Optimizing Longhorn for High Performance Cloud-native Storage

Konstantinos Kampadais
Institute of Computer Science
FORTH
Heraklion, Greece
kampadais@ics.forth.gr

Antony Chazapis
Institute of Computer Science
FORTH
Heraklion, Greece
chazapis@ics.forth.gr

Angelos Bilas*
Institute of Computer Science
FORTH
Heraklion, Greece
bilas@ics.forth.gr

Abstract

Longhorn is a popular open-source, cloud-native software-defined storage (SDS) engine that provides distributed block storage in Kubernetes, emphasizing on simplicity and ease of deployment. However, running Longhorn on-premises has revealed that its data path limits the ability to exploit modern high-performance hardware such as NVMe solid-state drives and low-latency, high-bandwidth networks. This paper presents a set of architectural and implementation-level optimizations to Longhorn's storage engine that substantially improve I/O capacity while preserving compatibility with the existing design. By integrating ublk at the frontend, restructuring controller–replica communication to remove serialization bottlenecks, and employing DBS, a simplified direct-to-disk storage backend optimized for snapshot-heavy workloads, the system achieves up to an order-of-magnitude improvement in IOPS and bandwidth compared to the upstream version. We further demonstrate comparable—and in several cases superior—performance relative to OpenEBS Mayastor, a high-performance Kubernetes storage engine, despite relying on fewer system dependencies. Our results contribute to enhancing Longhorn's applicability in both cloud and on-premises deployments, as well as providing insights applicable to the design and implementation of future high-performance cloud-native SDS systems.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Software and its engineering → Cloud computing**; • **Information systems → Cloud based storage**; *Storage virtualization.*

Keywords: Longhorn, Software-defined storage, Distributed storage, Kubernetes, Cloud-native

*Also with Computer Science Department, University of Crete.

CHEOPS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2604-0/26/04
https://doi.org/10.1145/3805687.3806255

ACM Reference Format:
Konstantinos Kampadais, Antony Chazapis, and Angelos Bilas. 2026. Optimizing Longhorn for High Performance Cloud-native Storage. In *6th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '26), April 27–30, 2026, Edinburgh, Scotland Uk*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3805687.3806255

1 Introduction

In cloud-native software architectures, the storage setup plays a significant role in determining the scalability and reliability properties of the whole system. Microservice developers requiring persistent storage commonly rely on services provided by the cloud hosting platform or deploy a third-party volume management software. The cloud-native ecosystem is abundant with such solutions that either interface Linux-native or custom-built software-defined storage (SDS) platforms to container orchestration APIs and deployment strategies. Implementing a cloud-native SDS has several benefits over choosing a similar product already available from a cloud provider, as it provides tighter data placement controls (for both efficiency and policy), vendor independence, cost efficiency, as well as access to advanced features that may not be part of standard offerings.

Longhorn [5] is a popular open-source, cloud-native volume manager that implements its own distributed block storage system. It uses a set of services that collectively handle all aspects related to capacity management, performance, and fault tolerance while interfacing with both Kubernetes and the end user. Longhorn is an actively developed and mature project, part of the Cloud Native Computing Foundation (CNCF) software catalog [2]. However, our installations have revealed that it currently lacks the ability to take full advantage of the hardware performance available in servers featuring high-speed solid-state disks and high-bandwidth network connectivity. The bottleneck we observe may not be critical in typical cloud deployments where the environment imposes stringent performance limits. On the other hand, it dominates when high-specification instances are leased at significantly increased costs, or hardware is on-premises.

In this paper, we investigate a series of optimizations that have been engineered in Longhorn's core, called the Longhorn engine [6] (hereafter referenced simply as *engine*), that collectively allow the system to achieve an order of

magnitude better IOPS and bandwidth. The engine, which is separately deployed in full for each managed volume, consists of a *controller* (data aggregator) and several *replicas* (data storage endpoints). Our remodeling applies to three points of the engine architecture, which have been found to be the most effective for boosting the overall performance: (i) the connectivity between the controller and the host operating system, where we use the *ublk* framework [15] instead of the current solution based on *iSCSI/tgt* [14], (ii) the communication between the controller and the replicas, where we employ a strategy that minimizes locks between involved threads, and (iii) the data storage scheme used by the replicas, in which we utilize a custom direct-to-disk block management layer, named *Direct Block Store* (DBS), instead of the default sparse file-based implementation. All our changes are available in a public repository [10].

Our optimized engine specifically targets disaggregated storage architectures, where storage and compute tiers are scaled independently over high-speed networks. On the other hand, even in typical setups, a variety of applications can benefit from the increased performance. The system's bandwidth improvements ensure massive data volumes do not stall processing pipelines; an issue faced when checkpointing large ML training workflows or performing high-resolution video analytics. Increased IOPS provide the low-latency responsiveness required by transactional database workloads or microservices with high concurrency.

This paper makes three key contributions: First, we demonstrate that targeted optimizations at key points in the I/O path yield substantial improvements in efficiency. Second, we provide a reference framework for distributed SDS implementations seeking to integrate emerging operating system technologies—particularly ublk—into their frontends. Finally, we introduce DBS, an open-source solution that, while not architecturally novel, offers a practical foundation for addressing analogous challenges in related projects.

In the following sections we briefly describe the engine's architecture, followed by a detailed analysis of our optimizations. The evaluation section compares our version of Longhorn with the upstream version, along with two different setups of Mayastor [8] (part of OpenEBS [9]), an existing high-performance storage engine for Kubernetes.

2 Related Work

Cloud-native environments require on-demand, reliable storage, compatible with the Container Storage Interface (CSI) [4]. Available volume managers either implement their own SDS stack or interface with existing systems, usually distributing data across nodes for durability. Longhorn is a favorable CSI solution due to its ease of deployment and use that implements its own independent engine. Rook [12] is based on Ceph [21], while OpenEBS is typically deployed with an LVM backend, but can also use other storage implementations (such as Jiva, cStor, and Mayastor). Piraeus [11] (based on DRBD [3]) and Carina [1] (based on LVM), also provide distributed or node-local storage within the CNCF ecosystem. Very few cloud-native storage systems prioritize high-performance scalability across multiple nodes; among these, Mayastor stands out as the most prominent. In this paper, we use Mayastor as a baseline for our evaluation.

Custom-built SDS stacks require an interface to expose internal blocks at the OS level. Longhorn and OpenEBS (Jiva and cStor storage engines) use iSCSI frontends, which, while flexible, introduce significant overhead. Modern frameworks like NVMe-oF (via SPDK [13]) and ublk offer superior efficiency [18–20]. To update existing systems, SPDK may require complete refactoring, while ublk, leveraging Linux's io_uring for high-performance, asynchronous I/O, provides a simpler path. Although Longhorn's "v2" engine explores NVMe-oF (also used by Mayastor), this work focuses on a ublk-based solution that is easier to integrate and has been reported to match and even surpass NVMe-oF performance in early tests of similar setups [7]. Longhorn v2 is a complete rewrite of the engine and is still in beta. We are not aware of any other cloud-native storage system employing ublk.

3 Longhorn Architecture

Longhorn components form a small web of microservices that provide distributed block storage in Kubernetes (Fig. 1). At the core is the Longhorn engine, responsible for managing data replication and ensuring high availability across multiple replicas. Each Longhorn engine instance—a controller with associated replicas—implements a single volume. The Longhorn manager acts as the control plane, orchestrating the lifecycle of storage volumes, snapshots, and backups while interfacing with the Kubernetes API via the Longhorn CSI plugin. Additionally, the Longhorn UI provides a user-friendly dashboard for managing and monitoring storage operations. This work focuses primarily on the engine service, which is the only component critical to performance since it implements the actual I/O path.

Longhorn developers refer to the engine as the "world's smallest storage controller." Indeed, its design is simple and lightweight, consisting of three basic components (Fig. 2, left side): (i) the *frontend*, which is responsible for interfacing with the OS, so the controller's block API can be accessed seamlessly by applications over a virtual block device, (ii) the *controller*, which routes I/Os to the replicas, functioning as a simple RAID controller (although only supporting mirroring), and (iii) the *replica*, which stores the volumes' blocks in an actual device (the current implementation uses sparse files for storage). In the distributed environment of Kubernetes, all these components are deployed in containers and communicate over the network. The frontend and controller are grouped together and run on the same node that requests the

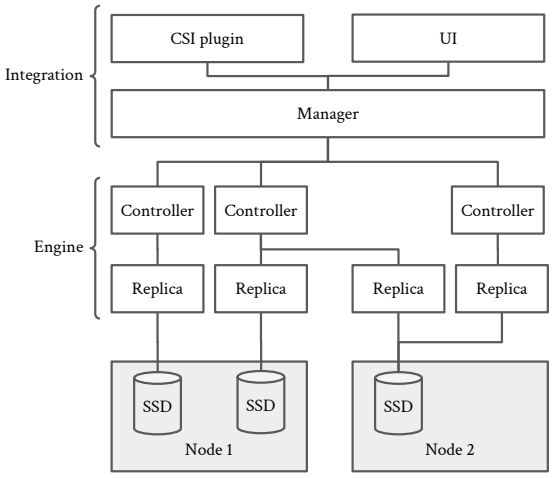


Figure 1. Longhorn components.

device/volume, while the replicas typically run on different nodes (one replica can be co-located with the controller).

The frontend creates a block device using the kernel iSCSI driver, by leveraging tgt, a third-party project that implements iSCSI targets in userspace. Tgt has a plugin architecture that allows developers to materialize I/O operations using custom methods. To connect tgt with the engine, the Longhorn tgt plugin uses a Unix socket to serve each I/O request from the controller (the controller acts as the server, the tgt plugin as the client). Thus, each read and write issued to the exposed virtual iSCSI block device is passed on to tgt, which, in turn, uses the Longhorn plugin to forward it to the controller over a Unix socket. Tgt is packaged along with the controller, in the same container. The controller includes a Go wrapper that executes CLI commands on volume startup, which start tgt after the Unix socket from the controller's side is ready to accept connections.

The controller's basic responsibility is to proxy I/O to replicas and monitor for failures. It also employs a secondary out-of-band communication mechanism to receive management commands from the Longhorn manager (*i.e.*, volume start/stop, snapshot, backup). The layer of the controller that communicates with the replicas is internally called the backend. With no erasure coding or deduplication support, each write is simply copied to all replicas, and each read is served by one replica in round-robin fashion. Note that writes create multiple backend requests that must all be replied before the command completes and the final response is sent back through the frontend.

The replica is the final component of the engine, responsible for storing data in a physical medium. Replicas consist of two basic layers: replica-to-controller communication and the storage mechanism. Communication is implemented similarly to the controller's Unix socket server (albeit over TCP); the replica acts as the server, using multiple threads to serve

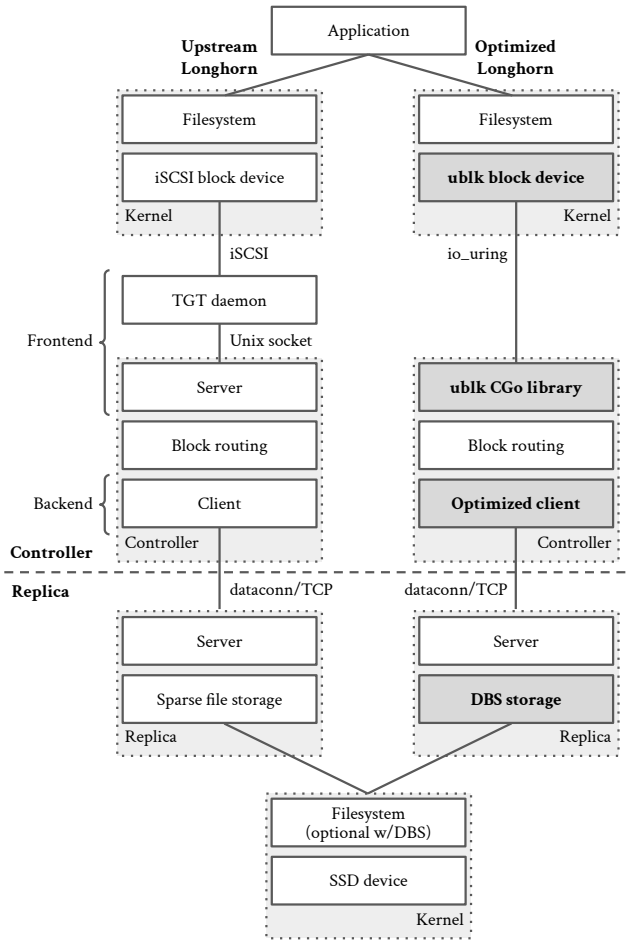


Figure 2. Side-by-side comparison of the upstream and optimized architecture of the Longhorn engine. Modified components are marked with a dark background shade.

TCP connections and handle requests. The protocol used for exchanging commands and results is custom to Longhorn and internally called "dataconn."

The replica backing store receives read and write block requests that should be persisted to a physical medium. The default Longhorn implementation uses Linux sparse files for data storage, which effectively delegates space allocation to the filesystem and abstracts the underlying physical device, allowing it to write to diverse backends such as block devices, local storage, or cloud volumes. Sparse files efficiently manage storage by only allocating space for blocks that have been written to. This reduces storage overhead for volumes with significant unused capacity, which is particularly useful in cloud environments where applications may allocate large volumes but only use a fraction of the space. Snapshots are supported by creating a new sparse file for each snapshot to record changes. The latest snapshot and "version" of each replica is kept in a separate metadata file, in order to verify

consistency. In the case of a faulty replica, the controller is responsible for identifying it and rebuilding it using data from the most up-to-date copy.

4 Implementation

This section details the implementation of the major changes introduced to upstream Longhorn, specifically addressing the OS block device interface, inter-node communication, and backend storage (Fig. 2, right side).

4.1 Frontend

Upstream Longhorn uses iSCSI via the tgt framework to expose a volume to the OS as a local block device, which adds performance overhead and limits scalability. We surveyed alternatives and identified ublk as the best solution. ublk is a modern Linux framework that utilizes io_uring [17] to implement block devices in user space, significantly reducing the overhead compared to older technologies like iSCSI, NBD, or FUSE, as well as the complexity traditionally associated with kernel-level drivers. It allows storage engines to handle I/O requests with high efficiency and parallelism, boosting performance by reducing latency and memory copies.

The current ublk framework includes a userspace daemon, ublksrv [16] that allows users to implement a custom driver to handle each I/O request (similar to tgt for iSCSI). A working proof-of-concept driver already existed for Longhorn, and we started by extending this implementation. The PoC driver used the same UNIX socket and protocol as tgt to communicate with the controller. Experimental evaluation showed a significant performance improvement; however, the use of a UNIX socket for communication introduced a large number of context switches. As a result, our next step was to integrate ublk more tightly with the rest of the controller in order to reduce overheads as much as possible.

Since Longhorn is written in Go and the ublk framework in C/C++, the integration could either be achieved by rewriting the entire framework in Go or by using CGo to bridge C and Go. We embedded the core functions of ublksrv directly within the controller using CGo, which proved challenging because of the heavy use of C++ coroutines in the I/O data path. Ublksrv also provides an asynchronous interface fully realized in C++ that is essential for optimal throughput. To address these limitations, we replaced the C++ functions with C equivalents, mirroring the implementation using pthreads and POSIX semaphores.

The resulting C-only codebase could easily be integrated with the rest of the controller. The device admin commands now come from Go arguments passed in a C function that communicates with the ublk_drv kernel module. The I/O data path connectivity is bridged using a Go callback to serve each I/O operation in a separate Go thread, while C threads use semaphores to wait for the results. The Go wrapper of ublksrv's basic functionality can be used autonomously to interface any other storage system to ublk.

4.2 Controller-Replica Communication

The Longhorn controller's primary responsibility is to propagate I/O requests to replicas via function calls. We discovered that the baseline implementation exhibits excessive buffer allocation and redundant data copying along the I/O path, inflating CPU utilization and latency. To address these inefficiencies, we implemented a minimal-copy pipeline that enforces a strict allocation discipline using Go slices and channels. By preallocating a fixed pool of message structures and recycling them via bounded channels, we eliminate per-request allocations on the hot path. Furthermore, read and write buffers were decoupled: write requests carry kernel-provided buffers consumed in place, while read buffers are drawn from a reusable pool, effectively reducing heap churn.

The most significant bottleneck was identified in the default communication logic, which relies on a single thread running a loop function to coordinate all I/O. This thread handles all requests and responses using a single Go map for message id tracking, which imposes sequential processing to avoid race conditions. While sufficient for the original iSCSI frontend, this architecture fails to scale when subjected to the high-volume parallel I/O issued by the optimized ublk frontend. The high rate of incoming requests overwhelms the single-threaded loop, creating a synchronization bottleneck that limits the overall throughput of the engine.

To overcome the scalability limit, we replaced the centralized Go map and loop function with a parallelized token-based system. We introduced a fixed-size message array, sized to the maximum number of in-flight I/O operations, and a companion Go channel populated with array indices acting as unique request tokens. This design allows issuing threads to independently acquire a token, store request data at a unique index in the array, and forward the message without central coordination. Similarly, The receive path uses the response's ID to directly index the array and mark the request as completed.

The overhauled data path leverages Go's channel mechanics to guarantee thread-safe token distribution, ensuring that only one thread manipulates any given index in the messages array at any time. By eliminating the lock-contention and sequential processing of the original map-based implementation, the controller can now sustain the high IOPS throughput provided by the ublk frontend. Additionally, we made the number of concurrent connections to each replica configurable, allowing the system to be tuned for maximum performance across different hardware environments.

4.3 Block Storage

The original storage scheme used in replicas is based on sparse files. While sparse files do not strictly require a specific filesystem, their performance and efficiency may significantly vary depending on the filesystem's architecture. Furthermore, each replica maintains a separate metadata file per volume, imposing a management overhead, especially for writes; disabling write versioning increases write performance almost to the level of reads. In addition, performance degrades severely with snapshots, as each new snapshot creates a new sparse file, and reads may potentially traverse the entire snapshot chain to locate a block.

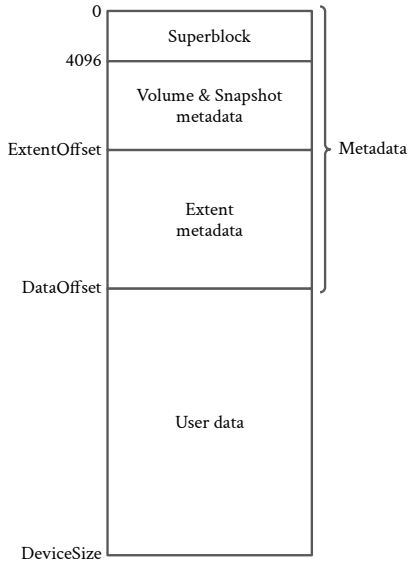


Figure 3. Internal structure of storage managed by DBS.

To address these issues, we implemented Direct Block Store (DBS), a lightweight, direct-to-disk storage framework. DBS manages volumes and snapshots directly on physical devices or files, exposing an extensive API and CLI for querying and manipulating metadata even outside Longhorn's context. Logically, DBS divides storage into 1 MB extents (extent size practically affects the granularity of allocations; any extent size can be used, as long as it is a power of 2). Volume operations use in-memory metadata and can proceed in parallel, while only writes to unallocated space require serialization to update the superblock mark. DBS extensively utilizes bitmaps for efficient metadata operations.

Snapshot management in DBS is handled through a hierarchical extent mapping system. Instead of traversing file chains, DBS maintains an in-memory map of extents for each snapshot, allowing for $O(1)$ block lookups. When a new snapshot is created, DBS performs a "copy-on-write" allocation only when a block is modified, effectively minimizing the storage footprint while maintaining high read performance across deep snapshot trees. This approach significantly reduces the I/O amplification typically associated with Longhorn's default sparse file mechanism.

Beyond the integration of DBS within the replica, we also revamped the replica's data path following the same principles applied to the controller. We introduced reusable buffers and preallocated message arrays, carefully propagating each request to DBS to enhance system performance. Written in Go (approx. 1000 lines), DBS provides a stand-alone, high-performance block storage solution with point-in-time snapshots and can be easily incorporated into other SDS stacks.

4.4 Longhorn Integration

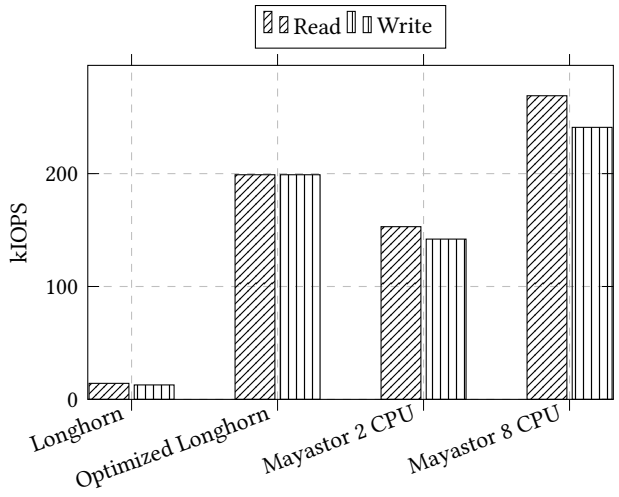
The engine optimizations have additionally been integrated as additional features into the broader Longhorn ecosystem, enabling users to configure optimizations and tuning parameters through Kubernetes StorageClass manifests. The process involved extending existing API schemas and internal communication interfaces to ensure that configuration values could be validated, transmitted, and applied consistently across all system components. Configurable parameters include the number of frontend queues, the queue depth for each queue, and the number of TCP connections between the controller and each replica. By exposing these settings, we provide flexibility for users to experiment and identify optimal values based on their specific hardware.

5 Evaluation

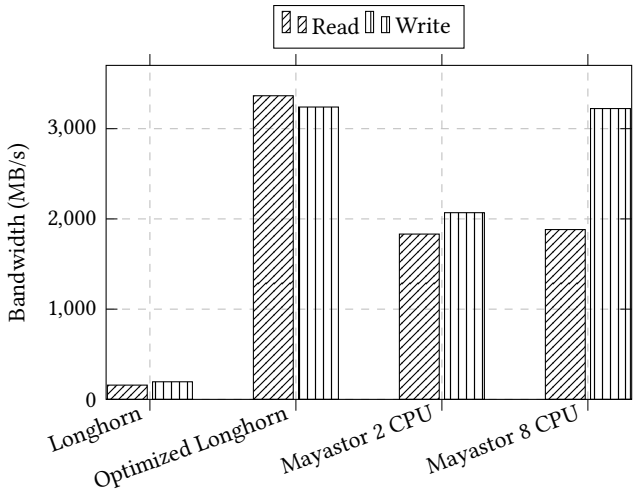
5.1 Setup

To evaluate optimized Longhorn, we deploy it in a local Kubernetes cluster of three identical servers (AMD EPYC 7551P, 256 GB RAM, Samsung NVMe 990 Pro 4 TB) interconnected via 56 Gb/s Infiniband. For all tests we use standard TCP connectivity between nodes. We compare our system against mainline Longhorn v1, default Mayastor (using 2 exclusive cores), and tuned Mayastor (using 8 exclusive cores). Preliminary runs of Longhorn v2 suggest a level of performance similar to tuned Mayastor; Longhorn v2 is still in beta and we failed to deploy equivalent test configurations with the other systems. For experiments, we run *fio* with direct I/O for measuring IOPS (4k random reads/writes) and bandwidth (1 MB sequential reads/writes). We have also used fio to verify the integrity of data (with CRC32C checksums).

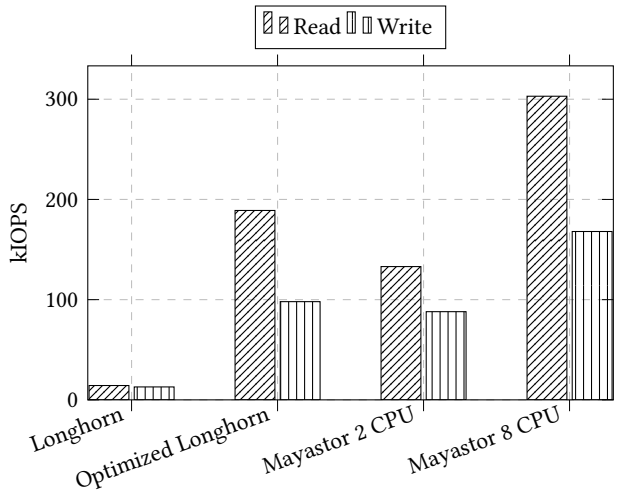
Beyond the local cluster, we have verified the performance on AWS using c5d.2xlarge instances, where the ublk version quickly maxed out the IOPS capacity (40k), validating reported measurements from cloud nodes [22]. The analysis presented here focuses on local cluster nodes to fully observe architectural improvements; higher performance on AWS requires provisioning significantly more expensive node and storage offerings which are again capped to larger limits.



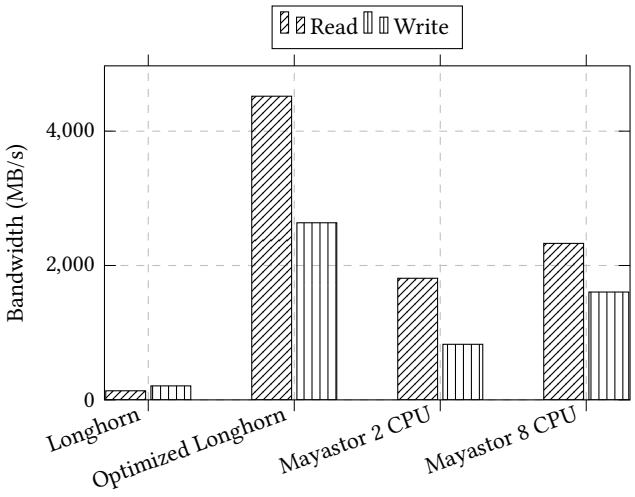
(a) IOPS (1 replica)



(b) Bandwidth (1 replica)



(c) IOPS (3 replicas)



(d) Bandwidth (3 replicas)

Figure 4. Evaluation results across different configurations.

5.2 Frontend Results

Since the most significant optimization has been in the frontend, we provide a breakdown of performance for each implementation. Table 1 illustrates the impact of ublk on overall system performance. Using the initial ublksrv driver without tuning, we observe a 5x improvement over the baseline, which goes up to 25x with multiple queues. In the next iteration, by eliminating the UNIX socket, we reach up to a 35x gain. The ublksrv driver already outperforms iSCSI, however moving ublk handling within the controller reduces context switches and CPU overhead.

Table 1. Frontend evaluation.

	IOPS			
	1 queue		6 queues	
	Read	Write	Read	Write
Longhorn (iSCSI)	20k	20k	-	-
Ublksrv Longhorn	100k	100k	515k	500k
Ublk Longhorn	200k	200k	700k	586k

5.3 Full System Results and Discussion

Using OpenEBS LocalPV as a reference point for direct-to-disk performance (850k IOPS, 3.1 GB/s write bandwidth), we first test with a single local replica (Fig. 4a, 4b) comparing the optimized version with vanilla Longhorn and OpenEBS Mayastor. We have tuned Mayastor to the level of best performance on our hardware. While Mayastor then excels in IOPS, optimized Longhorn reaches 3.3 GB/s read bandwidth, almost double than Mayastor. With one local and

two remote replicas (Fig. 4c, 4d), Mayastor's NVMe-oF core handles replicated write operations more efficiently, but optimized Longhorn again maintains a clear lead in bandwidth matching the local baseline. During the evaluation, we noticed ublk-based Longhorn required roughly 20% per replica plus 20% for the controller. On the other hand, SPDK-based engines (Mayastor and Longhorn v2) that need dedicated CPUs, consume approximately 12% per replica with minimal controller (initiator) utilization.

The "fixed-core" model delivers high efficiency for replicated I/O but requires strict resource preallocation. Optimized Longhorn employs an "on-demand" CPU model that scales with load, making it more suitable for dynamic environments. This flexibility is also important when both applications and storage share the network. When faster paths are available, the system will automatically shift the bottleneck to the available CPUs, whereas when the network is busy (or generally slow, as in typical cloud setups), the performance will adapt to the limits of the available communication throughput and latency. Thus, we argue that optimized Longhorn, although currently outperformed by Mayastor in high-concurrency write IOPS, can provide a balanced, high-performance solution for cloud-native storage. We are currently working on using io_uring for controller-replica communication, which we expect to further boost performance and minimize CPU utilization.

Regarding DBS, when comparing it in isolation with the default sparse files backend we get similar raw performance (as long as metadata versioning is turned off), However, DBS provides better snapshot management and greater architectural extensibility for future features.

6 Conclusion

This paper demonstrates how to overcome the limits of Longhorn's engine in high-performance hardware environments while maintaining architectural compatibility. By replacing the iSCSI-based frontend with ublk, while embedding io_uring handling within the controller, optimizing controller-replica communication, and integrating Direct Block Store (DBS), the engine achieves up to an order-of-magnitude better IOPS and bandwidth. These enhancements have been submitted as pull requests to the upstream Longhorn repository. Future work focuses on improving the controller's communication protocol—potentially using io_uring for network operations as well—and adding caching and layered storage features to DBS to further enhance versatility.

Acknowledgments

The authors thankfully acknowledge the support of the European Commission under the Horizon Europe Programme through project DaFab (GA-101128693), as well as the European Commission and the Greek General Secretariat for Research and Innovation through project REBECCA (GA-101097224); REBECCA is a Chips JU project.

References

[1] [n. d.]. *Carina: A high performance and ops-free local storage for Kubernetes.* https://github.com/carina-io/carina
[2] [n. d.]. *Cloud Native Computing Foundation.* https://www.cncf.io
[3] [n. d.]. *Distributed Replicated Storage System.* https://linbit.com/drbd/
[4] [n. d.]. *Kubernetes Container Storage Interface (CSI) Documentation.* https://kubernetes-csi.github.io/docs/
[5] [n. d.]. *Longhorn: Cloud native distributed block storage for Kubernetes.* https://longhorn.io
[6] [n. d.]. *Longhorn Engine: World's smallest storage controller.* https://github.com/longhorn/longhorn-engine
[7] [n. d.]. *Longhorn Performance Investigation.* https://github.com/longhorn/longhorn/wiki/Longhorn-Performance-Investigation
[8] [n. d.]. *Mayastor: Cloud-native declarative data plane written in Rust.* https://github.com/openebs/mayastor
[9] [n. d.]. *OpenEBS: Kubernetes Storage Simplified.* https://openebs.io
[10] [n. d.]. *Optimized Longhorn Repository.* https://github.com/CARV-ICS-FORTH/longhorn-engine
[11] [n. d.]. *Piraeus: An easy-using cloud native datastore for Kubernetes.* https://piraeus.io
[12] [n. d.]. *Rook: Open-Source, Cloud-Native Storage for Kubernetes.* https://rook.io
[13] [n. d.]. *SPDK: Storage Performance Development Kit.* https://spdk.io
[14] [n. d.]. *Tgt: User-space iSCSI target daemon.* https://github.com/fujita/tgt
[15] [n. d.]. *Ublk: Userspace block device driver.* https://docs.kernel.org/block/ublk.html
[16] [n. d.]. *ublksrv.* https://github.com/ublk-org/ublksrv
[17] [n. d.]. *What is io_uring?* https://unixism.net/loti/what_is_io_uring.html
[18] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. doi:10.1145/3534056.3534945
[19] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. 2018. Performance Characterization of NVMe-over-Fabrics Storage Disaggregation. *ACM Transactions on Storage* 14 (12 2018), 1–18. doi:10.1145/3239563
[20] Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (Rome, Italy) *(CHEOPS '23)*. Association for Computing Machinery, New York, NY, USA, 35–45. doi:10.1145/3578353.3589545
[21] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 307–320.
[22] Sheng Yang. 2020. *Performance and Scalability Report for Longhorn v1.0.* https://longhorn.io/blog/performance-scalability-report-aug-2020/