, because BigBFT parallelizes state access and significantly reduces the number of overall disk access. In the baseline, every state access translates to 100 disk accesses across the 100 nodes, while in BigBFT only 7 nodes need to access the disk for it. Furthermore, due to caching, the performance of both systems

improves as the workload becomes more skewed. BigBFT can achieve the throughput of 350k transactions per second under a skewed workload, 1.14 $\times$  that of the baseline.

Unfortunately, the current implementation of BigBFT is not optimized and the performance scalability has not been fully realized, so the performance improvement over fully replicated baseline and previous works is not as significant as we expect. In the future, we will apply the optimizations described in §3 and expect to see the performance of BigBFT to scale with the number of nodes and significantly outperform the other systems.

## 5 Related Works

*Sharded blockchains.* SharPer [1], ByShard [14], GriDB [16] and BlockchainDB [9] are sharded blockchains with a similar security model to BigBFT. They propose to deploy multiple consensus instances on different sets of nodes, each tolerating up to  $f$  Byzantine faults. Other sharded blockchains, such as Omniledger [18], RapidChain [36], and Dang et al. [8], adopt a different security model where the systems tolerate a certain fraction of Byzantine nodes across the entire system, and the node shards are formed via random sampling at runtime. As discussed in §2, both categories of sharded blockchains suffer from performance degradation due to cross-shard transactions. Basil [30] discussed more shortcomings of sharded blockchains in detail.

*BFT scalability.* Many prior work have focused on scaling BFT consensus protocols, including SBFT [13], HotStuff [34], Tusk [7], Bullshark [28], Autobahn [12], and Mysticeti [2]. Some of them also adopt sharding techniques, such as Elastico [22] and OHIE [35]. While these works significantly improve the scalability of BFT consensus, the transaction execution is still sequential and fully replicated, becoming the bottleneck of the system.

*Ordering, commutativity, and determinism.* EPaxos [23] and CURP [24] deliberately leave commutative operations unordered, and in fail-stop replication this pays off, since consensus dominates the cost and bypassing it saves round trips. Basil takes the same step into the Byzantine setting: it is the upward branch of figure 1, trading the global order for client-coordinated concurrency control at the price of the replication factor and cryptographic overhead discussed in §2. BigBFT deliberately takes the downward branch instead, because both sides of the tradeoff invert under Byzantine faults. Ordering has become cheap: DAG-based BFT protocols [2, 7, 12, 28] sustain a total order whose throughput grows with the number of nodes. Forgoing it has become expensive, since concurrency control must then stay correct against arbitrarily misbehaving clients rather than merely

crashed ones. We therefore treat a total order not as an obstacle to parallelism but as an enabler of it, in the spirit of deterministic databases [21, 32], which fix a transaction order up front so that partitions can execute against it independently and still converge. BigBFT uses BFT consensus as precisely such a deterministic concurrency control policy, which is what lets the bottom row of figure 1 have every server access every state shard concurrently with no coordination: all correct nodes derive the identical state access sequence from the ordered blocks, so no shard can diverge.

*Blockchain execution engines.* LVMT [19], COLE [37], Block-STM [10] and ParallelEVM [20] improve the performance of blockchain execution engines by optimize IO efficiency and parallelizing execution. They improve the performance of individual nodes and can be used together with BigBFT as scalably performant BFT systems.

## 6 Conclusion

In this paper, we present BigBFT, a scalable BFT system that practices a novel architecture design to achieve high performance. BigBFT demonstrates that consensus protocols can be regarded as efficient concurrency control mechanisms, and sharded state access is the key to improve the scalability of BFT systems. We will further optimize the implementation of BigBFT and conduct more comprehensive evaluations in the future.

## Acknowledgement

We thank the anonymous reviewers for their valuable feedback. This work was supported by the Singapore Ministry of Education Academic Research Fund Tier 1 grant T1 251RES2104.

## References

- [1] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 international conference on management of data*. 76–88.
- [2] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2025. Mysticeti: Reaching the Latency Limits with Uncertified DAGs. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society.
- [3] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (New Orleans, Louisiana, USA) (OSDI '99)*. USENIX Association, USA, 173–186.
- [4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Debora A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
- [5] Allen Clement, Edmund Wong, Lorenzo Alvisi, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine

- Faults. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. USENIX Association, Boston, MA.
- [6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
  - [7] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.
  - [8] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.
  - [9] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1597–1609.
  - [10] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2022. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. arXiv:2203.06871 [cs.DC] <https://arxiv.org/abs/2203.06871>
  - [11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. Association for Computing Machinery, Bolton Landing, NY, USA. doi:10.1145/945445.945450
  - [12] Neil Girdharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 1–23.
  - [13] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.
  - [14] Jelle Hellings and Mohammad Sadoghi. 2021. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2230–2243.
  - [15] Maurice P. Herlihy and Jeannette M. Wing. [n. d.]. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July [n. d.]). doi:10.1145/78969.78972
  - [16] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. 2024. GriDB: Scaling blockchain database via sharding and off-chain cross-shard mechanism. *arXiv preprint arXiv:2407.03750* (2024).
  - [17] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.
  - [18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.
  - [19] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. 2024. LVMT: An efficient authenticated storage for blockchain. *ACM Transactions on Storage* 20, 3 (2024), 1–34.
  - [20] Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. 2025. ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems*. 211–225.
  - [21] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.
  - [22] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 17–30.
  - [23] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 358–372.
  - [24] Seo Jin Park and John Ousterhout. 2019. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 47–64.
  - [25] rocksdb 2012. RocksDB persistent key-value store. <https://rocksdb.org/>.
  - [26] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990). doi:10.1145/98163.98167
  - [27] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.
  - [28] Alexander Spiegelman, Neil Girdharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.
  - [29] Alexander Spiegelman, Neil Girdharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The partially synchronous version. *arXiv preprint arXiv:2209.05633* (2022).
  - [30] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.
  - [31] Florian Suri-Payer, Neil Girdharan, Liam Arzola, Shir Cohen, Lorenzo Alvisi, and Natacha Crooks. 2025. Pesto: Cooking up High Performance BFT Queries. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 529–554.
  - [32] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.
  - [33] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*. 307–320.
  - [34] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM symposium on principles of distributed computing*. 347–356.
  - [35] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 90–105.
  - [36] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 931–948.
  - [37] Ce Zhang, Cheng Xu, Haibo Hu, and Jianliang Xu. 2024. {COLE}: A column-based learned storage for blockchain systems. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 329–345.