IONIA: High-Performance Replication for Modern Disk-based KV Stores

Yi Xu, *University of California, Berkeley;* Henry Zhu, *University of Illinois Urbana-Champaign;* Prashant Pandey, *University of Utah;* Alex Conway, *Cornell Tech and VMware Research;* Rob Johnson, *VMware Research;* Aishwarya Ganesan and Ramnatthan Alagappan, *University of Illinois Urbana-Champaign and VMware Research*

https://www.usenix.org/conference/fast24/presentation/xu

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings of the 22nd USENIX Conference on File and Storage Technologies is sponsored by

IONIA: High-Performance Replication for Modern Disk-based KV Stores

Yi Xu[‡§]
University of California, Berkeley

Henry Zhu[‡]
University of Illinois Urbana-Champaign

Prashant Pandey
University of Utah

Alex Conway
Cornell Tech and VMware Research

Rob Johnson
VMware Research

Aishwarya Ganesan
University of Illinois Urbana-Champaign and VMware Research

Ramnatthan Alagappan
University of Illinois Urbana-Champaign and VMware Research

Abstract. We introduce IONIA, a novel replication protocol tailored for modern SSD-based write-optimized key-value (WO-KV) stores. Unlike existing replication approaches, IONIA carefully exploits the unique characteristics of SSD-based WO-KV stores. First, it exploits their interface characteristics to defer parallel execution to the background, enabling high-throughput yet one round trip (RTT) writes. IONIA also exploits SSD-based KV-stores' performance characteristics to scalably read at any replica without enforcing writes to all replicas, thus providing scalability without compromising write availability; further, it does so while completing most reads in 1RTT. IONIA is the first protocol to achieve these properties, and it does so through its storage-aware design. We evaluate IONIA extensively to show that it achieves the above properties under a variety of workloads.

1 Introduction

Key-value stores play a central role in datacenter applications. Today, many KV stores such as LevelDB [31], RocksDB [26], Cassandra [2] and others [8, 15, 62, 64] are built using *write-optimized indexes* (WOIs) such as LSMs [58]. We refer to these stores as WO-KV stores. WO-KV stores have become a popular choice because they offer significantly higher write performance than B-tree stores [9, 62]. Further, recent WO-KV stores are optimized for modern SSDs, and can extract their high bandwidth and offer low latencies [73].

As KV stores are increasingly deployed in datacenters, making them fault tolerant is critical. A common way to achieve this goal today is to replicate the store on many machines and use off-the-shelf replication protocols like MultiPaxos [43], Raft [56], or Viewstamped Replication (VR) [46] to coordinate writes and reads to the store. For example, ZippyDB at Meta [65, 68] uses MultiPaxos to replicate RocksDB. Several other systems use a similar layered design [11, 20, 21, 47].

Unfortunately, however, using off-the-shelf protocols to replicate modern WO-KV stores squanders their high write performance (§2). These protocols offer low write throughput because they must apply writes sequentially on a single thread to ensure that the replicas are identical [66]. They also incur high latencies because replicas must coordinate to agree on the order of writes. Although many prior protocols [13, 37, 42] have been proposed to safely apply writes on multiple threads for high throughput, they incur high latencies. At the same time, many prior approaches [28, 44, 59, 61] that achieve low-latency writes suffer from low throughput. Existing replication approaches thus cannot preserve the high write performance of WO-KV stores.

Off-the-shelf protocols also lead to poor read performance. For strong consistency, these protocols restrict reads to a designated replica called the leader [38, 52, 55]. Thus, the read bandwidth of the followers goes unused. This is particularly bad for WO-KV stores because of their read-write asymmetry [9]: reads are more expensive than writes and thus performance drops with more reads. As a result, serving all reads at the leader pushes it to a less performant regime, impairing overall throughput. Many prior protocols have devised ways to scalably read from followers. However, as we discuss (§2), many of them suffer from high latencies [11, 74]; others that offer low latencies do so by (regrettably) trading off write availability and slowness tolerance [14, 27, 39, 70].

An ideal protocol must preserve the high write performance of WO-KV stores, offering high throughput and low latency. It must also safely (i.e., with consistency) scale reads while offering low-latency reads without impacting availability. We observe that a main reason why existing approaches do not achieve these ideal properties is that they are largely *oblivious* of the underlying SSD-based WO-KV store's characteristics.

In this paper, we design IONIA, a novel replication protocol that carefully exploits the interface and performance characteristics of the underlying SSD-based WO-KV store. We show that such careful storage-aware design enables IONIA to achieve high-throughput, 1RTT writes, and scalable, 1RTT reads without impacting availability (§3).

IONIA avoids the high latency of writes inherent in prior parallel-execution protocols [13, 37, 42] by exploiting the interface atrributes of WO-KV stores, improving the write path. In particular, WO-KV stores convert all writes into *blind* writes to avoid performing a slow read before a write. Consequently, they do not return an execution result but only an acknowledgment to clients when writes complete; i.e., the update interfaces in WO-KV stores are nil-externalizing, a

[‡]Co-primary authors
[§]Did part of the work during an internship at VMware Research

property we identify in our prior work on Skyros [28]. Thus, similar to Skyros, IONIA needs to only guarantee durability when writes complete, which can be achieved in 1RTT without coordination. IONIA then defers ordering and parallel execution to the background, achieving both high-throughput and 1RTT writes.

IONIA's main novelty is its scalable, 1RTT read path. We first observe that existing low-latency, scalable read protocols conflate scalability and locality due to their focus on in-memory stores. Locality means that reads can be locally served by any replica without additional messages to other replicas. This conflation, however, compromises write availability and slowness tolerance: because clients must be able to locally read at any replica, writes are replicated to *all* replicas (not just a quorum). Locality is a necessary condition for scalability for in-memory stores, where network messages are the bottleneck. In contrast, we realize that for SSD-based stores, *scalability can be decoupled from locality*: because SSD is the bottleneck (instead of network), even non-local reads that send additional messages won't impact scalability as long as these additional messages access only in-memory state (not the SSD) on other replicas.

Based on this insight, IONIA replicates only to a quorum for availability while allowing reads from any replica. Then, to handle lagging followers, for every follower read, IONIA performs a check (which we call a meta query) at the leader to validate the result returned by the follower. However, this approach can still scale because the leader serves meta queries from memory at high throughput. Overall, compared to prior protocols that focus on in-memory stores, IONIA exploits the performance gap between SSDs and DRAM to decouple scalability from locality, achieving scalable reads.

Second, while the meta-query approach offers scalability, it in itself does not offer 1RTT reads, a requirement for modern SSD-based stores that offer low latencies. To achieve 1RTT reads, IONIA sends the read to a follower and the meta query to the leader in parallel. However, a challenge is that, since the requests are concurrent, the leader cannot directly indicate to the client whether or not the follower's result is up-to-date. To solve this problem, IONIA proposes a *client-side consistency-check* mechanism, where the leader returns enough information about the key being read and the client makes the decision about the freshness of the follower's result.

We have designed and implemented IONIA (§4). A main challenge in our design is to ensure that meta queries can always be served from the leader's memory. IONIA addresses this by maintaining a compact history of recently modified keys instead of all keys in the store. A related challenge is how to ensure that the leader returns the correct information for meta queries for keys not present in the history. IONIA solves this problem by returning slightly inaccurate information for such keys but without endangering consistency. We have model checked IONIA to show its correctness.

Our evaluation (§5) shows that for writes, IONIA matches the throughput of parallel-execution protocols while offering the low-latency of Skyros; IONIA also approximates the performance of an unreplicated server. For reads, IONIA offers linear scaling, saturating the read bandwidth of all replicas without meta queries becoming the bottleneck. With mixed workloads, IONIA offers 1.8× higher throughput than IONIA-LR (a variant where reads are restricted to leader). We show that most reads finish in 1RTT and that this is achieved with small histories (e.g., 50MB). With YCSB [19], IONIA improves throughput by 16× to 38× over MultiPaxos.

This paper makes three contributions.

- We first show how designing a replication protocol by paying attention to the underlying SSD-based WO-KV store layer yields desirable properties.
- Second, we present novel ideas such as decoupling scalability from locality and client-side consistency checks, which enable IONIA to achieve scalable and low-latency reads without compromising availability or slowness tolerance.
- Finally, we present a thorough experimental evaluation, showing IONIA's benefits.

2 Background and Motivation

We provide background on WO-KV stores and building replicated KV stores. We discuss why existing protocols are insufficient for WO-KV stores, leading to undesirable properties.

2.1 WO-KV Stores Background

KV stores are implemented using a disk-based index structure. Stores built using B-tree or its variants [57] are a poor fit for write-intensive workloads because writes in B-trees require random IO, which is significantly slower than sequential IO. As a result, modern KV stores have turned towards write-optimized indexes (WOI) such as LSMs [58] or B^{ε}-trees [12]. WOIs offer higher write throughput than B-trees because they batch writes and sequentially transfer large batches to disk, amortizing IO cost [9]. Today, many local KV stores including LevelDB [31], RocksDB [26], and several others [60, 62, 64, 73] are built atop WOIs. WO-KV stores are also used as storage engines in distributed systems like BigTable [15], Cassandra [2], and CockroachDB [17].

Read-Write Asymmetry. In B-trees, both writes and reads require random IO and thus both are limited by device's random IOPS [9]. WOIs provide the same read performance as B-trees [9] and are limited by random IOPS. However, writes in WOIs are limited by sequential bandwidth. Consequently, WOIs have a *read-write asymmetry* in performance: writes in WOIs are much faster than reads.

An important implication of this asymmetry is that KV stores built using WOIs avoid *query-before-update* [9]; issuing a query before an update squanders the benefits of WOIs, essentially making WOIs behave like B-trees. As a result, WO-KV stores convert all writes into blind writes. For example, a *put* simply absorbs a write to a key without checking if the key is already present since the check would

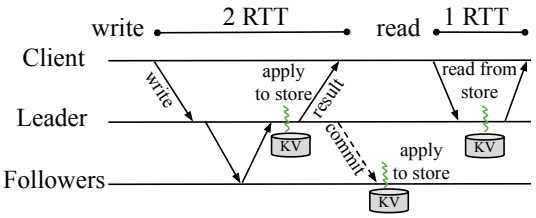


Figure 1: **Off-the-Shelf Replication.** *The figure shows how writes and reads are processed in replicated stores built using off-the-shelf protocols.*

require a (slow) read [50]. Similarly, a *delete* blindly inserts a tombstone without checking for key presence [67]. Even read-modify-writes (RMW) are transformed into blind upserts [9]; as an example, RocksDB implements upserts using merge [25]. An upsert encodes a RMW by specifying a key k and a function f that changes the value of k. k and f are then blindly recorded; the value is evaluated later only when needed (e.g., on a read). This, in turn, means that when writes complete, WO-KV stores do not reveal system state to the clients [28]: they do not return an execution result or execution error (e.g., to indicate key presence or absence) but only an acknowledgment. Our prior work calls such updates nil-externalizing or nilext [28]. Note that while a nilext interface does not return *execution* errors, it can still return validation errors (e.g., malformed requests, unreachable server).

2.2 Consistent Replicated KV Stores Background

A common way to build a strongly consistent replicated KV store is to layer an off-the-shelf replication protocol like MultiPaxos atop the local KV store. Many systems [11, 20, 21, 47] including Meta's ZippyDB [65, 68] use such a layered design.

Figure 1 shows how writes and reads are processed in a replicated KV store built using off-the-shelf replication protocols such as MultiPaxos (or Raft [56] or VR [46]). As shown, clients submit writes to the leader, which orders the requests by appending them to its consensus log; the leader then sends the request to the followers. The followers append to their logs and respond. Once a majority has agreed to the order, the leader applies the writes to the KV store and returns the result. Asynchronously, the leader sends a *commit*, upon which all followers apply the ordered writes. All replicas apply writes sequentially on a single thread to avoid non-determinism. Applying writes on multiple threads is non-deterministic [42]: replicas may apply writes in different orders, causing the KV store state across replicas to diverge.

Because followers could be lagging and thus could return stale data, off-the-shelf protocols allow reads only at the leader to ensure strong consistency [38, 48, 52, 55]. To prevent a deposed old leader from serving stale data, these protocols employ leader leases [38, 46], which ensures that a new leader is elected only after the old leader's lease has expired.

2.3 Why Are Existing Protocols Insufficient?

We first explain the drawbacks of off-the-shelf protocols to replicate SSD-based WO-KV stores. Prior work has built optimized protocols to improve over off-the-shelf replication. However, as we discuss, these protocols are still not the ideal choice for SSD-based WO-KV stores, leaving a huge room for improvement in performance and availability.

	Protocol/System	Write throughput	Write latency	Scalable reads	Read latency	HA writes	Slowness tolerance
	Unreplicated	high	1RTT	×	1RTT	×	×
	MultiPaxos [43], Raft [56], VR [46]	low	2RTT	×	1RTT	√	√
Improving Writes	Parallel execution (e.g., CBASE [42], Eve [37])	high	2RTT+Δ	×	1RTT	√	√
Improving Writes	Specpaxos [61], CURP [59], Skyros [28]	low	1RTT	×	1RTT	√	√
Improving Reads	Gaios [11]	low	2RTT	√	2.5RTT + Δ	√	√
Improving Reads	CRAQ [70]	high	(n/2)RTT*	√	1RTT	×	×
Improving Reads	Hermes [39]	high	2RTT	√	1RTT	×	×
	IONIA	high	1RTT	√	1RTT	√	√

Table 1: **Existing Approaches.** *The table compares different existing approaches. Δ: waiting delay for bigger batches (in parallel execution) or followers to catch up (in Gaios); *: CRAQ builds upon chain replication [71] which incurs O(n) RTTs for writes; n: number of replicas.*

We will use an unreplicated server as a baseline to show the drawbacks of existing protocols. As shown in Table 1 (first row), an unreplicated server offers low latency for writes and reads because clients can submit a request to the server and get a response in 1RTT. The unreplicated server can also offer high throughput because it can safely apply writes on multiple threads. However, the unreplicated server has an obvious problem: it cannot tolerate server failure or slowness. Further, read performance is limited by the single server.

An *ideal* system must tolerate failures and slowness while matching the unreplicated server's latency and write throughput. It must also scale read performance with replicas.

2.3.1 Off-the-Shelf Protocols are Ill-Suited for WO-KV

Off-the-shelf protocols like MultiPaxos tolerate failures and slowness (Table 1 second row). However, these protocols offer significantly lower throughput than an unreplicated server because they must apply writes sequentially on a single thread to avoid non-determinism. Modern WO-KV stores, however, are optimized for multi-core CPUs and SSDs [18, 26], where ingesting data using many threads is necessary for high throughput. These protocols also incur high latencies for writes because the replicas must coordinate to order the writes in the critical path. Specifically, writes incur 2RTTs (client→leader→followers→leader→client), doubling the latency of an unreplicated server (client→server→client). Modern WO-KV stores offer low latencies [18] and thus latency from additional RTTs is undesirable. As we soon show, such eager coordination is, in fact, unnecessary for WO-KV stores.

Off-the-shelf protocols restrict reads to the leader. Because the leader sees all writes, with leader leases [46], reads are guaranteed to see the latest data, completing them in 1RTT.

However, this means that reads cannot scale with replicas. Restricting reads to the leader also reduces *overall* throughput because reads contend with writes. This is particularly bad for WO-KV stores because of read-write asymmetry, where performance drops with more reads. If the leader processes both reads and writes, the ratio of reads to writes will be higher than a system that splits the read load across replicas. Since WO-KV stores offer lower performance with more reads, the leader will operate at a lower throughput regime than a system that spreads reads. Since the leader's throughput is the bottleneck, the overall performance suffers.

2.3.2 Shortcomings of Existing Improvements

We now discuss the vast body of work that has attempted to improve the write and read performance of off-the-shelf protocols. We discuss their shortcomings and argue why they are not a great fit for replicating WO-KV stores.

High-Throughput *Or* Low-Latency Writes. As we discussed, applying writes on multiple threads is essential to realizing high throughput in modern WO-KV stores. Fortunately, prior protocols such as CBASE [42], Eve [37], and others [1, 13, 24] (third row) have shown how to leverage multi-threaded execution in replicated systems while avoiding inconsistencies. For correctness, these protocols concurrently execute only non-conflicting writes; conflicting writes are serialized and applied in the same order across all replicas. Thus, these protocols can offer high write throughput and could serve as a good base for replicating WO-KV stores. Unfortunately, however, these protocols incur high latencies similar to off-the-shelf protocols. In fact, the latency can be higher than off-the-shelf protocols because these protocols must create bigger batches to find opportunities to concurrently apply many writes. While prior protocols have proposed techniques for low-latency writes by exploiting commutativity [44, 53, 59], speculation [41, 61], network ordering [45], and interface semantics [28] (fourth row), these protocols apply writes sequentially and thus suffer from low throughput. Existing protocols to improve write performance thus either suffer from high latencies or low throughput.

Scalable Reads: Write Availability *Or* Low Latency. Prior work has proposed ways to scale reads by allowing reads at all replicas. The main challenge these protocols must solve is to ensure that a read from a replica returns the most up-to-date data; a stale read will violate strong consistency [33].

Systems like Gaios [11] and Gnothi [74] (fifth row) ensure consistency by routing all reads to the leader; the leader then dispatches the read to a follower. The follower next waits until it has caught up with the leader before serving the read. While this approach scales reads, it incurs several RTTs and waiting delay. This was acceptable for spinning disks where disk latency was much higher than RTTs. However, for modern SSD-based WO-KV stores, incurring multiple RTTs increases latency considerably compared to an unreplicated server.

A few protocols [14, 27, 30, 39, 70] achieve scalability while offering 1RTT reads (sixth row). These protocols target in-memory stores and thus, network IO is often the scalability bottleneck. Therefore, for scalable 1RTT reads, these protocols ensure a read at a replica is always *local*: a client locally reads at a target replica without additional messages to other replicas. However, to enable local reads, these protocols must replicate writes to *all* replicas (not just a quorum) before writes can complete. This conflation of scalability and locality, however, has a critical impact on availability because writes cannot complete if any replica fails [30][†]. Further, they cannot exclude slow replicas when processing writes. We thus observe a *fundamental tradeoff* in these protocols: they offer scalable reads with low latency but do so at the expense of availability and slowness tolerance.

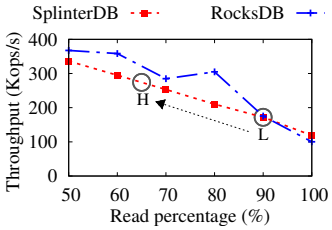


Figure 2: **Read-Write Asymmetry.** *The figure shows how performance decreases with more reads in WO-KV stores.*

Summary. Off-the-shelf protocols suffer from poor performance and thus are not suitable for building replicated WO-KV stores. While many solutions have been proposed to improve writes, they do not achieve high throughput and low latency simultaneously. A few solutions have been proposed to achieve scalable reads. These protocols either compromise on latency, or tradeoff write availability and slowness tolerance for low latency. Further, protocols that improve writes suffer from poor read performance (e.g., parallel-execution protocols do not scale reads); similarly, scalable read protocols do not offer optimal write performance (e.g., CRAQ incurs O(n) and Hermes incurs 2RTTs for writes).

We next show how a protocol that carefully exploits the characteristics of the underlying SSD-based WO-KV store can advance beyond existing approaches and offer high-throughput 1RTT writes, and scalable 1RTT reads without compromising on availability or slowness tolerance (last row).

3 IONIA Ideas and Protocol Overview

We now describe the key insights behind IONIA and provide an overview. The next section presents the detailed design.

3.1 Key Insights and Ideas

Deferred Parallel Execution with Immediate Durability. IONIA achieves high-throughput writes with low latency. To achieve high throughput, IONIA employs techniques from prior parallel-execution protocols and concurrently executes

[†]CRAQ and Hermes must wait for an expensive reconfiguration to complete before the system can become available after failures.

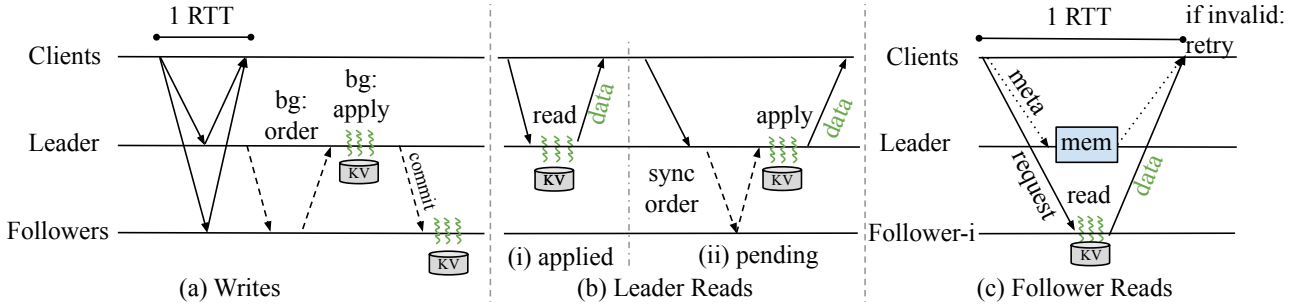


Figure 3: **IONIA Protocol Overview.** *The figure shows how* IONIA *handles writes, leader reads, and follower reads.*

only non-conflicting writes to avoid inconsistencies. However, existing high-throughput protocols incur high latencies. We realize this latency cost can be avoided by exploiting the interface semantics of WO-KV stores. Because WO-KV stores convert all writes into blind writes, they do not return an execution result; i.e., the writes are nilext. Thus, similar to Skyros [28], IONIA need not immediately order and apply writes to the underlying WO-KV store. Instead, it must only guarantee that writes are durable before acknowledgment. IONIA achieves durability in 1RTT by having the clients write to many replicas in parallel without coordination across replicas. Ordering and parallel execution of writes are deferred to the background. Thus, IONIA combines the benefits of prior parallel-execution and low-latency approaches to achieve both high-throughput and 1RTT writes.

Decoupling Scalability from Locality. Existing low-latency, scalable read protocols conflate scalability and locality as they focus on in-memory stores. To achieve scalability, they require that reads are local (no additional messages to other replicas). Locality is necessary for scalability in in-memory stores where network messages are the bottleneck. This conflation, however, impacts availability and slowness tolerance. We realize that in SSD-based WO-KV stores, SSD random IOPS is the bottleneck, not the network messages. Thus, reads can scale even if they are non-local (i.e., they send additional messages to other replicas), as long as the additional messages *access only in-memory state (not the SSD)* on other replicas. That is, scalability can be *decoupled* from locality in SSD-based stores.

Based on this insight, IONIA writes only to a quorum for availability and slowness tolerance and allows reads at any replica. The challenge, however, is that a read at a replica cannot rely on just the local state because the replica might be lagging. IONIA addresses this challenge as follows. For every read at a follower, IONIA sends a meta query to the leader (which is guaranteed to have seen all writes). The meta query helps identify whether or not the result returned by the follower is up-to-date.

A critical requirement for scalability is that meta queries must be cheap; in particular, meta queries must avoid SSD IO. Otherwise, the meta-query throughput could saturate before the replicas' collective SSD IOPS are saturated, limiting scalability. We soon discuss how IONIA can always serve meta queries from memory without SSD IO (§4). Two factors help ensure that in-memory meta queries will not become the bottleneck in practice. First, reads in WO-KV stores are limited by random IOPS, which is ∼600K IOPS [35] in today's fast SSDs. On such hardware, even with caching and skewed workloads, modern WO-KV stores offer only about 850K reads/s [18]. Second, most systems typically use a small replication factor (3 or 5) [34]. However, even an unoptimized server in our setup could handle 12M meta-query-like RPCs/s, which is well over the collective read bandwidth ($5 * 850K$). Thus, meta queries will not be the scalability bottleneck.

By spreading the read load, IONIA also improves overall performance under mixed workloads. In WO-KV stores, as reads increase, the performance drops due to read-write asymmetry as illustrated in Figure 2. Thus, under mixed workloads, taking off reads from the leader and spreading it across replicas, reduces its read-write ratio which improves leader's throughput. For example, consider a workload with 90% reads and 10% writes; this is a low performance regime for WO-KVs (L in Figure 2). If the 90% reads are split across five replicas, the leader would process 18% reads and 10% writes (i.e., read-write ratio is 65:35), which is a higher performance regime (H). Overall, distributing reads improves the leader's throughput; since the leader's storage layer is the bottleneck, improving it boosts the overall performance.

Low-latency Reads via Client-side Consistency Checks. While the meta-query approach enables scalability, it in itself does not ensure 1RTT read, which is important for modern SSD-based stores that offer low latencies. To achieve 1RTT and scalable reads, our main idea is to have clients send the meta query to the leader and the actual read to a follower *in parallel*. However, this raises a challenge – the leader cannot definitively determine whether or not the data returned by the follower is up-to-date. This is because the leader does not know the follower's status at the time the read was performed at the follower. To solve this, IONIA uses a novel *client-side consistency check*, where the leader returns information about the key being read, and the client makes the final decision about the freshness of the follower's data. If the client determines that the data is stale, it retries the read. Client-side checks offer 1RTT reads while ensuring strong consistency.

3.2 IONIA Protocol Overview

We now provide an overview of IONIA. Figure 3(a) shows the write path. Clients send writes to all replicas in parallel. The replicas make the writes durable without any coordination and respond directly to clients. Once a client receives enough replies including one from the leader, the write is complete; clients can proceed without waiting for the writes to be ordered or applied. Thus, all writes complete in 1RTT. The 1-RTT write path is similar to Skyros [28].

While IONIA does not order and apply writes in the critical path, writes must be eventually ordered and applied in the real-time order to preserve linearizability [33]. That is, if operation y starts after another operation x completes, then y must be ordered after x. However, when concurrently executing requests, non-conflicting writes (or, writes that update different keys) can be applied in any order across replicas. Only conflicting writes that update the same key must be applied in the same real-time order across replicas. IONIA ensures such correct write ordering in the background. The leader periodically determines the order for the writes and gets enough followers to accept the order; all replicas then apply conflicting writes in the order determined by the leader in the background.

We next discuss the read path (see Figure 3(b) and (c)). Reads can be served by any replica. When a read to a key arrives at the leader, there might be updates to the key that have not been ordered and executed yet. This is because IONIA orders and executes writes only lazily. Thus, in IONIA, before the leader can serve a read, it must first check if there are pending updates to the key being read. If no, the leader serves the read immediately (3(b)(i)). If there are pending updates, then the leader synchronously orders and executes the updates before serving the read (3(b)(ii)). Fortunately, such slow-path reads are rare in practice due to two reasons. First, the leader keeps ordering and executing writes in the background; thus, in most cases, updates are executed already by the time a read arrives. Second, traces from deployed systems [23] show that reads to recently written objects are rare [28].

In IONIA, reads are served by followers as well. To prevent clients from seeing stale data from lagging followers, in addition to sending the actual read request to a follower, clients also send a meta query to the leader as shown in 3(c). The follower locally reads its KV store and returns the kv pair. The leader responds to the meta query with information about the key being read (in particular, the latest update applied to the key). IONIA ensures that the leader can get this information from memory without an SSD IO. The client uses this information to decide whether or not the result from the follower is valid. If valid, the read completes in 1RTT; in contrast, if the data is stale, the client retries the read at the leader.

4 IONIA Design and Implementation

We use viewstamped replication (VR) as a baseline to describe IONIA's design. VR is leader-based and makes progress through a sequence of views. In each view, one replica serves as the leader. VR tolerates f failures with $2f+1$ replicas and offers linearizability. Upon writes, the leader appends requests to a *consensus log* and sends a *prepare* to the followers. Once f followers acknowledge via a *prepare-ok* (after adding to their logs), the leader applies the writes and returns the result to the client. Reads are served by the leader and the system uses leader leases for consistency. The replication layer interacts with the KV store via upcalls; writes are applied via the *Apply* upcall and reads are served via the *Read* upcall.

```
MakeDurable(w) // add write w to durability log
AddToExecQueuesWithDeps(B)
// add batch to execution queues with dependencies
Apply(w) // apply write w to KV store
LeaderRead(k) // returns data, must_sync
MetaQuery(k)
// returns must_sync, modified_index for k
FollowerRead(k) // returns data, applied_index
```

Figure 4: **Storage-system Upcalls in IONIA.**

IONIA is also leader-based and offers the same guarantees as VR. IONIA augments the interface between the replication layer and the storage system with additional upcalls as shown in Figure 4. These upcalls help IONIA handle different operations. We first explain how IONIA handles writes (§4.1) and reads (§4.2) during normal operation. We then describe how IONIA handles failures and view changes (§4.3). Finally, we present correctness proof sketch (§4.4), model checking results (§4.5), and implementation details (§4.6).

4.1 Writes

IONIA's goal is to achieve high-throughput writes with low latency (1RTT). Our idea to achieve this end is to apply writes on multiple threads but avoid high latency by deferring ordering and execution of writes to the background. IONIA can defer ordering and execution because WO-KV stores do not return an execution result. IONIA must only ensure that writes are durable before acknowledgment (i.e., they will not be lost even if f replicas fail).

To achieve low-latency durability, IONIA borrows the idea of durability logs from Skyros [28]. IONIA clients directly send writes to all replicas. Each replica adds the write to a separate durability log (via the *MakeDurable* upcall); the replicas then respond directly to clients without any coordination, completing writes in 1RTT. Intuitively, the durability logs contain writes that are not yet ordered and applied. A client waits for a supermajority ($f + \lceil f/2 \rceil + 1$) acknowledgments including one from the leader to complete a write. Because the leader's response is required to complete writes, the leader's durability log captures the correct (real-time) ordering. The leader uses this property to order requests in the background during normal operation. When the current leader fails and a new one is elected, supermajority quorums enable IONIA to reconstruct the linearizable order as we soon discuss (§4.3).

Although the leader's durability log captures the correct

order, the order is *not finalized* yet as writes might be present in different orders in durability logs across replicas. This is because writes are added to durability logs without any coordination. Thus, the leader must finalize the order before the writes can be applied to the KV store. To do so, the leader periodically gets a majority of replicas (including self) to agree on the order. The leader adds requests from its durability log to its consensus log in order and sends a *prepare*. The followers append the requests to their consensus logs and respond with *prepare-ok*. Once *f* followers respond, the ordering is finalized. The leader can now apply the writes to the store. The leader also sends a *commit* which informs the followers of the latest consensus-log index up to which the order has been established; the followers can apply writes up to that index. To improve throughput, the leader batches several requests in a single *prepare*; however, this batching does not affect client-visible latency since the ordering (and execution) happen entirely in the background.

Once ordering is established, each replica applies writes to the store in multiple threads. The replicas execute only non-conflicting writes in parallel. Conflicting writes are executed serially and in the order they appear in the consensus log; this ensures that conflicting writes are executed in the same order across replicas. IONIA realizes the above idea as follows. Each replica maintains a set of execution queues, one for each execution thread; an execution thread applies writes from its queue in order. However, before applying a write *w*, IONIA must ensure that all writes that conflict with *w* and appear before *w* in the consensus log have been executed. To capture and set these dependencies, the replication layer invokes *AddToExecQueuesWithDeps* when a batch of writes has been ordered. The storage layer captures conflicting writes to the same key by adding writes to a particular key to the same queue in the order they appear in the batch; the replicas use a deterministic hash of the key to achieve this. Other conflicts between requests can be captured by explicitly annotating a request *w* with requests from other queues that must be executed before *w*. For example, a multi-key write can be added to any queue with explicit dependencies to the other requests in the queues that conflicts with *w*.

To execute requests, each execution thread retrieves a request *w* from its queue and waits until the dependencies of *w* are executed before it executes *w*. Once a replica has applied a request to the KV store, it removes it from its durability log. The replica also updates its *applied-index*, the latest index in the consensus log up to which it has applied to the KV store. The leader keeps learning each follower's applied-index. Figure 5 shows how writes are ordered, assigned to execution queues, and finally applied to the KV store.

4.2 Reads

IONIA's goal is to provide scalable, 1RTT reads without impacting availability. As we discussed earlier, to ensure strongly-consistent reads, IONIA issues a meta query to the leader to check the validity of a read at a follower. IONIA ensures scalability by serving the meta queries from the leader's memory while the reads at the replicas are bound by SSDs' random IOPS. Further, to achieve 1RTT reads, IONIA clients send the actual read request and meta query in parallel and employ a client-side check to validate the returned data. We now explain how the meta query and client-side consistency check mechanisms work. We then describe how IONIA ensures that the leader can always serve the meta query from its memory by maintaining a compact history.

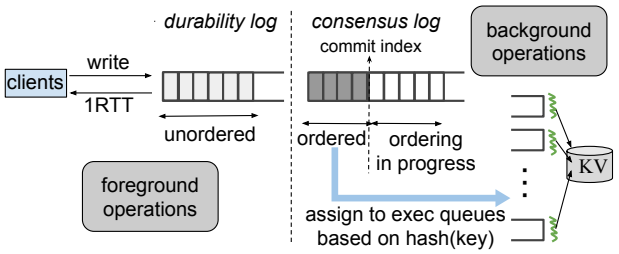


Figure 5: **IONIA Write Processing at a Replica.** *The figure shows how writes are processed in one replica (other replicas are not shown).*

4.2.1 Meta Queries and Client-side Consistency Check

Reads at the leader can be served directly without any cross-replica checks because the leader is guaranteed to have seen all writes. However, when a read arrives at the leader, there might be unordered (and hence not-yet-applied writes) in its durability log. Thus, upon a read, the leader first checks if there are such pending updates to the key being read (via the *LeaderRead* upcall). If there are no pending updates, the leader reads and returns the kv pair, finishing the read in 1 RTT. If there are pending updates, the upcall returns a *must_sync* flag. The replication layer then synchronously appends the pending writes from the durability log to its consensus log, gets the followers to agree, applies the writes to the KV store, and finally reads and returns the kv pair. In this case, the read completes in 2 RTTs. However, such synchronous reads are rare as we discussed (§3) and will show in our evaluation.

Reads at the followers require a check at the leader because followers could be lagging. To read at a follower, a client sends the read to the follower and a meta query to the leader. To enable 1RTT reads, both requests are sent in parallel. The meta query specifies the key being read. Upon receiving a read, the follower reads the kv pair from the store (via the *FollowerRead* upcall) and returns it. To answer meta queries, the leader maintains a *history*; this history maps a key to the consensus-log index corresponding to the latest write that modified the key. Upon a meta query, the leader first checks its durability log to see if there are pending updates to the key *k* being read. If there are none, the leader queries the history and obtains the latest index *i* that modified *k*.

One way to implement the meta queries is to have the leader make the decision about freshness the of follower's data. Specifically, the leader can compare the follower's applied-

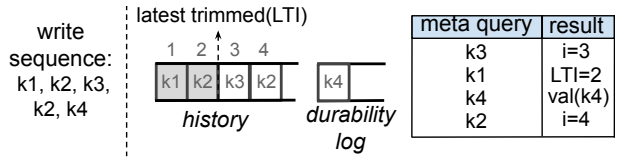


Figure 6: **History and Meta Queries at Leader.** *Writes completed: k1, k2, k3, k2, k4; history trimmed up to 2, so LTI=2. Meta queries for k3 and k2 return the actual modified index; query for k1 returns LTI because k1 is not in history; query for k4 returns value of k4 after synchronous execution.*

index a_f (which it learns periodically) and i. If $a_f > i$, then the leader could decide that the follower is to up-to-date. But this approach is *incorrect*. This is because the follower could return an older value, then apply a later update, and then update a_f at the leader. Now, if the meta query reaches the leader, it would incorrectly determine that the follower's result is valid, violating strong consistency.

To address this issue, IONIA pushes the check to the client. In addition to the kv pair, the follower also returns its own *applied-index*. Note that the follower reads its applied-index a *before* reading from the KV store, so that the kv pair returned is never older than a, even with concurrent writers. The leader, instead of making a decision locally, returns the latest index i that modified k. The client then performs the check by comparing a and i. If $a >= i$, then the follower has applied all the updates to k and thus the returned data is up-to-date, finishing the read in 1RTT. If $a < i$, then the follower has not applied all updates to k and thus the data is stale. In such cases, the check fails and the client retries the read at the leader.

When there are pending updates in the durability log for key being read, the leader knows that the follower's data will be stale and could instruct the client to retry. However, this would increase latency. IONIA optimizes this case by having the leader synchronously order and execute updates and return the actual read result. The client then ignores the result from the follower and uses the one from the leader.

4.2.2 Cheap Meta Queries with Compact History

The history at the leader logically needs to maintain the latest consensus-log index that modified every key. To ensure meta queries are fast, the history must be maintained in memory, not disk. However, maintaining the modified index for *every* key in memory is impractical for large disk-based stores.

To solve this problem, IONIA maintains the history only for recently modified keys. A key is added to the history when a write to it is recorded on the consensus log. The leader periodically learns each follower's *applied-index*. When all followers have applied upto i, the history upto i is trimmed. This raises a problem, however: when a meta query arrives for a key k, the history may not contain k. Note that a key k being absent from the history means that all followers have applied all writes to k. However, the leader still *cannot* indicate to the client that the data from the follower is up-to-date. This is because, the leader doesn't know which version the follower returned. Specifically, the follower could have returned an older version and then applied the latest update, causing the leader to trim the history before the meta query arrives.

What must the leader return when k is not in the history? Intuitively, the leader must return an index greater than or equal to the actual modified index. Returning anything smaller is unsafe: the client may incorrectly believe the follower has given the latest data. Thus, when the leader does not find a key in its history, it returns the index that was last trimmed from the history; we call this the last-trimmed index or *lti*. Returning *lti* is correct, because if the modified index was part of the trimmed history, *lti* would be greater than or at least equal to the modified index. Figure 6 shows how the leader maintains the history and returns results for meta queries.

Optimizations. IONIA uses two techniques to optimize the procedure described so far. First, it uses lazy history trimming. The history could be trimmed up to i *immediately* after the leader learns that all followers have applied up to i. However, such eager trimming can lead to inefficiencies. For example, consider a case where a client reads at a follower and get an *applied-index* a. After this, the follower applies m more writes and informs the leader of its *applied-index* $a+m$; the leader trims history upto $a+m$. If the meta query now arrives at the leader, the history would not contain the key. The leader would return $a+m$ (its lti), causing the client check to fail. To avoid such scenarios, IONIA only lazily trims the history.

Second, a follower that has been disconnected for long or failed could prevent the leader from trimming the history. To avoid this problem, the leader maintains a set called active-followers. Failed followers are removed from the set. The leader waits only for the followers in the active-followers set to trim the history. Thus, reads at a disconnected follower that has missed writes will always be rejected because its *applied-index* will be less than the index returned by the leader.

Summary. Figure 7 summarizes IONIA's read protocol. To read key k at the leader, the clients invoke LEADER_READ. If there are no pending updates in the durability log, the leader reads and returns k. If there are pending updates to k, the leader orders and applies them, after which it reads and returns k. When reading at a follower, the client parallely invokes FOLLOWER_READ and META_QUERY. The follower reads k and returns it along with its *applied-index*. In META_QUERY, the leader first checks if there are pending updates to k. If yes, the leader orders and executes the pending updates, and returns the actual data and indicates this in a flag. If there are no pending updates, the leader returns the modified index (if the key is in the history) or the last-trimmed index. The client finally compares the results and either returns the data to the end application, or retries the read at the leader.

4.3 Failures and View Changes

So far, we described IONIA's normal operation. We now discuss failures and view changes. IONIA is similar to VR with respect to both replica recovery [46, §4.3] and view changes [46, §4.2]. The only difference stems from IONIA's durability logs

```
1: procedure LEADER_READ(k)
2:     if pending request for k in durability_log then
3:         trigger sync ordering and execution
4:         wait till updates are executed
       return store.read(k)
---
1: procedure FOLLOWER_READ(k)
       return (store.read(k), applied_index)
---
1: procedure META_QUERY(k)
2:     if pending request for k in durability_log then
3:         trigger sync ordering and execution
4:         wait till updates are executed
5:         return (flag=data, value=store.read(k))
6:     if history.contains(k) then
7:         return (flag=index, value=history[k])
8:     return (flag=index, value=lti)          ▷ last trimmed index
---
1: procedure CLIENT_READ_AT_FOLLOWER(k)
2:     invoke_parallel(f_res = FOLLOWER_READ(k), l_res =
   META_QUERY(k))
3:     if l_res.flag == data then
4:         return l_res.value          ▷ leader returned actual result
5:     a = f_res.value                      ▷ follower's applied index
6:     i = l_res.value                ▷ latest index that modified k
7:     if a ≥ i then                        ▷ follower's result is valid
8:         return f_res.value
9:     return LEADER_READ(k)              ▷ invalid; retry at leader
```

Figure 7: **IONIA Reads.** *Summary of* IONIA*'s read protocol.*

that VR doesn't have. IONIA borrows durability logs from Skyros [28] and thus it inherits Skyros' recovery and view-change. We give a brief overview of these procedures.

In IONIA, when a replica recovers, in addition to recovering the consensus log from the leader of the latest view, a replica must also recover its durability log. This is straightforward: the leader sends its durability log (along with the consensus log as in VR) and the replica sets its durability log as the one sent by the leader. This is correct because the leader's durability log contains completed writes in the correct order.

A view change happens when the leader fails. The main challenge is that the new leader must recover the requests in the old leader's durability log and in the correct linearizable order. This is where supermajority quorums help. To see why, consider an *incorrect* protocol where updates are acknowledged after writing to a majority of durability logs. Suppose that update a completes after which b starts and completes (i.e., $a \rightarrow b$). Let D_i be the durability log of replica S_i. Then, a possible state is D_1:[ab], D_2:[ab], D_3:[ab], D_4:[ba], D_5:[ ]. Now, if S_1 (the current leader) and S_2 fail, then it is impossible to determine the correct order from the remaining durability logs. Writing to a supermajority ensures that, after f failures, at least $\lceil f/2 \rceil + 1$ (i.e., a majority within any available majority) will have the requests in the correct order.

During view change, a new leader in IONIA contacts a majority, collecting the requests in their durability logs. It then constructs the set of acknowledged writes, i.e., requests that are present on at least $\lceil f/2 \rceil + 1$ logs. The leader next establishes the order: for every pair of requests a, b, it examines if b appears after a on at least $\lceil f/2 \rceil + 1$ logs. If so, then it concludes b follows a. Such pairwise dependencies are added to a DAG G; IONIA produces the total order by topologically sorting G. These steps are similar to Skyros [28, §4.6]. In addition, the new leader in IONIA constructs the history over its consensus log and sets lti to the index of the first log entry.

Fallback Path. To ensure availability in cases where a supermajority is not available (but a bare majority is), IONIA falls back to a slow mode where writes are acknowledged only after being synchronously ordered on a majority in 2 RTTs.

4.4 Correctness Proof Sketch

Two conditions must hold for linearizability. **P1.** Writes must respect linearizable ordering. **P2.** Reads must never expose stale data. We provide a proof sketch of P1 and P2.

P1. During normal operation, the leader's durability log is guaranteed to have the completed writes in the correct order because a leader response is required for a write to be considered complete. The leader adds requests to the consensus log from its durability log in order; thus, the consensus log captures the linearizable order. IONIA replicas execute ordered writes in parallel. However, for correctness, conflicting writes must be executed serially and in the same order across replicas. The consensus log establishes a total order of writes. Conflicting writes are executed in the order in which they appear in the consensus log. The consensus logs are identical across replicas (this is ensured by base VR). Therefore, conflicting writes are executed in the same order across replicas.

When the current leader fails, a view change occurs, and the leader of the new view must recover the latest consensus log and the durability log. The recovered logs must reflect the correct linearizable order. This recovery procedure in IONIA is the same as Skyros, which ensures that the new leader reconstructs the correct ordering of writes in the both durability and consensus logs [28, §4.7].

P2. First, reads can go to the leader. Because the leader is guaranteed to have seen all completed writes, and because IONIA checks the durability log for pending writes, reads at the leader are guaranteed to see the latest data. We next discuss the correctness of reads performed at followers.

For linearizability, a read r to a key k must see the effects of all writes to k that completed before r started. Let w be the latest write to k. There are two cases to consider.

C1. First, w could have just completed but not executed yet. In this case, w will be present in the leader's durability log. The meta query will thus catch the pending write w in the durability log and return the latest data after execution. The client check ensures that r sees the latest data as it ignores the follower result in this case.

C2. If w was executed, then w may not be in the durability log and will be present in the leader's consensus log and history. Let w_i be the index of w in the consensus log; since

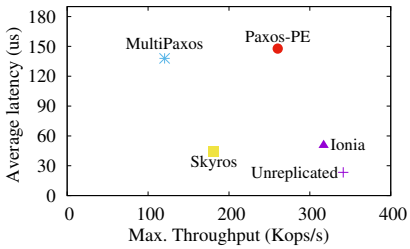


Figure 8: **Write-only Workload.** *The figure plots the maximum throughput and the average latency for a write-only workload;* IONIA, PAXOS-*PE, and unreplicated use 8 threads. The workload loads 350M records.*

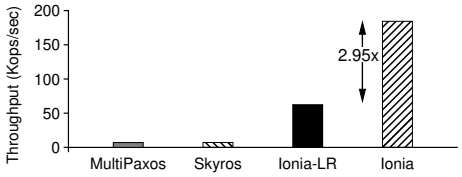


Figure 9: **Read-only Throughput.** *Maximum throughput in Multi-Paxos, Skyros,* IONIA-LR, *and* IONIA *under a uniform read-only workload.*

w is the latest write to k, w_i is k's latest modified index. For correctness, the index result returned by the leader for a meta query, res_i, has to be at least w_i, i.e., $res_i \geq w_i$. If the history was not trimmed, then k will be part of the history and the leader will return w_i. Instead, if the history was trimmed then k will not be part of the history. However, we argue why res_i will be greater than or equal to w_i even in this case.

Suppose there were m additional writes independent (i.e., non-conflicting) of w were executed and also the history was trimmed up to $w_i + m$. In this case, leader's last trimmed index, lti, will be $w_i + m$. When the leader doesn't find k in the history, it will return its $lti = w_i + m$, which is greater than w_i. If there were no additional writes and the leader trimmed up to w_i, then the leader's $lti = w_i$, which is safe.

We have established that the leader returns the correct index for a meta query. For final correctness, stale results from a follower must be correctly identified. If w_i was not applied at a follower, the applied-index, a, returned by the follower will be less than w_i. Since the leader's $res_i \geq w_i$, the client check will correctly discard the stale result from the follower.

4.5 Model Checking

We have model checked IONIA's request-processing and view-change protocols. We focus our discussion on IONIA's read protocol because write processing and view changes are similar to Skyros. We generated and explored over 8M different states (e.g., followers having applied up to different points, the leader's history being trimmed up to different points). Linearizability was met in all states.

Our checker finds violations when we intentionally introduce bugs. For example, we modified the model to skip the durability-log check before querying the history upon a meta query. This is unsafe because there could be a completed (but unexecuted) update V_2 in the durability log and follower could have only applied V_1. The leader will return the modified index of V_1, making the client incorrectly trust V_1. Our checker catches this violation. Similarly, the checker identifies a violation when the leader returns indexes lower than the modified index. Finally, we modified the model to have the leader perform the check instead of the client. In this incorrect version, the leader compares the follower's applied-index (a) and the modified index (i) and indicates to the client that data can be trusted if $a >= i$. However, this is unsafe: a client could read the stale version and by the time its meta query reaches the leader, the history could be updated. The leader would then incorrectly indicate to the client that the read is valid. Our checker catches this violation as well.

4.6 Implementation

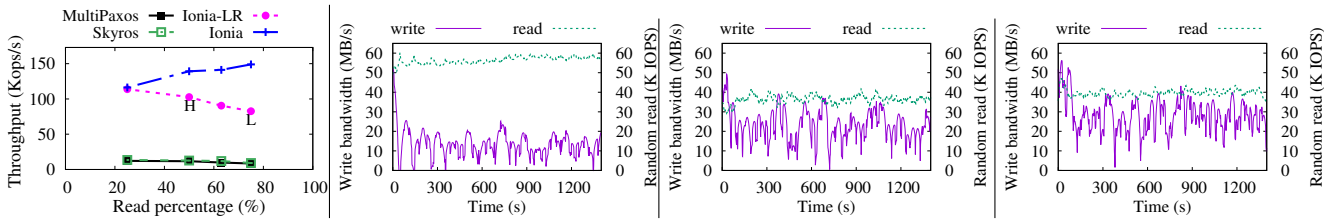
We provide key implementation details. IONIA replicas run SplinterDB [73] as the state machine and communicate via eRPC [36]. We implemented the upcalls in a wrapper, requiring no changes to SplinterDB. IONIA implements a highly concurrent durability log. The IONIA replication layer manages all threads, balancing foreground work (e.g., adding writes to durability logs) and background work (i.e., applying writes to SplinterDB). The leader maintains highly concurrent and compact inverse maps to lookup the durability log and history.

5 Evaluation

To evaluate IONIA, we ask the following questions:

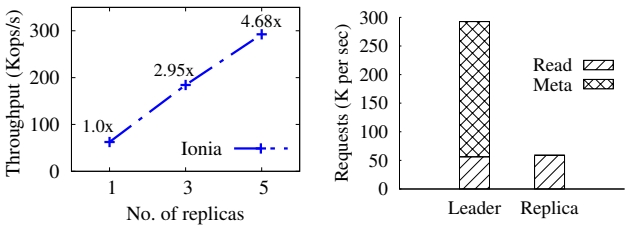
- How does IONIA perform compared to various existing replication approaches for write-only workloads? (§5.1)
- How does IONIA perform for read workloads? (§5.2)
- Does IONIA improve read and also overall performance (by taking off leader reads) under mixed workloads? (§5.3)
- Does IONIA scale throughput with replicas? (§5.4)
- Does IONIA offer low-latency reads? (§5.5)
- How does history size impact IONIA performance? (§5.6)
- How does IONIA perform on the YCSB benchmark? (§5.7)
- Does IONIA improve performance over unreplicated server for read and mixed workloads? (§5.8)
- How does IONIA perform under failures? (§5.9)

Setup. We run our experiments on Cloudlab with three replicas. We compare against: 1. off-the-shelf MultiPaxos (equivalently VR [46]), 2. Skyros [28], a recent low-latency protocol, 3. MultiPaxos with parallel execution (we build this baseline based on CBASE [42]) which we refer to as Paxos-PE, 4. IONIA-LR, an IONIA variant where reads are restricted to the leader, and 5. an unreplicated (fault-intolerant) server. All baselines use batching wherever possible to improve throughput. IONIA replicates SplinterDB. We integrated SplinterDB as the state machine in all baselines. Each replica uses an Intel DC S3520 SATA SSD. Unless specified, we use a 670M KV-pair dataset with 24B keys and 100B values. SplinterDB uses a 4GB cache (as prescribed [73]). For fairness, we modified all baselines to also use eRPC. Unless specified, MultiPaxos-PE, IONIA, IONIA-LR, and unreplicated use 15 threads.



(a) Mixed throughput (b) IO Util IONIA-LR 75R:25W (c) IO Util IONIA 75R:25W (d) IO Util IONIA-LR 50R:50W

Figure 10: **Mixed workloads.** *(a): throughput at various read fractions; (b)-(d): IO utilization of leader in* IONIA-LR *and* IONIA *at two read-write points.*



(a) Scaling With Replicas (b) Requests at Replicas

Figure 11: **Read Scaling.** *(a): read-only throughput with varying replicas. (b): meta queries vs. read requests at* IONIA*'s leader with five replicas.*

5.1 Write-Only Workload

We first analyze write-only performance of MultiPaxos, Skyros, Paxos-PE, and IONIA by comparing them to the an unreplicated server. For each system, we vary the number of clients and measure the maximum throughput and the corresponding average latency. Figure 8 shows the result. We make the following observations. First, as expected, unreplicated server offers high throughput and low latency. Second, MultiPaxos offers significantly lower throughput due to single-threaded execution; further, coordination in the critical path and batching increases MultiPaxos's latencies notably (up to 5.8×). Third, while Paxos-PE is able to achieve higher throughput by via parallel execution, its latency is still significantly higher compared to unreplicated (up to 6.3 ×). Skyros is able to closely match the low latency of unreplicated, but it still offers low throughput similar to MultiPaxos.

Overall, no existing protocol is able to match the high write performance of unreplicated. IONIA, in contrast, closely matches the unreplicated server's high throughput (via parallel execution) and low latency (by deferring ordering and execution to background). This shows that the fault tolerance provided by IONIA comes at no to little performance cost.

5.2 Read-Only Workload

We next analyze the performance of MultiPaxos, Skyros, IONIA-LR, and IONIA for a uniform read-only workload. Figure 9 shows the maximum throughput for each system. Standalone SplinterDB's read performance is limited by random IOPS. However, MultiPaxos use a single thread for executing reads and thus cannot generate the high queue depths needed to saturate SSD IOPS, resulting in lower throughput. Skyros suffers from the same low performance as MultiPaxos due to its single-threaded execution. IONIA-LR executes reads on multiple threads, thus achieving higher throughput, but measuring one level deeper reveals that the IOPS of leader's SSD is saturated. Thus, adding more load after this point doesn't yield higher throughput in IONIA-LR. In contrast, IONIA distributes the reads and extracts bandwidth of followers' SSDs too, providing 2.95× higher throughput than IONIA-LR, achieving essentially linear scaling with 3 replicas. We note here that the leader in IONIA performs meta queries upon every follower read, but this does not create a bottleneck because the meta queries are served from leader's memory.

5.3 Mixed Write-Read Workload

We now analyze mixed read-write workloads. Figure 10(a) shows the maximum throughput of MultiPaxos, Skyros, IONIA-LR, and IONIA for different read percentages with a uniform workload. MultiPaxos offers significantly lower performance than IONIA-LR. Although Skyros commits writes in 1RTT, it achieves only low throughput due to single-threaded execution. As reads increase, IONIA-LR's performance decreases. This is due to IONIA-LR's performance asymmetry (see Figure 2) as all reads go to the leader. IONIA improves performance in two ways by distributing reads. First, reads are served by all replicas, improving read throughput. Second, offloading some reads moves the leader to a more performant regime. IONIA has higher benefits with higher read fraction as more reads can be offloaded. For example, at 75%R, 25%W, IONIA offers 1.8× better throughput than IONIA-LR.

We show how the leader moves into a more performant regime by considering the 75%R, 25%W point. Figure 10(b) and (c) show the leader's IO utilization in IONIA-LR and IONIA, respectively, for this point. As shown in 10(b), IONIA-LR's leader serves more reads than writes; this is a low-performance operating point (denoted as *L* in 10(a)). IONIA splits the 75% reads across 3 replicas, and thus the leader processes equal amount of reads and writes (i.e., 50%-50%). This operating point of IONIA shown in 10(c) roughly resembles the 50%R, 50%W point of IONIA-LR's leader shown in 10(d), a more performant operating point (denoted as *H* in 10(a)).

5.4 Scaling Reads with Replicas

We next show that IONIA can scale reads for cluster sizes widely used in practice [34]. We run a read-only workload with 3 and 5 replicas. Figure 11(a) shows that the through-

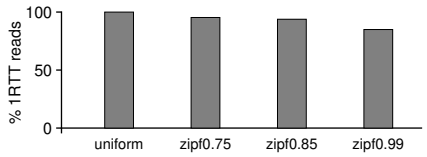


Figure 12: **Fast Reads.** *1-RTT reads for different distributions.*

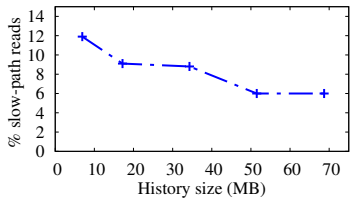


Figure 13: **Impact of History Size on Read Performance.**

put scales linearly. We examine the 5-replica case in more detail. Figure 11(b) shows the overall request throughput at the leader and the average throughput of followers for the 5-replica case. Although the leader receives 4× more load, IONIA is able to scale reads without the leader becoming the bottleneck because the leader serves this 4× greater load (meta queries) cheaply from its memory. Examining the IO utilization reveals that the IOPS on all replicas were saturated, indicating that the collective IOPS remains the bottleneck.

5.5 Low-Latency Reads

We now show that IONIA offers fast reads under different request distributions. A read may take more than 1RTT in two ways. First, a read of key k performed at the leader could trigger synchronous ordering and execution (because of a pending update to k). Second, a client may detect that the follower's result is stale and retry the read at the leader. Both these cases would happen more often with more skewed workloads. Figure 12 shows fraction of 1RTT reads for a read-write (50%-50%) mixed workload with different distributions. As expected, we observe no conflicts for uniform, and thus almost all reads finish in 1RTT. With a zipfian workload (skew factor $\theta = 0.75$ and $\theta = 0.85$), we see a slight decrease as some reads take more RTTs. However, even with a very skewed workload ($\theta = 0.99$), 85% of reads finish in 1RTT.

5.6 Impact of History Size

We next analyze the impact of history size. Intuitively, with a large history, the meta query will more often return the accurate last-modified index. With smaller histories, the leader may often return the last-trimmed index and thus more client checks will fail, resulting in many reads taking the slow path. To study this, we run a read-write (50%-50%) workload with zipfian distribution for various history sizes. As shown in Figure 13, as expected, slow reads decrease with larger histories. However, to achieve good performance in our experimental setup, a 50-MB history is sufficient, a meager 0.06% of the dataset (which contains 670M KV-pairs of 124B each, totalling about 77GB).

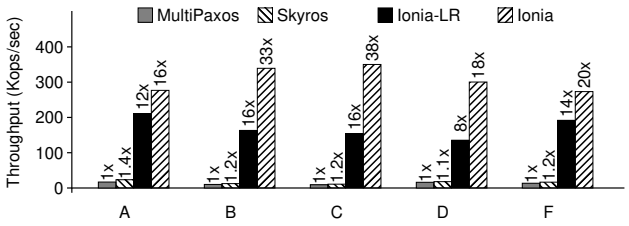


Figure 14: **YCSB Benchmark.** *Throughput under YCSB workloads. Number on top of each bar shows the performance normalized to MultiPaxos.*

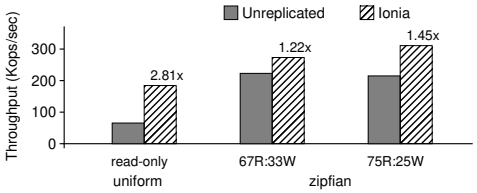


Figure 15: **IONIA vs. Unreplicated: Read and Mixed Workloads.**

5.7 YCSB Benchmark

We now show performance under YCSB [19] workloads: A (50%W, 50%R), B (5%W, 95%R), C (readonly), D (5%W,95%R), and F (50%RMW, 50%R); all workloads are zipfian except D which is latest. Figure 14 plots the throughput for MultiPaxos, Skyros, IONIA-LR, and IONIA. Under all workloads, MultiPaxos offers only low throughput. While Skyros offers marginally higher throughput than MultiPaxos in some workloads, it offers significantly (an order of magnitude) lower throughput compared to IONIA-LR as IONIA-LR employs parallel execution. IONIA preserves the write performance of IONIA-LR and improves over it by distributing reads. The improvement over IONIA-LR in write-heavy workloads (A and F) is roughly 23% to 42%. However, under read-heavy workloads (B, C, and D), IONIA improves performance by ∼2× over IONIA-LR by distributing the large fraction of reads.

5.8 IONIA vs. Unreplicated: Read and Mixed

Finally, we compare IONIA's performance to that of an unreplicated SplinterDB server under read-only and mixed workloads. As shown in Figure 15, for read-only workloads, IONIA offers 2.8× higher throughput by scaling reads, and for mixed workloads IONIA is 1.22× to 1.45× faster.

5.9 Performance under Failures

In the experiments so far, we demonstrated IONIA's performance without failures. We now show that IONIA's performance and availability are not impacted as long as a supermajority is available. Figure 16 shows the throughput over time for a write-only workload with five replicas. Initially, all replicas are up and IONIA achieves high throughput. After a while, one of the replicas fails; however, since four replicas (i.e., a supermajority) are available, IONIA continues to commit requests in 1RTT, thereby maintaining high performance. In contrast, existing approaches (such as CRAQ [70] and Hermes [39]) that offer 1RTT and scalable reads would suffer

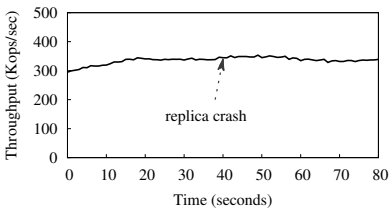


Figure 16: **IONIA Performance Under Failures.**

from write unavailability even when one replica fails.

6 Discussion

Beyond KV stores. In this paper, we focus on designing efficient replication for SSD-based KV stores. However, our ideas apply to other storage systems such as file systems and databases that are built atop write-optimized structures as well. A requirement of our read protocol is that it must be possible to identify which piece of data is read by a request, which holds in many existing systems. Further, while updates in WO-KV stores do not return execution results, other storage systems might support updates that return execution results; however, such updates can be readily supported by IONIA. In particular, when a client issues an update that returns an execution result, IONIA can use the fallback path (described in §4.3) to commit such updates in 2RTT.

Performance with bigger clusters. With larger cluster sizes, a concern might be that meta-queries can become the bottleneck before SSD IOPS. However, this is not a concern in practice today. Most practical systems use only a handful of replicas (3, 5, or 7) [34] and as shown in §5.4, IONIA scales well for such practical cluster sizes.

Performance with slower networks. In most storage systems, SSD IOPS becomes bottlenecked prior to network/NIC. We expect this trend to hold: as SSD IOPS increases, so would the network/NIC packet rate and CPU core count. For example, even when each replica can serve tens of millions of reads/second (via many SSDs), latest NICs (Connect-x6) can serve 215M messages/s [51]. Thus, meta-queries would not be the bottleneck in common scenarios. However, in the (uncommon) deployment scenario where the network/NIC is the bottleneck, as an optimization, many meta-queries can be batched on the client side to amortize CPU/NIC overheads. We leave this optimization as an avenue for future work.

7 Related Work

Other High-Throughput Protocols. Apart from the approaches discussed in §2, a few prior approaches enable multi-core replication through deterministic execution [22, 32]. Other prior systems first execute writes on multiple threads and then replicate the resultant state [71], avoiding non-determinism. Such early execution is a mismatch for WO-KV stores that defer execution for performance. Also, unlike these approaches, IONIA hides ordering and execution latency.

Other Benefits over Low-Latency Protocols. As discussed in §2, IONIA offers 1RTT writes like prior low-latency protocols while offering much higher throughput. IONIA *guarantees* 1RTT writes. Speculative [41, 61] and commutative [44, 53, 59] protocols, in contrast, incur additional RTTs when speculations fail or when writes do not commute. Skyros [28] is a recent protocol that guarantees 1RTT for blind writes by similarly deferring execution. However, Skyros (and the above prior protocols) do not improve throughput or read scaling while reducing latency.

Reading at Non-Leader Replicas. Apart from the protocols discussed in §2, a few prior systems allow reads from a quorum of followers [3, 16]. Shared registers [4] also allows reads at a quorum. While these approaches avoid reads at the leader, they cannot linearly scale read throughput as reads must contact a quorum. Applications over shared logs [5–7] can scale reads by checking the current tail of the log. However, in these approaches, both reads and writes incur high latency, unlike IONIA. Quorum leases [54] maintains per-object leases to scale reads. However, it suffers from high write latency. Also, it is impractical to maintain per-object leases for large disk-based stores. Finally, recent approaches can scale reads through in-network conflict checking [69, 76] or specialized transport protocol [40]. However, these approaches require specialized network hardware (e.g., programmable switches).

Leaderless Approaches. In in-memory replicated systems, message processing overhead at the leader is often the bottleneck [53, 61]. To address this, prior work has built leaderless protocols [49, 53] where any replica can process writes. In a disk-based replicated storage system, however, the leader's storage throughput is the bottleneck. IONIA improves this throughput by batching writes in the background and also taking off read requests from the leader.

LSMs in Distributed Systems. Many prior efforts have optimized LSMs in distributed systems. However, their goals are different from ours, aiming to optimize compactions [29, 72], storage management [75], and load balance [10].

8 Conclusion

We present IONIA, a new replication protocol suited for modern WO-KV stores. IONIA exploits the unique characteristics of WO-KV stores to achieve high performance. We experimentally show that IONIA offers high-throughput, 1RTT writes, and scalable, 1RTT reads. Given that WO-KV stores are widely used, IONIA offers a way to make these stores fault-tolerant with almost no overhead and scale reads.

Acknowledgments

We thank Patrick P. C. Lee (our shepherd) and the anonymous FAST '24 reviewers for their insightful comments and suggestions. We thank the members of DASSL for their discussions. We also thank CloudLab [63] for providing a great environment to run our experiments.

References

[1] Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. Early Scheduling in Parallel State Machine Replication. In *Proceedings of the ACM Symposium on Cloud Computing*, 2018.

[2] Apache. Cassandra. http://cassandra.apache.org/.

[3] Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Yanan Zhi, and Jianfeng Zhu. Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '17)*, Santa Clara, CA, July 2017.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-passing Systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[5] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*, Banff, Canada, November 2020.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[7] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, October 2013.

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, July 2019.

[9] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. An Introduction to Be-trees and Write-optimization. *USENIX ;login:*, 40(5):22–28, 2015.

[10] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Lausanne, Switzerland, March 2020.

[11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[12] Gerth Stølting Brodal and Rolf Fagerberg. Lower Bounds for External Memory Dictionaries. In *SODA*, volume 3, 2003.

[13] Aldenio Burgos, Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. Exploiting Concurrency in Sharded Parallel State Machine Replication. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[14] Tushar D Chandra, Vassos Hadzilacos, and Sam Toueg. An Algorithm for Replicated Objects with Efficient Reads. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing (PODC '16)*, Chicago, IL, July 2016.

[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[16] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Linearizable Quorum Reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '19)*, Renton, WA, July 2019.

[17] CockroachDB. Pebble. https://github.com/cockroachdb/pebble.

[18] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Online, July 2020.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[20] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, October 2012.

[21] James Cowling and Barbara Liskov. Granola: Low-overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[22] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos Made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 105–120, Monterey, California, October 2015.

[23] Effi Ofer, Danny Harnik, and Ronen Kat. Object Storage Traces: A Treasure Trove of Information for Optimizing Cloud Workloads. https://www.ibm.com/cloud/blog/object-storage-traces.

[24] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. Boosting Concurrency in Parallel State Machine Replication. In *Proceedings of the 20th International Middleware Conference*, 2019.

[25] Facebook. Merge Operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator.

[26] Facebook. RocksDB. http://rocksdb.org/.

[27] Pedro Fouto, Nuno Preguiça, and João Leitão. High Throughput Replication with Integrated Membership Management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022.

[28] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual, October 2021.

[29] Panagiotis Garefalakis, Panagiotis Papadopoulos, and Kostas Magoutis. ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 211–220, 2014.

[30] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*, Online, April 2021.

[31] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[32] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the Speed of Multi-core. In *Proceedings of the EuroSys Conference (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[33] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.

[34] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[35] Intel. Intel Optane 905P. https://www.intel.com/content/www/us/en/products/sku/129833/intel-optane-ssd-905p-series-1-5tb-12-height-pcie-x4-20nm-3d-xpoint/specifications.html.

[36] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, February 2019.

[37] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All About Eve: Execute-verify Replication for Multi-core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, October 2012.

[38] Karthik Ranganathan. Low Latency Reads in Geo-Distributed SQL with Raft Leader Leases. https://blog.yugabyte.com/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/.

[39] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A Fast, Fault-tolerant and Linearizable Replication Protocol. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Lausanne, Switzerland, March 2020.

[40] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*, Heraklion, Greece, April 2020.

[41] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[42] Ramakrishna Kotla and Michael Dahlin. High Throughput Byzantine Fault Tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '04)*, Florence, Italy, June 2004.

[43] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[44] Leslie Lamport. Generalized Consensus and Paxos. 2005.

[45] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[46] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. 2012.

[47] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, CA, October 1991.

[48] LogCabin. LogCabin. https://github.com/logcabin/logcabin.

[49] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[50] Mark Callaghan. Types of Writes. http://smalldatum.blogspot.com/2014/04/types-of-writes.html.

[51] Mellanox. ConnectX-6 Card. https://support.mellanox.com/s/productdetails/a2v50000000p8ReAAI/connectx6-card.

[52] MongoDB. Read Concern Linearizable. https://docs.mongodb.com/manual/reference/read-concern-linearizable/.

[53] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, October 2013.

[54] Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, November 2014.

[55] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[56] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[57] Oracle. Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[58] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[59] Seo Jin Park and John Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, February 2019.

[60] Percona. Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[61] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015.

[62] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building Key-value Stores using Fragmented Log-structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[63] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[64] Robert Escriva. HyperLevelDB. https://github.com/rescrv/HyperLevelDB.

[65] Sarang Masti. How we built a general purpose key value store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-data/zippydb/.

[66] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[67] SpeeDB. RocksDB Basics. https://docs.speedb.io/rocksdb-basics#key-tombstones-delete.

[68] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, July 2019.

[69] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[70] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, CA, June 2009.

[71] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

[72] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: Index Shipping for Efficient Replication in LSM Key-Value Stores. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys '22)*, Rennes, France, April 2022.

[73] VMware. SplinterDB. https://splinterdb.org/.

[74] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[75] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, and Si Wu. DEPART: Replica Decoupling for Distributed Key-Value Storage. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, Santa Clara, CA, February 2022.

[76] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. 13(3):376–389, nov 2019.