CODA: Breaking CPU Bottlenecks for Scalable Ceph with DPU-Based OSD Decoupling

Kyuli Park*, Sungmin Yoon*, Awais Khan†, Sungyong Park*‡, Youngjae Kim*‡
*Sogang University, Seoul, Republic of Korea, †Oak Ridge National Laboratory, Oak Ridge, TN, USA
{kyuripark, zserty12, parksy, youkim}@sogang.ac.kr, khana@ornl.gov

***Abstract*—Ceph, a widely used distributed storage system, underutilizes NVMe SSD bandwidth due to CPU bottlenecks, even in high-performance environments. This inefficiency is amplified in resource-constrained settings such as edge clouds and small-scale clusters. Prior work offloads the transport layer (e.g., TCP/IP) to DPUs, but fails to address the dominant overheads arising from OSD-level logic, including message handling and replication coordination. We present CODA, a Ceph OSD Decoupling Architecture that separates the OSD stack and offloads CPU-intensive front-end logic to the DPU. It runs Messenger and OSD core on the DPU while keeping BlueStore on the host, enabling a disaggregated OSD across heterogeneous devices. We further design: (i) *Flow-Direct* for split execution, (ii) *Flow-Adapt* for batching and pipelining under DMA constraints, and (iii) *Flow-Sync* for low-overhead synchronization under OSD scaling. We implement CODA on a Ceph cluster with NVIDIA BlueField-3 DPUs. CODA reduces host CPU usage by up to 88% and improves throughput by up to 49% without replication and 94% with 3-way replication.**

***Index Terms*—Ceph, Data Processing Unit (DPU), Distributed Storage Systems, Resource-Constrained Environments, CPU Offloading**

I. Introduction

As cloud infrastructure expands beyond large-scale data centers to edge clouds and small-scale clusters, there is a growing demand for operating distributed storage on resource-constrained hardware [1], [2]. In such environments, a limited number of CPU cores and constrained memory must serve multiple storage workloads, making the CPU efficiency of the storage software stack a decisive factor in overall system performance. The emergence of NVMe SSDs, which deliver multi-GB/s bandwidth per device, has shifted the performance bottleneck from the storage devices themselves to the software stack that manages them [3], [4]. In response, techniques such as polling-based I/O, kernel bypass, and user-level drivers have been explored to better utilize CPUs [4]–[7]; however, these approaches assume abundant CPU resources and offer limited benefit in resource-constrained settings.

This CPU-centric bottleneck is further exacerbated in distributed storage environments. As NVMe SSD capacities grow from a few to tens of terabytes, high-density deployments have become commonplace, in which a single device is partitioned into multiple storage daemons or multiple NVMe drives are attached to a single host [8]. Distributed storage systems must continuously exchange network messages and perform cluster coordination for data replication, consistency maintenance, and failure recovery, incurring significant CPU overhead beyond storage I/O (§II-A).

Motivation. Our analysis of Ceph [9], a widely deployed distributed storage system, reveals that approximately 90% of a storage daemon's CPU consumption originates not from disk I/O but from application-level operations above the TCP/IP stack, including message serialization, replication coordination, and transaction processing. The disk I/O backend accounts for only 10% (§III). Consequently, as the number of storage daemons increases, the cumulative cost of replication traffic and control-path processing rapidly saturates the CPU, causing disk underutilization even when NVMe devices have ample bandwidth headroom (§III).

Recent interest in Data Processing Units (DPUs), a class of programmable SmartNICs, offers a promising direction [10]–[12]. A DPU integrates ARM-based CPU cores, an independent operating system, a network interface controller, and hardware accelerators within a single SoC; the latest generation (e.g., NVIDIA BlueField-3/4) features 16 or more ARM cores and tens of gigabytes of on-board DRAM, making it a capable execution platform. Prior work has demonstrated the versatility of DPUs through in-path read processing [13], GPU-direct object storage [14], compression and erasure-coding acceleration [15], [16], and key-value store offloading [17]–[19], as well as kernel-level TCP offloading to reduce network stack overhead [20]. However, as noted above, fundamentally resolving the CPU bottleneck caused by daemon scaling requires offloading not only the transport layer but also the application-level CPU-intensive logic, leveraging the DPU's rich computational resources. *A key challenge is that network processing, replication coordination, and the storage backend form a tightly coupled path within the storage daemon, necessitating a design that selectively separates only the CPU-intensive components while preserving cluster consistency semantics.*

Contributions. Building on these observations, we present CODA, a Ceph OSD decoupling architecture that separates the OSD stack and offloads CPU-intensive front-end logic to the DPU. This allows the host CPU to focus on disk I/O, enabling more storage daemons to scale on the same CPU budget. CODA introduces three key techniques: (i) **Flow-Direct** provides a hybrid communication interface based on `ProxyObjectStore`. It routes bulk data through DOCA

‡Corresponding authors: Youngjae Kim and Sungyong Park.

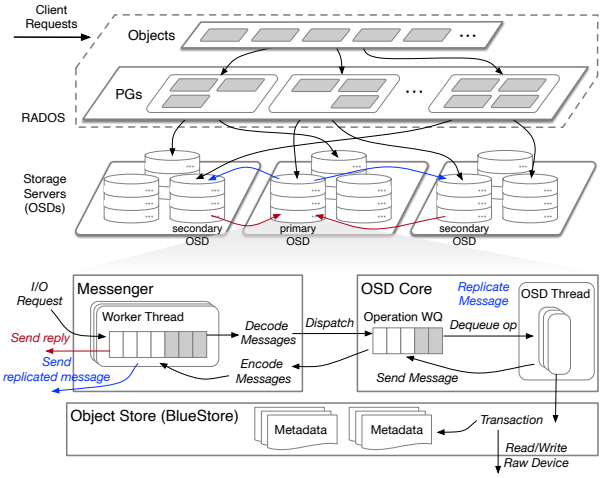


Fig. 1: Ceph architecture and the internal structure of an OSD.

DMA and control messages through a lightweight RPC channel. (ii) **Flow-Adapt** reduces DPU–host transfer overhead. It batches small transactions and applies intra-request pipelining to requests that exceed the 2 MB DMA hardware limit. (iii) **Flow-Sync** supports scalable completion handling and preserves consistency across multiple OSDs. It uses per-channel polling and CAS-based lock-free request–completion matching.

We implement and evaluate CODA on a Ceph cluster equipped with NVIDIA BlueField-3 DPUs. Our results show that CODA reduces host CPU utilization by up to 88% with 4 OSDs. In throughput, CODA improves by up to 49% over an all-BASELINE deployment, and this gap widens to 94% when 3-way replication is enabled. As OSDs scale to 8, the all-CODA configuration achieves 37% higher throughput than all-BASELINE, and a hybrid placement (H4+D4) further improves throughput by up to 71%.

II. Background

A. Object Storage System

Distributed object storage manages data in units of objects, providing scalability and fault tolerance through replication and distributed placement. Among representative systems such as Amazon S3 [21], MinIO [22], OpenStack Swift [23], and Ceph [24], Ceph is the most widely adopted in the open-source ecosystem: it supports block (RBD), file (CephFS), and object (RGW) storage in a single cluster, and serves as the default backend for OpenStack and Kubernetes [25]. We therefore use Ceph as the representative target in this paper.

Internally, Ceph relies on RADOS (Reliable Autonomic Distributed Object Store) [24] to manage cluster-wide object distribution. Each object is assigned to a logical unit called a Placement Group (PG), and the CRUSH algorithm [26] deterministically maps each PG to a set of Object Storage Daemons (OSDs). An OSD daemon is the core component responsible for handling data I/O, and as illustrated in Figure 1, it consists of three main modules.

Messenger handles message transmission and reception through a messenger thread pool (msgr-worker). It inserts client I/O requests into the work queue of the corresponding PG, and also manages communication with other OSDs and Monitors for replication traffic, heartbeats, and recovery coordination. Data serialization, checksumming, and encryption are performed on the CPU via the kernel TCP/IP stack.

OSD Core handles replication and transaction processing through the OSD thread pool (tp_osd_tp). It dequeues requests, forwards replication requests to peer OSDs, and issues transactions to the local object store. The replication coordination process requires more message exchanges as the OSD count increases, making it a primary source of growing CPU overhead.

Object Store is responsible for persisting data. The current default object store in Ceph is BlueStore [27], which directly manages raw devices without an intervening local file system. BlueStore independently manages metadata, executes transactions, and performs reads and writes to the physical device.

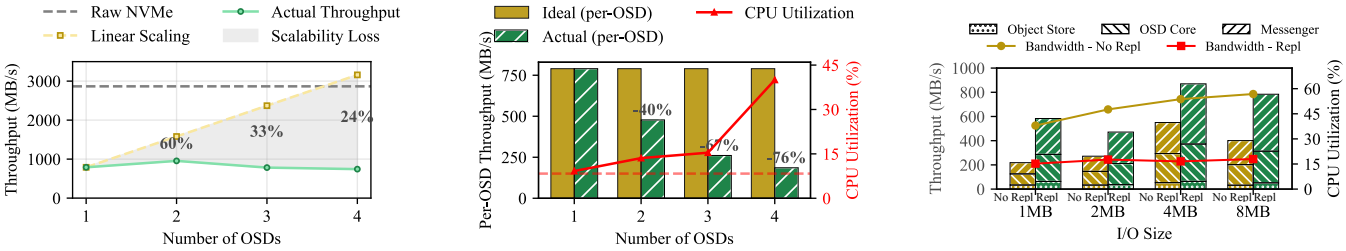
B. CPU Bottlenecks in High-Density Ceph OSD Deployments

Ceph is designed to scale capacity and performance horizontally by adding OSDs. As NVMe SSD capacities have grown to tens of terabytes, high-density Ceph deployments have become commonplace, where a single device is partitioned into multiple OSDs or multiple NVMe drives are attached to one host [8]. The official Ceph documentation recommends at least one physical CPU core per OSD [28], with more required under replication. While such deployments are spreading across large-scale data centers, they are equally prevalent in CPU-constrained environments such as 5G edge nodes [1], Kubernetes-based small-scale clusters [2], [25], and hyperconverged infrastructure at remote sites, where multiple OSDs must operate on a limited CPU budget. In these settings, scaling the OSD count causes Messenger and OSD Core CPU consumption to accumulate, saturating the CPU and leading to disk underutilization despite the available NVMe bandwidth(Section III).

C. Offloading Opportunities with DPUs

A Data Processing Unit (DPU) is a class of processor designed to offload data-centric workloads such as networking and storage from the host CPU. It integrates multiple low-power CPU cores (ARM or RISC-V based), an independent Linux operating system, a network interface controller, and hardware accelerators (for erasure coding, encryption, compression, etc.) within a single SoC. It features on-board DDR memory and built-in flash storage, and communicates with the host system and other PCIe devices via DMA or peer-to-peer transfers over the PCIe interface. Major chip vendors including NVIDIA (BlueField) [10], Intel (IPU) [11], AMD (Pensando) [12], Marvell (Octeon) [29], and Broadcom (Stingray) [30] offer DPU products, while hyperscalers such as AWS (Nitro) [31], Alibaba (CIPU) [32], and Microsoft [33] operate large-scale infrastructure built on proprietary DPUs.

DPU usage models fall into two broad categories. The **on-path** approach places the DPU between the network and the host to process packets in transit (e.g., kernel-level TCP offloading, IPsec, OVS acceleration); it reduces network stack



(a) Aggregate throughput scaling (b) Per-OSD efficiency (c) CPU utilization breakdown

Fig. 2: OSD scalability analysis with 4 MB writes (CPU utilization normalized to 12 host cores). (a) shows the throughput gap between ideal linear scaling and actual measurements, (b) shows per-OSD throughput degradation and CPU utilization, and (c) shows CPU utilization breakdown across OSD modules under different replication settings. In (c), the object store uses BlueStore, the default backend in Ceph. On the x-axis, No Repl denotes the configuration without replication, while Repl denotes 3-way replication.

overhead but cannot replace application-level logic such as replication coordination or transaction processing. The **off-path** approach instead runs part or all of a host application directly on the DPU's general-purpose ARM cores and independent OS, treating the DPU as an execution platform rather than a packet processing engine. The latest generation DPUs (e.g., NVIDIA BlueField-3/4) feature 16 or more ARM cores and tens of gigabytes of on-board DRAM, sufficient to host complex services such as distributed storage daemons.

This paper adopts the off-path approach, directly executing the CPU-intensive modules of the distributed storage daemon on the DPU's SoC to resolve the application-level CPU bottleneck that kernel-level TCP offloading alone cannot address.

III. Preliminary Results: CPU Bottlenecks Limit OSD Scalability

A. Rising CPU Consumption under OSD Scaling

Ideally, performance should scale linearly with the number of OSDs. Whether such scaling is achieved in practice, and what factors limit it, remain open questions. In particular, when OSDs are densely packed within a single host, host resources may become the scaling bottleneck. To investigate this, we increase the OSD count from one to four on a single host with replication disabled and run a 4 MB object write workload using RADOS bench [34]. The goal is to quantitatively characterize the actual scaling behavior and to identify the dominant system bottleneck. The detailed experimental configuration is described in Section VI-A.

Figure 2(a) and (b) show the throughput scaling characteristics as the OSD count increases. In Figure 2(a), Linear Scaling represents the ideal projection based on the single-OSD peak throughput of 790 MB/s, while Raw NVMe indicates the physical performance ceiling of the same device. Actual Ceph throughput falls far short of the ideal: it reaches only 60% of linear scaling at two OSDs, 33% at three, and 24% at four.

To analyze the cause of this scaling gap, we additionally measure per-OSD throughput alongside CPU utilization. Figure 2(b) shows the result. As the OSD count increases, per-OSD throughput drops sharply from 790 MB/s to 186 MB/s, while CPU utilization rises continuously from 14.3% to 39.9%. This result indicates that OSD scaling does not translate into performance gains but instead leads to rapid CPU resource exhaustion. In other words, CPU resources act as the primary bottleneck limiting OSD scalability.

To identify the root cause of the high CPU consumption, we analyze the per-module CPU utilization within the OSD daemon. To isolate the impact of network and replication processing, we measure both the no-replication and 3-way replication configurations. This allows a quantitative assessment of the CPU overhead attributable to replication coordination.

Figure 2(c) shows the CPU utilization distribution for 4 OSDs across I/O sizes ranging from 1 to 8 MB. The analysis reveals that the majority of CPU consumption originates not from actual storage I/O but from network processing and coordination tasks. Without replication, at 4 MB I/O, Messenger and OSD Core consume 18.5% and 17.3% of CPU respectively, while Object Store, responsible for disk I/O, accounts for only 3.8%. That is, approximately 90% of total CPU utilization is spent on non-storage operations.

Enabling replication amplifies this trend. Messenger and OSD Core consume 36.0% and 22.3% respectively, totaling 63.2%, while throughput drops to 229 MB/s, a 69% decrease compared to the no-replication case. This is because replication introduces additional message exchanges and coordination tasks. Furthermore, CPU utilization increases with I/O size, as message serialization and replication coordination costs grow with larger I/O.

B. Application-Level CPU Bottlenecks in Ceph OSD

Two key observations emerge from the above analysis.

- First, CPU consumption in Ceph OSDs is dominated not by disk I/O itself but by request processing and replication coordination performed in the Messenger and OSD Core. The application-level processing path, including message parsing, work dispatching, and replication coordination, accounts for the vast majority of CPU utilization rather than simple data transfer.
- Second, these CPU-intensive operations scale proportionally with the number of OSDs. Consequently, adding OSDs rapidly saturates the CPU, leading to disk underutilization where the potential NVMe performance cannot be realized.

These findings imply that offloading only the network transport path is insufficient to resolve the bottleneck. Even if

packet transmission and kernel-level transport processing are offloaded, the application code executed afterwards in the Messenger, including request parsing, PG work queue dispatching, replication coordination with secondary OSDs, and transaction creation and completion handling, still accounts for the bulk of CPU consumption. In other words, the bottleneck lies not in the data transport path itself but in the OSD's application-level processing path that operates above it.

Moreover, these CPU-intensive operations are functionally separable from the storage backend that persists data to disk. This structural characteristic presents an opportunity to offload this path independently. Therefore, to fundamentally alleviate the CPU bottleneck, it is necessary to go beyond kernel-level transport offloading and restructure the OSD's application-level processing path, including the Messenger, in an off-path manner. The following sections describe the system design and implementation based on this approach.

IV. CODA Architecture and Design Challenges

As shown in Section III, the scale-up efficiency of Ceph is limited primarily by host-side CPU consumption in the OSD front-end path. In particular, a large fraction of CPU time is spent in the Messenger and OSD Core, and this overhead grows with OSD density, reducing the CPU budget available for backend storage processing. Although these components are tightly integrated with BlueStore within a single OSD process, they play distinct roles. Messenger is responsible for network packet processing and message serialization, while OSD Core handles request parsing and replication coordination. Neither component directly performs data persistence. In contrast, BlueStore is the storage backend that persists data through direct access to local NVMe devices, and must therefore remain on the host due to device affinity.

This separation motivates CODA, which decomposes the OSD into a DPU-resident front-end path and a host-resident persistence backend. CODA places the CPU-intensive but storage-device-independent components, namely Messenger and OSD Core, on the DPU, while retaining BlueStore on the host. Because the DPU provides its own ARM cores and network stack, it can execute communication and coordination tasks without consuming host CPU cycles. This design allows the host CPU to concentrate on backend storage processing, thereby reducing the front-end CPU bottleneck and improving OSD scale-up under limited host CPU budgets.

Realizing this architecture requires addressing three key challenges.

- **C1. Scalable DPU-host data transfer.** BlueStore must remain on the host because it directly accesses local NVMe devices. Therefore, transaction data generated by the DPU-side front end must be transferred efficiently to the host. This transfer path becomes more heavily used as the number of OSDs per node increases. It must therefore provide high throughput and low latency without creating a new PCIe bottleneck. CODA addresses this challenge with *Flow-Direct* (§V-B).
- **C2. Transfer overhead under DMA constraints.** CODA uses DMA for high-speed transfer between the DPU and host. However, a hardware constraint limits each DMA transfer to about 2 MB. Each transfer also incurs fixed overheads, including setup, header delivery, and completion acknowledgment. As a result, large requests must be split into multiple DMA operations, while small requests can suffer from high per-transfer overhead. An efficient transfer path must therefore adapt to request size. CODA addresses this challenge with *Flow-Adapt* (§V-C).
- **C3. Scalable synchronization and completion handling.** Splitting the OSD across the DPU and host introduces synchronization overhead that grows with OSD density, as both the number of DPU-host channels and completion events scale with OSD count. The design must therefore avoid synchronization costs that scale linearly with OSDs. In addition, while the original Ceph design assumes that OSD threads wait synchronously for BlueStore completion, the split design must deliver completion notifications efficiently across the DPU-host boundary. CODA addresses this challenge with *Flow-Sync* (§V-D).

V. Design and Implementation

A. System Architecture

Figure 3 shows the overall architecture of CODA. The DPU executes the OSD's front-end path, including client request reception, replication coordination, and transaction generation, while the host performs only local persistence through BlueStore. The two sides are not in a simple packet forwarding relationship; rather, they operate as cooperative execution entities that share a single OSD execution path.

To enable the DPU side to remotely invoke BlueStore on the host, CODA introduces `ProxyObjectStore` on the DPU and `Proxy Interface` on the host. In Ceph, `ObjectStore` is a pluggable backend interface that allows substitution of backends such as BlueStore [27] and FileStore [35]. CODA leverages this modularity by replacing the DPU-side `ObjectStore` with `ProxyObjectStore`, which intercepts all backend calls issued by the OSD, serializes them, and routes them to the host through one of two paths depending on the operation type. As shown in the upper right of Figure 3, `ProxyObjectStore` separates communication into a *Data Plane* that carries bulk I/O over DMA and a *Control Plane* that carries metadata operations over RPC. On the host side (lower right of Figure 3), `Proxy Interface` submits DMA-delivered write transactions to BlueStore and returns an ACK on commit, returns DMA-delivered read data to the DPU, and dispatches RPC metadata requests to the corresponding BlueStore function by parsing `func_id`. `ProxyObjectStore` and `Proxy Interface` thus operate as a symmetric pair that transparently bridges the DPU-host boundary.

The overall request processing flow proceeds as follows. For a write request, ❶ the DPU's Messenger receives the client request and dispatches it to OSD Core, which performs replication coordination and generates a transaction.

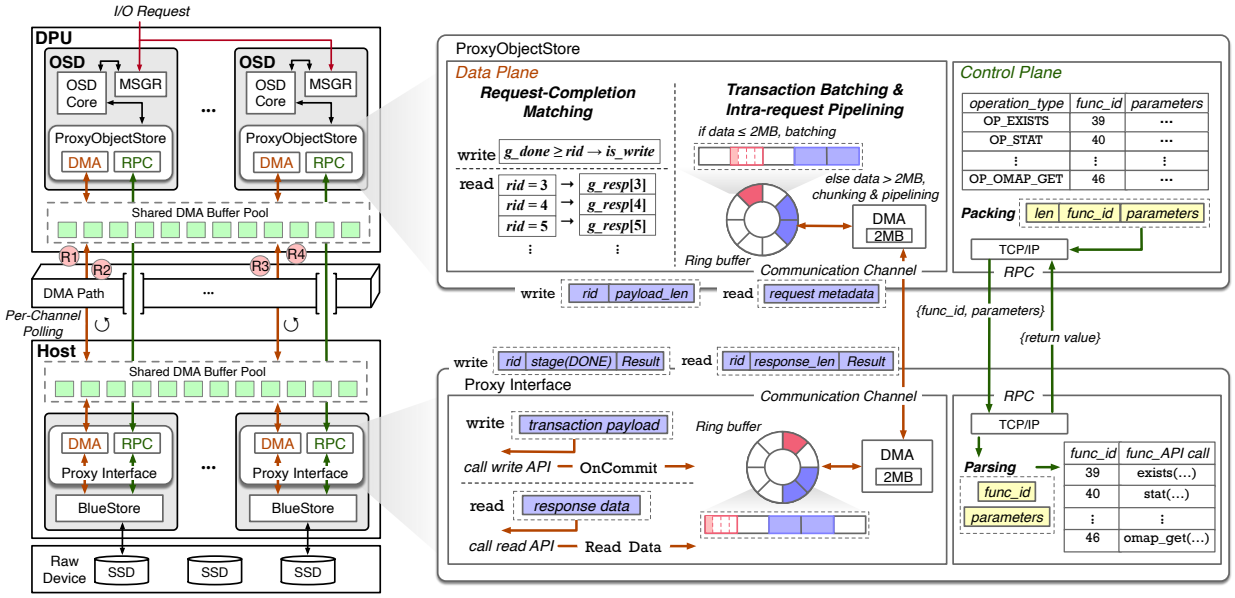


Fig. 3: Overall architecture of CODA.

❷ `ProxyObjectStore` serializes the transaction and transfers it to the host via DMA. ❸ `Proxy Interface` receives it and submits it to BlueStore. ❹ After BlueStore commits, the host sends an ACK. ❺ Upon receiving the ACK, the DPU returns a response to the client. Because the client response is sent only after BlueStore commits, Ceph's consistency and fault tolerance semantics are preserved. Read requests follow a symmetric path. The following sections describe the detailed design of each stage.

B. Flow-Direct: Cross-boundary transaction path

Backend requests issued by OSD threads residing on the DPU are first delivered to `ProxyObjectStore`, which selects the transfer method based on the operation type. `ProxyObjectStore` implements Ceph's standard `ObjectStore` interface, overriding all virtual functions including `queue_transactions`, `read`, `stat`, and `exists`. On each function call, the arguments are serialized into Ceph's `bufferlist` and encapsulated with a header containing the operation type, a unique request ID, and the payload length. The encapsulated request is then routed to one of two paths: all client I/O operations are directed to the data plane for maximum throughput, while metadata management and cluster coordination operations use the control plane.

- *Control Plane:* Lightweight operations such as metadata management and status monitoring (e.g., `stat`, `exists`) are converted into serialized RPC messages and transmitted over a persistent TCP socket. The RPC server in the host-side `Proxy Interface` runs an event-driven loop that parses the operation type from the header upon message reception, invokes the corresponding BlueStore function, and returns the result. This channel is initialized once at OSD startup and provides a reliable, simple communication path suited for small, infrequent messages.
- *Data Plane:* For read and write operations that involve frequent, large data transfers (e.g., `queue_transactions`, `read`), CODA performs DMA-based direct memory transfer using the NVIDIA DOCA Communication Channel (ComCh) [36].

Data plane operations proceed as follows. On the write path, `ProxyObjectStore` on the DPU negotiates a memory region with the host via ComCh ❶, and the DOCA DMA engine directly transfers the serialized transaction data to a pre-exposed memory region on the host ❷. After the transfer completes, BlueStore writes the data to NVMe and sends a DONE-ACK to the DPU through the transaction completion callback (`OnCommit`) ❸. The read path operates in reverse: `ProxyObjectStore` sends read metadata via ComCh ❶, the host's BlueStore reads data from the device and transfers it to the DPU via DMA ❷, and the DPU returns the response to the client ❸. Both paths completely bypass the host kernel's network stack, minimizing overhead.

To reduce unnecessary memory copies on the data plane, CODA employs ring buffer staging. Serialized payloads are pushed into the ring buffer, from which the DMA engine reads directly for transfer; copying from the original pointer occurs only when ring space is insufficient, enabling near-zero-copy transfer in most cases. Because reads and writes have different communication patterns, the two paths are protected by separate mutexes to eliminate mutual interference. On DMA transfer failure, the system immediately falls back to the RPC path while preserving completed segments to prevent duplicate transfers.

C. Flow-Adapt: Size-Aware Adaptive DMA Execution

The additional hop introduced by the DPU-host split is a potential source of overhead. Furthermore, a hardware limitation restricts a single DMA transfer to approximately 2 MB [37].

CODA applies two optimization strategies depending on transaction size.

1) Transaction Batching ($\leq$2 MB): For workloads with frequent small transactions, the fixed per-transfer overhead of DMA (setup, header transmission, ACK waiting) accumulates and can degrade performance. To address this, transactions smaller than 2 MB are accumulated in a batch buffer and transferred in a single DMA operation.

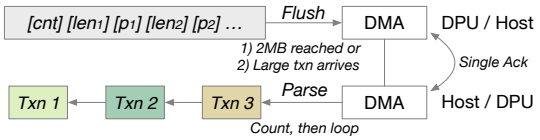


Fig. 4: Batching mechanism for DPU-host data transfer.

As illustrated in Figure 4, a batch is flushed when its size reaches 2 MB or when a transaction of 2 MB or larger arrives. The batch is encoded in the format [count][len$_1$][payload$_1$][len$_2$][payload$_2$]… and transferred via DMA. The receiver parses the batch, processes the N transactions sequentially, and sends a single ACK for the entire batch. This allows the fixed overhead of one DMA transfer to be amortized across multiple transactions, improving throughput.

2) Intra-request Pipelining (>2 MB): For requests of 2 MB or larger, CODA applies intra-request pipelining. When a client issues a write request larger than the maximum DMA transfer size, the total request size N is divided into $\lceil N/2\,\text{MB} \rceil$ segments, each sized as the minimum of the maximum transferable size and the remaining bytes.

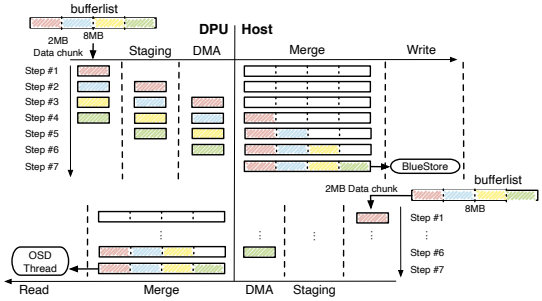


Fig. 5: DMA pipelining for DPU-host data transfer.

As shown in Figure 5, CODA pipelines the entire process using a staging buffer before DMA transfer and a write buffer after transfer. The staging buffer corresponds to the ring buffer described in Section V-B. As soon as the current DMA transfer begins, the system starts staging the next segment into the DMA-accessible buffer, allowing data preparation and data transfer to proceed concurrently. For example, an 8 MB write request is split into four 2 MB chunks (i, i+1, i+2, i+3). At a given point during pipeline execution:

- chunk(i): arrived in the host-side buffer, awaiting atomic commit
- chunk(i+1): DMA transfer in progress (DPU $\rightarrow$ host)
- chunk(i+2): waiting in the DPU-side staging buffer

Rather than performing ComCh negotiation on every transfer, CODA reuses pre-established memory regions to minimize transfer latency. After all chunks arrive in the host-side buffer, they are committed to BlueStore atomically to guarantee transaction atomicity.

This pipelining mechanism applies symmetrically to read operations, where a staging buffer on the host side enables overlapped execution for host-to-DPU DMA transfers. By overlapping communication and computation, CODA reduces per-request latency and minimizes idle cycles on the DPU.

D. Flow-Sync: Decoupled and Scalable Completion Handling

The physical separation of the DPU and host introduces synchronization complexity. We address this from two perspectives: OSD scalability and intra-OSD transaction consistency.

1) Per-channel Polling with Shared DMA Resource Pool: As the per-node OSD count increases, the number of DPU-host channels grows accordingly, making the polling design critical for scalability. A single dedicated thread traversing all channels suffers from growing traversal latency, while assigning one polling thread per channel scales thread count with OSDs and introduces lock contention on completion notifications. CODA adopts per-channel polling, in which the OSD thread that submitted a request directly polls only its own channel during the wait interval, without introducing a dedicated polling thread. Because each OSD accesses only its own channel, no lock contention arises with other OSDs, and by reusing existing OSD threads the total thread count remains identical to that of unmodified Ceph even as OSDs scale.

Meanwhile, the buffer `mmap` registration and DOCA object initialization required for DMA transfer incur a cost of hundreds of microseconds per request; creating these individually for each OSD would accumulate initialization overhead at scale. CODA shares a DPU-side transmit buffer pool and a host-side RX/TX buffer-`mmap` cache at the process level, so that channels are isolated per OSD while expensive DMA resources are reused across requests. Each OSD thread acquires a resource from the shared pool, transmits DMA data and ComCh metadata, and polls its own channel for completion. Write completion is determined through order-based completion tracking, and read completion through request-ID-based response matching; upon completion or timeout, resources are returned to the pool. In DOCA, message reception and DMA completion do not automatically trigger callbacks; the application must explicitly invoke the DOCA Progress Engine (PE) to process pending events. In CODA, each OSD thread periodically invokes PE on its own ComCh connection to process only that channel's events.

2) Request-Completion Matching: To accurately identify and reliably deliver transaction completions in the split DPU-host environment, CODA employs a request-completion matching mechanism. Each transaction (or batch) is assigned a unique `req_id`, and the host includes the same `req_id` in the DONE-ACK completion response. The DPU matches completed requests on a one-to-one basis by comparing this value during the wait. Algorithm 1 shows the matching logic.

Algorithm 1 Request-Completion Matching

Global State: $g\_done \leftarrow 0$ (highest completed write rid, atomic); $g\_resp \leftarrow \{\}$ (map: $rid \rightarrow (len, result)$)
On TXN_ACK received $(rid, stage, result)$:
if $stage =$ DONE **then**
  repeat
    $prev \leftarrow g\_done$
    if $rid \leq prev$ **then**
      break ▷ already superseded
    end if
  until CAS$(g_done, prev, rid)$
end if
On READ_RESP received $(rid, len, result)$:
$g\_resp[rid] \leftarrow (len, result)$ ▷ insert into hashmap
Function IsDone(rid, is_write):
if is_write **then**
  return $g\_done \geq rid$ ▷ all $rid' \leq rid$ are done
else
  return $rid \in g\_resp$ ▷ check hashmap member
end if

On the write path, multiple OSD threads may simultaneously receive completion notifications, requiring concurrency control. The global variable g_done stores the highest completed `req_id` and is updated using a compare-and-swap (CAS) operation. CAS atomically replaces g_done with the new value (rid) only if its current value matches the expected value ($prev$). Even when multiple threads attempt concurrent updates, the most recent completed request ID is maintained without locks. A waiting thread checks whether $g\_done \geq rid$ to determine if its own request has completed. On the read path, each request ID is unique, so responses are inserted directly into the hashmap g_resp for one-to-one matching. A waiting thread checks whether $rid \in g\_resp$ to determine if its response has been received. For batched transfers, a single DONE-ACK is sent only after all transactions in the batch have completed, so one DPU-side wait corresponds exactly to one host-side ACK. This check is sound because DONE-ACKs arrive in submission order: a single ComCh channel preserves message ordering and BlueStore's `OnCommit` callback fires in submission order. If an ACK is lost or a DMA transfer fails, the request is retransmitted with the same `req_id`, and the host treats duplicate `req_id`s idempotently to prevent duplicate commits. Each completion event is processed in amortized $O(1)$ time. Writes update g_done via a single CAS, with retry under contention, and reads perform an $O(1)$ hashmap insertion.

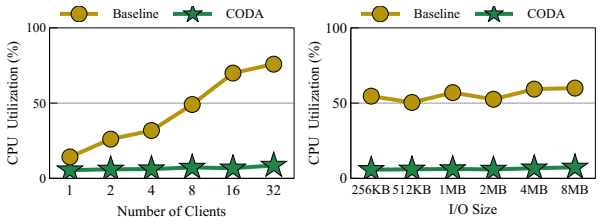
VI. Evaluation

A. Experimental Setup

Implementation. CODA is implemented on top of Ceph Quincy (v17.2), and uses NVIDIA DOCA SDK 2.2 for DPU-host communication. The complete implementation, including `ProxyObjectStore`, `Proxy Interface`, Flow-Direct, Flow-Adapt, and Flow-Sync, comprises approximately 27,000 lines of C++ and C code.

TABLE I: Testbed specification.

CPU	AMD Ryzen 9 3900X 12 cores
DPU	BlueField-3 integrated ConnectX-7
Memory	DDR4, 64 GB
Storage	SK hynix Platinum P41 SSD
Network (`Baseline`)	BlueField-3 integrated ConnectX-7 (NIC Mode, Ethernet 100Gbps)
Network (CODA)	BlueField-3 integrated ConnectX-7 (DPU Mode, Ethernet 100Gbps)



(a) Number of clients (b) I/O size

Fig. 6: Host CPU utilization comparison between BASELINE and CODA under single-node write workloads. CPU utilization is normalized to the 12 physical cores of one node.

Testbeds. Experiments are conducted on a testbed consisting of two nodes, each equipped with an NVIDIA BlueField-3 DPU. The two nodes are directly connected via 100 Gbps Ethernet. Detailed hardware specifications for each node are summarized in Table I. Each node partitions a single NVMe SSD into four partitions, allowing up to four OSDs per node. On our two-node testbed, we configure CRUSH so that each OSD is registered under its own host bucket, so that OSDs on the same physical node are treated as distinct hosts for replica placement and PG distribution. Combined with Ceph's shared-nothing architecture, in which each OSD operates as an independent unit, this lets OSD-count scaling on the testbed serve as a meaningful proxy for cluster-wide scaling dynamics.

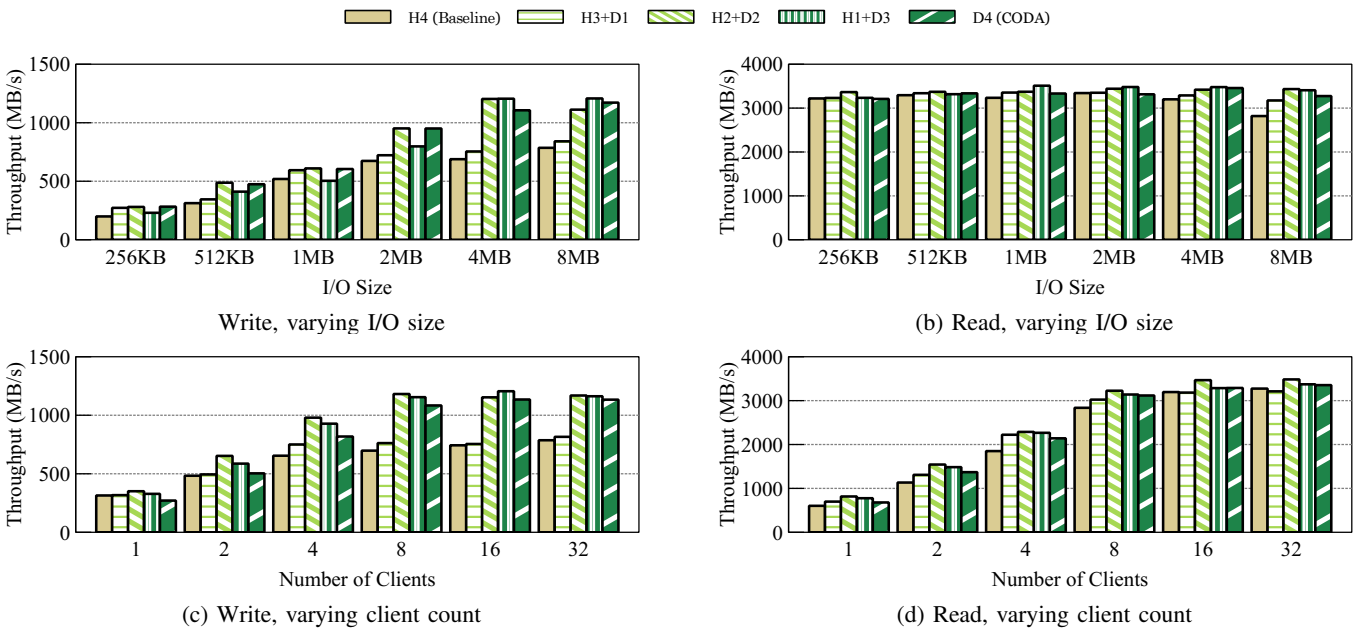
We evaluate two configurations:

- BASELINE: the stock Ceph configuration, in which BlueField-3 operates in NIC mode and the entire OSD stack, including Messenger, OSD Core, and BlueStore, runs on the host CPU.
- CODA: the proposed configuration, in which BlueField-3 is switched to DPU mode so that Messenger and OSD Core run on the DPU while only BlueStore and the Proxy Interface remain on the host (Figure 3).

Workloads. We use `rados bench` [34] to generate two workload modes: *Write-only*, which continuously creates objects of a specified size, and *Read-only*, which reads from pre-created objects. We do not evaluate mixed read/write ratios because RADOS bench is designed to stress a single I/O path at a time, and isolating each path provides clearer insight into per-path bottlenecks. For experiments with two or more OSDs, 3-way replication is enabled. Host CPU utilization is sampled every second using the `pidstat` utility.

B. Host CPU Utilization Under DPU Offloading

We evaluate the host CPU reduction achieved by CODA under Write-only workloads. One node serves as the storage server and the other as a dedicated client; the 100 Gbps direct connection eliminates network bottlenecks, isolating the storage server's CPU consumption and DPU offloading effect. All subsequent single-node experiments follow the same configuration. Figure 6 shows the host CPU utilization of BASELINE and CODA under varying client count and I/O



Write, varying I/O size

(b) Read, varying I/O size

(c) Write, varying client count

(d) Read, varying client count

Fig. 7: Throughput with 4 OSDs and no replication. (a) and (b) vary I/O size with 16 client threads, while (c) and (d) vary client count with 4 MB I/O. Hn denotes n BASELINE OSDs and Dn denotes n CODA OSDs.

size. Figure 6(a) varies the client count from 1 to 32 with 4 MB I/O, and Figure 6(b) varies I/O size from 256 KB to 8 MB with 16 client threads. As clients scale from 1 to 32, BASELINE's CPU utilization climbs steeply from 14.2% to 75.9%, whereas CODA remains flat at 5.5–8.6%. A similar contrast appears across I/O sizes: BASELINE fluctuates between 50.4% and 60.0%, while CODA stays within 5.8–7.3%.

In both cases, CODA's host CPU consumption remains largely insensitive to workload variation. This is because the Messenger and OSD Core, which dominate CPU consumption in BASELINE, have been relocated to the DPU, leaving only BlueStore and the Proxy Interface on the host. As a result, host-side CPU consumption is primarily determined by BlueStore's disk I/O processing, which varies little with client concurrency or request size. This enables CODA to provide predictable CPU utilization regardless of workload characteristics, which is particularly beneficial when the CPU must be shared with co-located applications.

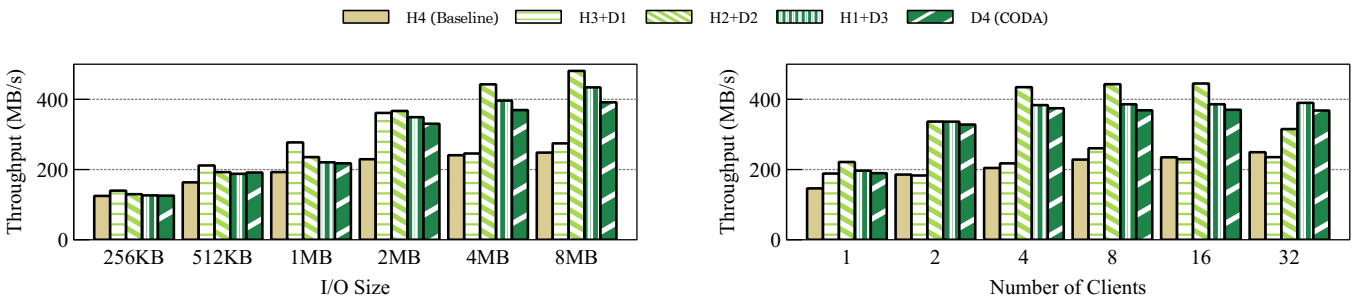
C. Impact of Host–DPU OSD Placement on Throughput

As the number of OSDs on a server increases, CPU contention among daemons significantly affects throughput. Furthermore, how OSDs are distributed between the host and the DPU leads to different performance characteristics. To investigate these effects, we deploy 4 OSDs on a single node and evaluate all placement ratios from H4 (all BASELINE) to D4 (all CODA). Figure 7 shows results. Figure 7(a) and (b) vary I/O size from 256 KB to 8 MB with 16 client threads, and (c) and (d) vary client count from 1 to 32 with 4 MB I/O. This section presents results without replication; the impact of replication is analyzed in Section VI-D.

Throughput recovery through DPU offloading. Across the write results in Figures 7(a) and (c), H4 consistently exhibits the lowest throughput under all conditions, and throughput recovers as more CODA OSDs are introduced. At 8 MB I/O, D4 achieves 49% higher throughput than H4. At 32 clients, H4 plateaus at 786 MB/s while configurations with CODA OSDs continue to scale with increasing load. This is because, as shown in Figure 6, the host CPU in H4 saturates at approximately 70%, leaving no headroom for additional requests. CODA absorbs CPU-intensive front-end logic on the DPU, preventing host CPU saturation and allowing throughput to grow with load. Note that this throughput recovery is achieved despite the additional DPU-host communication hop introduced by CODA, confirming that Flow-Direct's DMA transfer path and Flow-Adapt's batching and pipelining effectively offset the communication overhead.

Optimal hybrid placement ratio. An interesting observation is that certain hybrid placements achieve higher throughput than the all-CODA configuration (D4). At 4 MB writes, H1+D3 and H2+D2 reach 1,205 MB/s and 1,203 MB/s respectively, approximately 9% higher than D4 (1,106 MB/s). When CPU contention is mild, placing a subset of OSDs as BASELINE allows those OSDs to exploit the host CPU's higher single-thread performance compared to the DPU's ARM cores. However, as the BASELINE OSD ratio increases, inter-daemon CPU contention offsets this advantage, causing throughput to drop sharply for H3+D1 and H4. This result indicates that an optimal BASELINE-to-CODA ratio exists depending on workload characteristics, and that incrementally introducing CODA OSDs into existing clusters is a viable deployment strategy.

Read workload trends. The read results in Figures 7(b)



(a) Write, varying I/O size (b) Write, varying client count

Fig. 8: Write throughput with 4 OSDs under 3-way replication. (a) varies I/O size with 16 client threads, and (b) varies client count with 4 MB I/O. Hn denotes n BASELINE OSDs and Dn denotes n CODA OSDs

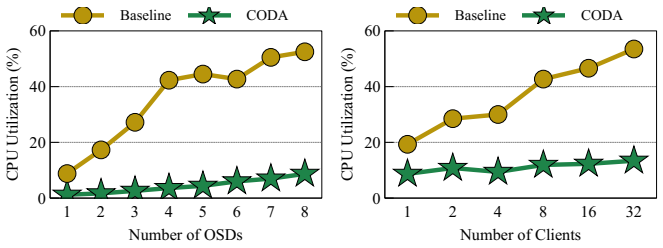
and (d) show a smaller throughput gap across configurations compared to writes. This is expected, as the read path does not involve replication coordination and is less CPU intensive, limiting the impact of inter-OSD CPU contention. Nevertheless, configurations with CODA OSDs consistently outperform H4; at 8 MB, H2+D2 delivers 22% higher throughput than H4.

D. Amplified Benefits of DPU Offloading Under Replication

3-way replication triggers replication traffic and coordination work on every write request, substantially increasing CPU load. This experiment evaluates how the offloading benefit of CODA changes when replication is enabled. Since replication affects only the write path, read throughput remains largely unchanged and is omitted from this analysis. Figure 8 shows write throughput under 3-way replication with 4 OSDs, where (a) varies I/O size from 256 KB to 8 MB with 16 client threads and (b) varies client count from 1 to 32 with 4 MB I/O.

Amplified offloading benefit under replication. The most prominent observation in the replication environment is that the performance gap between CODA and BASELINE widens significantly compared to the no-replication case. At 8 MB, H2+D2 achieves 94% higher throughput than H4, roughly double the 49% improvement observed without replication. The reason is clear: enabling replication adds Messenger's replication traffic processing and OSD Core's replica coordination, causing host CPU load to surge. In H4 this entire burden falls on the host, whereas CODA absorbs the replication-related work on the DPU, freeing the host CPU to focus on BlueStore. That is, the offloading benefit grows with replication overhead. Since replication is enabled by default in production deployments, this result highlights the practical relevance of CODA.

Shift in optimal hybrid placement under replication. As in the no-replication case, hybrid placements tend to achieve higher throughput than pure CODA deployment. At 4 MB, H2+D2 reaches 443 MB/s, 20% higher than D4 (369 MB/s). However, the minimum CODA OSD requirement changes under replication. H3+D1 performs similarly to or even worse than H4, because a single CODA OSD is insufficient to alleviate the host-side CPU load induced by replication. This result indicates that at least two CODA OSDs must be deployed to realize effective offloading benefits under replication.



(a) OSD scaling (b) Number of clients

Fig. 9: Host CPU utilization under Write-only workloads with 3-way replication as OSDs scale from 1 to 8 across two storage nodes. CPU utilization is normalized to the aggregate 24 physical cores of both nodes (12 cores × 2 nodes).

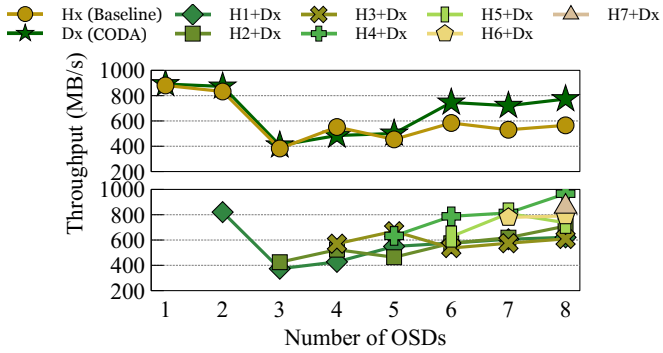


Fig. 10: Throughput with 3-way replication as OSDs scale from 1 to 8. Top: all BASELINE (Hx) and all CODA (Dx). Bottom: hybrid placements (Hn+Dx) with varying BASELINE-to-CODA ratios.

Scalability under increasing load. Figure 8(b) shows that as the client count increases, H4 quickly plateaus in the 200–249 MB/s range, while H2+D2 scales to 315–445 MB/s with load. This trend is consistent with the no-replication case, but under replication the saturation point of H4 arrives earlier. Replication increases per-daemon CPU consumption, exhausting host CPU headroom more rapidly, which further reinforces the need for DPU offloading in replicated environments.

E. Scaling OSDs: CPU Efficiency and Throughput Trade-offs

To evaluate how CODA scales as the OSD count grows, we increase the number of OSDs from 1 to 8 with 3-way replication enabled, generating inter-node replication traffic.

OSDs are interleaved across the nodes for load balancing (e.g., OSD 0 on node 1, OSD 1 on node 2, and so on). The client runs on one of the storage nodes. Since Ceph follows a shared-nothing architecture where each node operates independently, the scaling behavior observed in this configuration is expected to generalize to larger clusters. CPU utilization is reported as a percentage of the total host CPU resources (24 cores).

CPU utilization. Figure 9 shows host CPU utilization in the distributed cluster. In Figure 9(a), as the OSD count increases from 1 to 8, BASELINE surges from 8.8% to 52.2%, consuming more than half of the total host CPU, while CODA rises moderately from 1.2% to 8.7%. At 8 OSDs, CODA shows approximately 83% lower CPU utilization than BASELINE, consistent with the trend observed in the single-node experiments. In Figure 9(b), as the client count increases with 8 OSDs, BASELINE rises from 19.3% to 53.5%, while CODA remains stable in the 8.7–13.4% range. This demonstrates that CODA maintains predictable host CPU utilization even in a distributed environment.

Throughput scaling. Figure 10 shows throughput under 4 MB Write-only workloads with 16 client threads as OSDs scale from 1 to 8. At 1–2 OSDs all configurations perform similarly (820–892 MB/s), but throughput drops approximately 54% at 3 OSDs as 3-way replication activates. Beyond this point, Hx (all BASELINE) plateaus at 454–583 MB/s, while Dx (all CODA) scales progressively to 773 MB/s at D8, 37% higher than H8. This gap directly reflects the CPU saturation observed in Figure 9(a): BASELINE reaches 52.2% at 8 OSDs, whereas CODA remains at 8.7%.

Hybrid placement. H4+D4 achieves 966 MB/s, 71% higher than H8 and 25% higher than D8, representing the best configuration overall. This balance point exploits both the host CPU's higher single-thread performance and the DPU's offloading benefit. Configurations with an excessive CODA ratio (H1+D7, H2+D6) fall below H4+D4, as the DPU's ARM cores become the bottleneck.

VII. Related Work

Reducing host CPU load with DPUs. Offloading data path processing from the host CPU to DPUs has been actively studied across diverse storage and networking domains. DDS [13] processes read requests directly on the DPU in disaggregated storage architectures, and OS2G [14] relocates the object storage client to the DPU with GPU-direct support. DPC [38] offloads file system client operations to the DPU, while DPU-KV [39] performs fine-grained offloading of the communication engine for in-memory key-value stores. HiDPU [40] proposes DPU-aware hybrid indexing to minimize host-DPU communication overhead, and LEED [41] re-architects the I/O and memory interface for SmartNIC-based key-value operations.

On the hardware acceleration side, PEDAL [15] and INEC [16] leverage on-chip compression and erasure coding engines, Fuyao [42] enables sub-millisecond direct data transfer between serverless functions, and FSDP acceleration [43] offloads collective communication for distributed training.

At the kernel network layer, AccelTCP [20] performs TCP protocol processing on a SmartNIC to reduce host network stack overhead.

DPU-based storage application optimization. Research on offloading internal operations of storage engines to DPUs has also progressed. In LSM-tree based key-value stores, DComp [18] offloads RocksDB compaction to the DPU's hardware compression accelerators and ARM cores to reduce host CPU contention, and D2Comp [19] extends this to disaggregated environments, additionally mitigating the network overhead induced by compaction. DFlush [17] offloads flush operations to the DPU, achieving throughput improvements through pipeline parallelism. In disaggregated persistent memory environments, DoW-KV [44] designs a two-tier index comprising a cache table on the DPU and a persistent table on PM, absorbing and merging small random writes on the DPU side to reduce CPU load on the PM server. LineFS [45] offloads replication, publication, compression, and consistency management of a persistent memory based DFS to a SmartNIC. While LineFS redesigns the DFS itself, CODA preserves the production Ceph OSD stack and selectively decouples only the CPU-intensive front-end from the host resident BlueStore backend.

Despite prior work, applying DPUs to Ceph has received little attention. CODA builds on our prior workshop work, DoCeph [46], which decoupled a single Ceph OSD onto the BlueField-3 DPU to reduce host CPU consumption, with only the write path implemented. CODA extends this work toward OSD scalability, which raises challenges absent in the single OSD setting: contention on the DPU host transfer path, per request DMA overhead amplified by request count, and synchronization cost that grows with the number of OSDs.

VIII. Conclusion

In this paper, we presented CODA, a Ceph OSD decoupling architecture that separates the OSD stack across the host and the DPU by offloading the CPU-intensive front-end path, including Messenger and OSD Core, while retaining BlueStore on the host. CODA addresses the fundamental host-side CPU bottleneck that limits dense OSD deployment by moving communication and replication-intensive processing to the DPU and preserving backend persistence on the host. Our results show that CODA reduces host CPU utilization by up to 88%, improves throughput by up to 49% without replication and by up to 94% with 3-way replication. These results highlight the effectiveness of DPU-based OSD decoupling for improving Ceph performance under host CPU constraints.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00453929) (RS-2024-00416666). This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] S. Cohen, G. Fidente, and S. Han, "Distributed hyperconvergence: Pushing OpenStack and Ceph to the edge." Presented at OpenStack Summit Berlin, https://www.openstack.org/summit/berlin-2018/summit-schedule/events/22984/. Accessed: 2025-06-01.

[2] MicroCloud, "Canonical microcloud." https://canonical.com/microcloud.

[3] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung, "FlashShare: Punching through server storage stack from kernel to firmware for Ultra-Low latency SSDs," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 477–492, USENIX Association, Oct. 2018.

[4] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 154–161, IEEE, 2017.

[5] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[6] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 603–616, USENIX Association, July 2019.

[7] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "OS I/O path optimizations for flash solid-state drives," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 483–488, USENIX Association, June 2014.

[8] S. Hynix, "Storage system using nvme over fabric ssd-based ethernet jbof." https://files.futurememorystorage.com/proceedings/2019/08-08-Thursday/20190808_NVMF-302B-1_Chung.pdf.

[9] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pp. 307–320, 2006.

[10] Nvidia, "Nvidia bluefield-3 dpu: Programmable data center infrastructure on-a-chip." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2021.

[11] Intel, "Intel infrastructure processing unit(Intel IPU)." https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2022.

[12] AMD, "Amd pensando dpu technology: Front-end networking powering modern data centers." https://www.amd.com/en/products/data-processing-units/pensando.html, 2022.

[13] Q. Zhang, P. A. Bernstein, B. Chandramouli, J. Hu, and Y. Zheng, "Dds: Dpu-optimized disaggregated storage," *VLDB Endowment*, vol. 17, p. 3304–3317, July 2024.

[14] Z. Jin, Y. Chen, M. Liang, Y. Wang, G. Fang, A. Zhou, K. Zhang, J. Xu, W. Lin, Y. Lin, S. Zhao, W. Shi, Z. He, S. Cai, and W. Chen, "Os2g: A high-performance dpu offloading architecture for gpu-based deep learning with object storage," in *30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, (New York, NY, USA), p. 750–765, Association for Computing Machinery, 2025.

[15] Y. Li, A. Kashyap, W. Chen, Y. Guo, and X. Lu, "Accelerating lossy and lossless compression on emerging bluefield dpu architectures," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 373–385, 2024.

[16] H. Shi and X. Lu, "Inec: Fast and coherent in-network erasure coding," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–17, 2020.

[17] C. Ding, K. Lu, Q. Zhang, Z. Ye, T. Yao, D. Wang, H. Wu, and J. Wan, "Dflush: Dpu-offloaded flush for disaggregated lsm-based key-value stores," *Proc. ACM Manag. Data*, vol. 3, June 2025.

[18] C. Ding, J. Zhou, J. Wan, Y. Xiong, S. Li, S. Chen, H. Liu, L. Tang, L. Zhan, K. Lu, *et al.*, "Dcomp: Efficient offload of lsm-tree compaction with data processing units," in *52nd International Conference on Parallel Processing*, pp. 233–243, 2023.

[19] C. Ding, J. Zhou, K. Lu, S. Li, Y. Xiong, J. Wan, and L. Zhan, "D2comp: Efficient offload of lsm-tree compaction with data processing units on disaggregated storage," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 3, pp. 1–22, 2024.

[20] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 77–92, 2020.

[21] AWS, "Amazon s3." https://aws.amazon.com/de/s3/, 2025. Accessed: 2025-06-01.

[22] I. MinIO, "Minio." https://www.min.io, 2025. Accessed: 2025-06-01.

[23] OpenStack, "Swift." https://docs.openstack.org/swift/latest/. Accessed: 2025-06-01.

[24] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: a scalable, reliable storage service for petabyte-scale storage clusters," in *2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pp. 35–44, 2007.

[25] Rook, "Rook (kubernetes ceph operator)." https://rook.io.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *2006 ACM/IEEE conference on Supercomputing*, 2006.

[27] S. Weil, "Bluestore: A new storage backend for ceph one year in," *March2017. Retrieved August*, vol. 19, p. 2018, 2017.

[28] Ceph Foundation, "Hardware recommendations." https://docs.ceph.com/en/reef/start/hardware-recommendations/, 2025. Accessed: 2025-06-01.

[29] Marvell, "Marvell octeon data processing units." https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-white-paper.pdf, 2021.

[30] Broadcom, "Broadcom stingray smartnic accelerates baidu cloud services." https://www.broadcom.com/company/news/product-releases/53106, 2024.

[31] A. W. Service, "Aws nitro system." https://aws.amazon.com/ec2/nitro/, 2024.

[32] Alibaba Cloud, "A detail explanation about alibaba cloud cipu." https://www.alibabacloud.com/blog/a-detailed-explanation-about-alibaba-cloud-cipu_599183, 2022. Accessed: 2025-06-01.

[33] Mircrosoft, "Enhancing infrastructure efficiency with azure boost dpu." https://techcommunity.microsoft.com/blog/azureinfrastructureblog/enhancing-infrastructure-efficiency-with-azure-boost-dpu/4298901, 2024.

[34] Ceph, "Rados object storage utility." https://docs.ceph.com/en/latest/man/8/rados/, 2016.

[35] Ceph, "Filestore config reference." https://docs.ceph.com/en/reef/rados/configuration/filestore-config-ref/, 2016.

[36] NVIDIA, "Doca communication channel." https://docs.nvidia.com/doca/sdk/doca+comch/index.html, 2024.

[37] A. Kashyap, Y. Li, D. Ng, and X. Lu, "Understanding the idiosyncrasies of emerging bluefield dpus," 2025.

[38] K. Zhong, Z. Yu, Q. Li, X. Luo, L. Long, Y. Tan, A. Ren, and D. Liu, "Dpc: Dpu-accelerated high-performance file system client," in *53rd International Conference on Parallel Processing*, pp. 63–72, 2024.

[39] A. Kashyap, Y. Li, and X. Lu, "Dpu-kv: On the benefits of dpu offloading for in-memory key-value stores at the edge," in *34th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 1–14, 2025.

[40] W. Zhu, Z. Shen, Q. Wei, R. Chen, X. Yao, D. Yu, and Z. Shao, "HiDPU: A DPU-Oriented hybrid indexing scheme for disaggregated storage systems," in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, (Santa Clara, CA), pp. 271–285, USENIX Association, Feb. 2025.

[41] Z. Guo, H. Zhang, C. Zhao, Y. Bai, M. Swift, and M. Liu, "Leed: A low-power, fast persistent key-value store on smartnic jbofs," in *ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, (New York, NY, USA), p. 1012–1027, Association for Computing Machinery, 2023.

[42] G. Liu, L. Zhao, Y. Li, Z. Duan, S. Chen, Y. Hu, Z. Su, and W. Qu, "Fuyao: Dpu-enabled direct data transfer for serverless computing," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, (New York, NY, USA), p. 431–447, Association for Computing Machinery, 2024.

[43] M. Khalilov, S. Di Girolamo, M. Chrapek, R. Nudelman, G. Bloch, and T. Hoefler, "Network-offloaded bandwidth-optimal broadcast and allgather for distributed ai," SuperComputing'24, IEEE Press, 2024.

[44] Y. Zhang, G. Li, J. Wan, J. Wang, J. Li, T. Yao, H. Wu, and D. Wang, "Dow-kv: A dpu-offloaded and write-optimized key-value store on disaggregated persistent memory," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 271–283, IEEE, 2023.

[45] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 756–771, 2021.

[46] K. Park, S. Yoon, F. Talibli, S. Park, J.-H. Kwak, K. Jeong, A. Khan, and Y. Kim, "Doceph: Dpu-offloaded messaging in ceph for reduced host cpu utilization," in *Proceedings of the SC'25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1658–1666, 2025.