DeLiBA-K: Speeding-up Hardware-Accelerated Distributed Storage Access by Tighter Linux Kernel Integration and Use of a Modern API

Babar Khan, Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt, Germany
{khan,koch}@esa.tu-darmstadt.de

***Abstract*—We present DeLiBA-K, an improved version of the Development of Linux Block I/O Accelerators (DeLiBA) framework. DeLiBA-K operates at the Linux kernel level, bypassing the user-space interactions of DeLiBA-1 and -2 to interact with the block and network I/O kernel stack directly. Another critical feature of DeLiBA-K is implementing and benchmarking the modern io_uring Asynchronous I/O (AIO) API within a 16nm AMD Alveo U280 FPGA I/O framework. This allows for better parallelism and reduced latency in I/O operations. Our results show significant performance gains, up to a 3.2x improvement in I/O operations per second (IOPS) and 3.45x increased throughput for synthetic workloads. Real-world applications see a 30% reduction in execution time for data-intensive tasks. DeLiBA-K has been successfully tested in an industrial environment using real workloads, demonstrating its effectiveness in large-scale enterprise environments.**

I. Introduction

Datacenter I/O performance has become increasingly critical as the volume of data processing continues to grow, and it is estimated to reach 175 ZB in 2025 [1]. Additionally, as AI/ML models become more sophisticated and datasets grow more larger, the need for efficient data handling and processing capabilities becomes increasingly challenging for these large-scale AI applications [2] in storage [3]. To address these challenges, researchers and industry professionals have been exploring the use of specialized hardware accelerators like Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs) to cope with the deluge of I/O workload.

FPGA-based SmartNICs have become off-the-shelf hardware in recent years. However, their programming frameworks for distributed storage have *two* main drawbacks. First, many FPGA SmartNIC frameworks rely on decades-old system calls and treat storage devices as *black boxes*. To fully leverage the potential of these accelerators, it is necessary to delve deeper into their interaction with the operating system kernel, generally Linux in HPC [4], and understand their impact on the overall system design. Second, many solutions often deploy FPGA accelerators as *passive* offloading devices rather than autonomous, active participants in the storage stack.

Previous versions of the open-source DeLiBA framework, namely DeLiBA-1 (D1) [5] and DeLiBA-2 (D2) [6], demonstrated significant gains in accelerating Linux block I/O operations. However, these prior versions still had bottlenecks, specifically in how user programs interacted with the storage stack using decades-old traditional Linux APIs and in their internal implementation, which incurred many user/kernel context switches. We have carefully considered these two performance bottlenecks in our new DeLiBA-K framework.

First, we have extended and optimized the FPGA-based network and storage stack by redesigning it in RTL, enhancing interactions between the FPGA and host system to minimize latency and improve throughput performance. Additionally, we implemented partial-bitstream-based reconfigurability on the FPGA, allowing it to handle additional tasks independently without context switches.

Second, we have enhanced the host-side API through new system calls that offer fine-grained control over read and write I/O operations. The updated framework supports the state-of-the-art `io_uring` API for Asynchronous I/O [7]–[12], instead of the traditional read() / write() call based interaction. The fine-grained control allows for more efficient use of the I/O subsystem, reducing overhead and improving data throughput.

The subsequent sections of this paper are structured as follows: In Section II, we discuss the performance challenges in the monolithic Linux kernel affecting block and network I/O operations. Section III outlines the architecture of the DeLiBA-K framework, with a focus on the software baseline. Section IV describes recent enhancements in the hardware architecture of DeLiBA-K. Section V presents detailed hardware performance evaluation. Section VI reviews related research, and Section VII concludes with a summary.

II. Limitations of Traditional I/O APIs

To justify our overall architecture and subsequent microarchitectural design choices, we argue that both new [13]–[19] and relatively new [19]–[23] FPGA frameworks designed to accelerate I/O in systems require a critical reassessment of decades-old system calls that were prevalent in earlier FPGA frameworks [20], [22], [24]–[33]. Despite the significant acceleration achieved by various FPGA frameworks, we contend that existing system calls do not always perform their intended functions effectively.

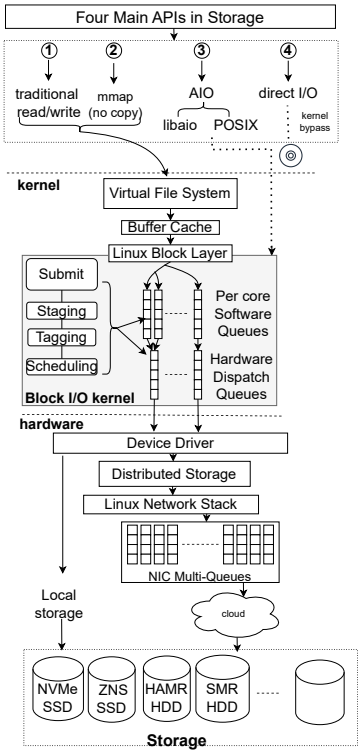


Fig. 1: Monolithic I/O in Linux Kernel

As shown in the figure, the application layer in the Linux I/O system interfaces with *four* main types [34] of I/O operations: traditional **`read()/write()`**, **`mmap`** [35]–[40], asynchronous I/O **`AIO`** [41]–[47], and **`O_DIRECT`** [48]–[51].

First, traditional **`read()/write()`** operations (buffered pool) are the most common [52], involving the standard file system calls to read data from or write data to storage devices. These read/write I/Os are inherently synchronous, i.e., the calling thread is blocked until the I/O is complete. Unfortunately, the synchronous model runs into severe performance problems on modern hardware. The results in work [41], [53]–[55] quantify the performance impact of synchronous I/O operations and show that blocking due to synchronous I/O can degrade the performance of disk-intensive benchmarks by two orders of magnitude.

Memory-mapped file I/O (**`mmap`**) [35]–[40] is a widely used technique that allows applications to map files or devices directly into memory, enabling direct access to data via memory addresses and potentially enhancing performance for specific workloads by eliminating the need for explicit read and write operations. However, it does have many issues in more complex scenarios, as raised in a notable study by Crotty et al., which [35] presents a compelling case against the blanket use of **`mmap`** in distributed systems, including databases, arguing that it is not suitable in scenarios where explicit control over memory management and handling of page faults is necessary, and high throughput on fast persistent storage devices is essential.

The third I/O technique is Asynchronous I/O (AIO). There are two types of AIO in Linux: libaio (native async I/O interface) [41]–[47] and POSIX [56]–[59]. Both have different APIs [60]. The Linux libaio library, introduced in the early 2000s, suffers from several limitations, i.e, it only supports asynchronous I/O for **`O_DIRECT`** (unbuffered) accesses.

On the other hand, the **`POSIX`**-based aio system call, which is nearly 30 years old, also faces significant challenges. The work titled "POSIX is Dead" [57] critiques its applicability in modern distributed systems. Lastly, direct I/O (non-buffered) [48]–[51] bypasses the kernel's page cache, directly transferring data between user space and storage devices. While this approach is effective in eliminating some I/O stack overheads, it has other implications, such as requiring a dedicated filesystem (fs).

Apart from the direct I/O, all three major I/O APIs go through the block I/O layer. The Linux block layer is a kernel subsystem that is responsible for handling block devices, e.g., hard disk drives (HDDs), SMR HDDs [61], [62], HAMR [63] [1], solid state disks (SSDs), ZNS [64]–[66] and remote storage. Over the years, the block layer has changed significantly from a single-request queue to a multi-queue model [67], as shown in Figure 1. Explicit multi-queuing support was added with Linux 3.13, and since Linux 5.0, the old single-queue implementation has been removed. This queuing scheme applies not just to modern storage devices used locally (e.g., SSDs) but also to the modern network interfaces (NICs) used to access remotely distributed storage, as these can send or receive packets in multiple hardware queues. Apart from the monolithic mainline Linux kernel I/O, there have been attempts to design operating systems with a more *fine-grained* control of I/O, including microkernels [68]–[70], and LibraryOS [71]–[73]. Despite these benefits, FPGA-based frameworks [13]–[19], [19]–[23] deployed in modern data centers continue to rely on the traditional Linux kernel for storage I/O acceleration.

III. DeLiBA-K Framework Architecture

The existing Linux I/O layer's lack of fine-grained control has been a significant performance bottleneck, as discussed in the previous section (Section II). The DeLiBA series of Linux block I/O accelerators focuses on speeding up client-side processing for the distributed Ceph file system [74]–[78]. However, the previous DeLiBA frameworks, namely DeLiBA-1 (D1) and DeLiBA-2 (D2), also faced the performance challenges discussed above. Specifically, *three* main performance pain points were identified in previous DeLiBA frameworks: 1) extensive memory copies between user-space and kernel, 2) lack of multi-tenancy support, and 3) programming model.

We have developed a significantly improved version of DeLiBA, called DeLiBA-K, to address these three issues. The first problem stemmed from the numerous kernel-userspace context switches and copying operations. To be precise,

[1]HAMR is a drive based on new media magnetic technology

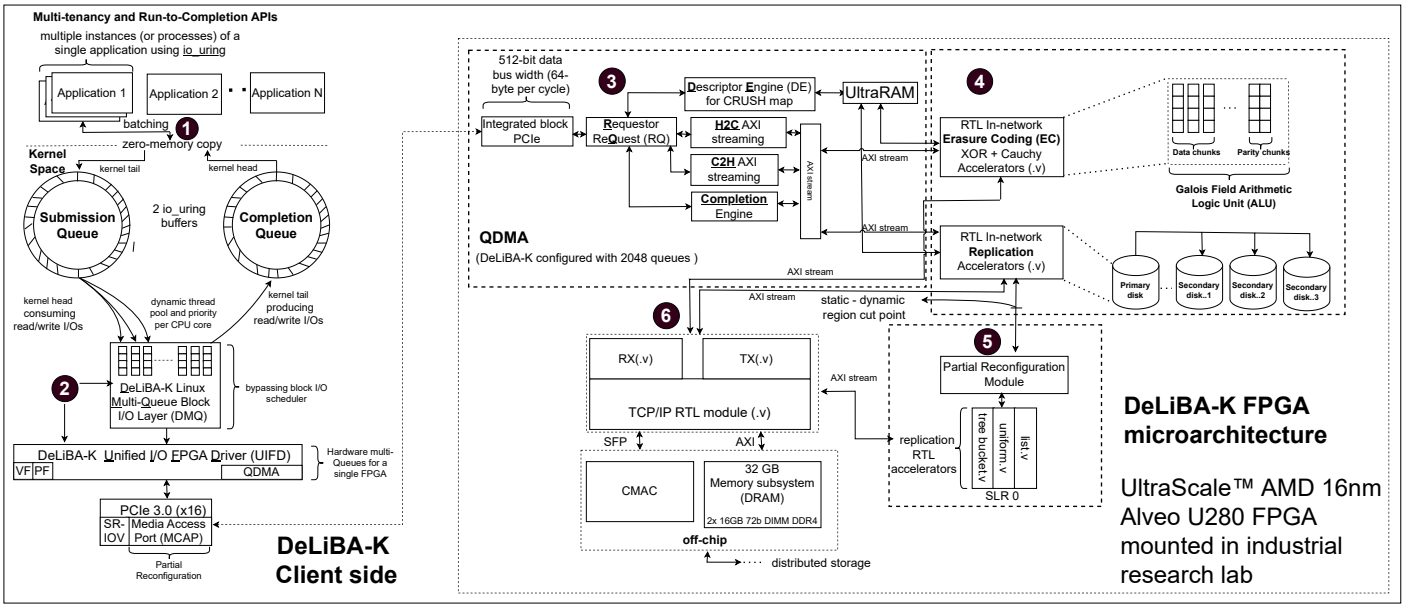


Fig. 2: Optimized DeLiBA Framework namely DeLiBA-K. Six ball points indicating the lifecyle of an I/O and likewise 6 major optimizations

DeLiBA-1 (D1) had at least *six* such context switches each per **`read()/write()`** call, with the previous DeLiBA-2 (D2) going through this copying process *five* times. This was evident in the results of both previous both DeLiBA frameworks. Despite using FPGA acceleration, DeLiBA-1 achieved only modest speedups of 1.9x for sequential reads (4 kB) and 1.2x for random writes (128 kB). DeLiBA-2, which moved the network stack onto the FPGA as well, showed more improvement, with cluster-level speedups now reaching up to 2.8x for both throughput and IOPS relative to software Ceph.

The second problem was the need for multi-tenancy support. Multi-tenancy is crucial in modern data centers, allowing multiple users or tenants to share computing resources. This feature became even more critical, as we aimed to deploy DeLiBA-K in an industry partner's[2] research lab, where real-world workloads were evaluated. Neither of the previous DeLiBA frameworks had support for multi-tenancy.

The third issue was related to the system architecture and the user APIs. Both previous DeLiBA frameworks relied heavily on multiple user-space libraries, which complicated the system architecture. These libraries, primarily NBD and Ceph-specific libraries, were not easy to maintain or scale. Moreover, they hindered the implementation of effective *asynchronous* I/Os that are important in multi-tenant scenarios where numerous concurrent I/O operations frequently occur.

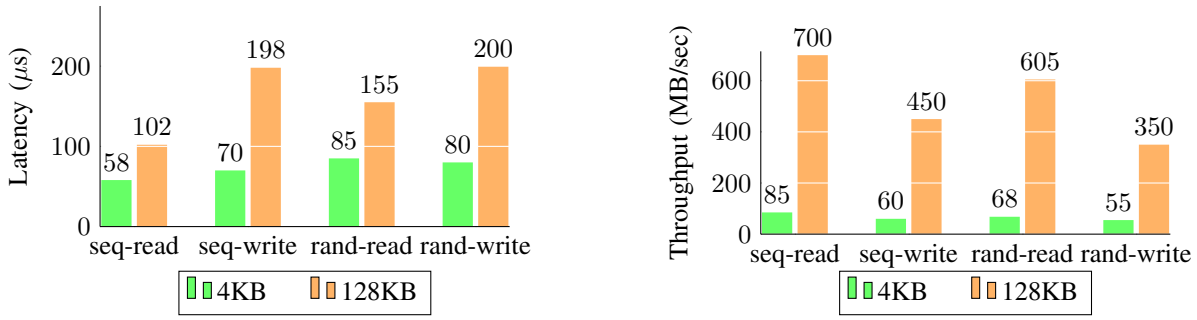
Figure 2 shows the main architecture of DeLiBA-K framework and it explicitly illustrates the lifecycle of an I/O request in *six* stages, each corresponding to one of the six major optimizations of the DeLiBA-K framework. It is important to note that these six stages do not represent context switches, as were the cases in the previous DeLiBA framework architectures, where bullet points were used to indicate them. In the following sections (Section III-A and Section III-B), we will discuss the application side and the kernel library in detail, focusing on how they relate to the first two optimizations in DeLiBA-K and the client-side components depicted in the Figure 2.

[2]The partner company, which is a global leader in DBMS and cloud operations, cannot be named for confidentiality reasons.

A. Application: Batching, Zero Memory Copy and Asynchronous I/O

The application end of the DeLiBA-K framework implements the **`io_uring`** I/O library [7]–[12], a new asynchronous I/O (AIO) interface introduced in the mainline Linux kernel version 5.1. This modern API allows considerably finer control of I/O than the traditional I/O APIs described in Section II. The name "io_uring" is derived from its use of two ring buffers in the interface. As shown in Figure 2, each io_uring instance consists of two main components (two ring buffers): **S**ubmission **Q**ueue (hereafter SQ) and **C**ompletion **Q**ueue (hereafter CQ). The application uses the SQ to submit I/O read/write requests to the kernel, while the kernel uses the CQ to return completed I/O operations to the application. This design allows for efficient, non-blocking I/O operations, significantly reducing the performance bottleneck caused by extra memory copies in previous DeLiBA frameworks. These fewer memory copies (illustrated by black circle ❶) are achieved by using ring buffers in io_uring as shown in Figure 2. The use of these two ring buffers minimizes memory copies, as they enable direct communication between the application and the kernel, eliminating the need for intermediate data copying between the application and kernel space.

One of the key features of io_uring is its support for batching (also illustrated by circle ❶), which allows multiple I/O requests to be submitted together in a single system call. In the context of io_uring, batching refers to the ability to



(a) Latency **(b)** Throughput

Fig. 3: Pure software baseline in replication mode: Latency and throughput of 4 kB and 128 kB I/Os using io_uring APIs in DeLiBA-K

submit multiple I/O requests in a single system call, thereby reducing the overhead associated with making multiple system calls. In contrast, traditional read and write operations typically require separate system calls for each I/O operation, even when batched. To this end, the earlier DeLiBA frameworks, based on traditional reads and writes, lacked the ability to truly batch operations at the system call level granularity, resulting in really minimal throughput increase.

This batching mechanism is implemented in io_uring by queueing up multiple SQ entries (SQEs) and then using a single call to **`io_uring_enter()`** to inform the kernel to process these requests.

Each SQE includes fields such as the operation type (e.g., read, write), the file descriptor, a pointer to the buffer, the buffer length, and additional flags for fine-grained control over the I/O operation.

Although each io_uring instance can be configured to operate in one of three modes: interrupt-driven, polled, or kernel-polled – DeLiBA-K specifically implements the kernel-polled mode. In this mode, the application actively checks the completion queue for any completed I/O operations, rather than waiting for interrupts.

While a single io_uring instance can provide substantial performance improvements, DeLiBA-K takes this concept further by creating *multiple* io_uring instances. This is achieved by calling the **`io_uring_setup`** system call multiple times, with each instance independently operating its own SQs and CQs. In current implementation, DeLiBA-K uses 3 instances as also shown in Figure 2.

In implementing this multi-instance design, a key decision was made to bind each io_uring instance of a particular application to a *specific* CPU core. This binding is achieved through the CPU affinity mechanism, which utilizes the **`sched_setaffinity`** system call. By assigning the submission queue threads of each io_uring instance to designated CPU cores, DeLiBA-K optimizes performance in several ways. First, it avoids contention on a single submission queue thread, which could become a bottleneck in high-load scenarios. Second, it ensures better utilization of available CPU cores by effectively distributing the I/O processing load across multiple cores. Finally, this approach can significantly improve cache locality, as each core consistently works with the same set of data structures associated with its assigned io_uring instance.

B. DeLiBA-K MQ and Unified Block I/O FPGA Driver

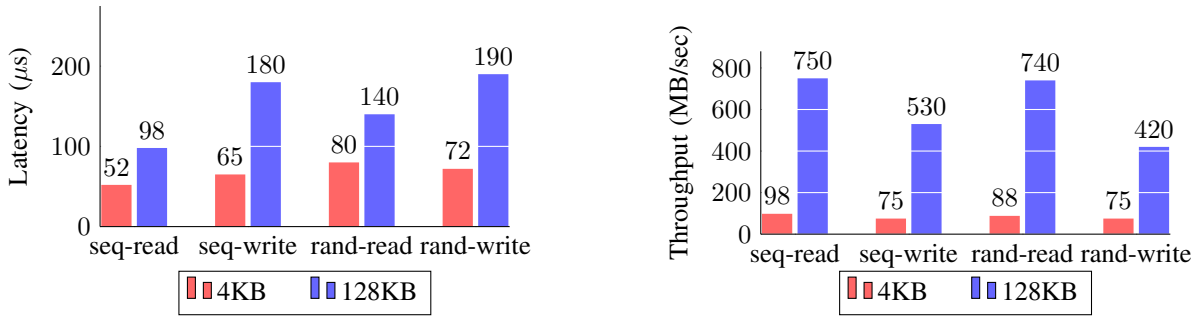
In the second phase (illustrated by circle ❷), the requests generated by each application pass through a modified Linux MQ-block I/O layer, referred to as the **D**eLiBA-K **MQ** (hereafter DMQ) layer in the main architecture of DeLiBA-K in Figure 2.

One of the main modifications is bypassing the block I/O MQ-scheduler which is DeLiBA-K specific design implementation. The bypass is implemented because each 'io_uring' instance of a particular application is already bound to a specific CPU core, rendering the block I/O scheduler's operations unnecessary.

The DMQ layer manages these requests using its multi-queue mechanism and forwards the I/O requests to a newly developed driver named the DeLiBA-K Unified I/O FPGA Driver. As the name indicates, the DeLiBA-K **U**nified **I**/O **F**PGA **D**river (hereafter UIFD) is an comprehensive unified kernel driver library developed from scratch, providing support for a range of storage devices, including emerging local storage such as ZNS and SMR disks[3]. At its core, the UIFD implements multiple hardware queues using the AMD's (earlier Xilinx) QDMA (Queue Direct Memory Access) [79]–[81] driver to talk to the actual FPGA cards via PCIe. When multiple io_uring instances are utilized, UIFD, managed by the DMQ layer, handles I/O requests from each instance concurrently. This multi-threaded per-core I/O processing is enabled by the multi-queue design of DMQ, which distributes I/O workloads across multiple queues per single hardware in UIFD driver. Each io_uring instance, bound to a specific CPU core, aligns directly with a corresponding per-hardware queue, reducing overhead from queue contention and inter-core communication.

In addition to the QDMA interface, the UIFD provides access to the CMAC block on the FPGA. This is particularly useful in scenarios like network monitoring in data centers,

[3]We have access to physical SMR and ZNS disks but it is out-of-scope for this work. We did not run DeLiBA-K tests on ZNS disks but we have run tests on SMR disk



(a) Latency **(b)** Throughput

Fig. 4: Pure software baseline in Erasure Coding mode: Latency and throughput of 4 kB and 128 kB I/Os using io_uring APIs in DeLiBA-K

where data volumes are small, and the system may rely solely on the CMAC interface without needing the QDMA.

Given that our industrial research partner operates data centers and cloud deployments where virtual disks of virtual machines play a significant role, it was equally important to consider the virtual disks. Additionally, since Ceph [74]–[78] has been a primary use case in our previous frameworks and continues to be in the DeLiBA-K framework, it was necessary to leverage Ceph's virtual disk [82] functions in its RBD kernel driver, which we had yet to utilize in earlier frameworks. To this end, UIFD includes a DeLiBA-K specific Ceph RBD virtual disk driver that adopts QDMA's virtualization functions. QDMA implements SR-IOV passthrough virtualization (thin hypervisor model) where the adapter exposes a separate virtual function (VF) for use by a virtual machine (VM).

C. Software Baseline and Evaluating DeLiBA-K with traditional Linux APIs in previous DeLiBA-2

After developing the new host-side interface in DeLiBA-K (Figure 2), we adopt the same incremental approach used in DeLiBA-1 (D1) and DeLiBA-2 (D2) for the initial benchmark. This initial benchmark was conducted without FPGA acceleration in the cluster, establishing two key points. First, it demonstrated the io_uring based improvement in DeLiBA-K compared to particularly DeLiBA-2 (D2) framework as host-side libraries (without FPGA). To be more precise, a software baseline of DeLiBA-K versus a software baseline of DeLiBA-2 (D2). This is significant since DeLiBA-K includes a modified Linux kernel block I/O layer (DMQ explained in Section Section III-B) and an improved Ceph-RBD kernel driver in its UIFD stack. Additionally, the application interface now uses the io_uring API instead of the NBD APIs used in previous DeLiBA-2 (D2). Second, this benchmark provided an initial understanding of the network and storage stack optimizations needed in DeLiBA-K to achieve overall FPGA-based acceleration compared to DeLiBA-2(D2) FPGA-based acceleration.

1) ***Software Testbed****:* The software testbed consists of a Ceph cluster built inside the industrial lab, featuring a single Ceph kernel client [78] and two remote servers, with each server housing 16 OSDs (Ceph terminology for OS layer on the drive) for a total cluster of 32 OSDs. The client node runs Red Hat Enterprise Linux (RHEL) 9.4 (Linux kernel version 6.0.9-1) and has an Intel Sky Lake-E 28-core CPU and 256 GB GB of memory (6 memory channels per socket). The *iperf* network testing tool was used to verify network speed between the nodes, achieving a raw bandwidth of 9.8 Gb/s on the 10 GbE network used.

In contrast to the prior DeLiBA work, which focused almost exclusively on synthetic micro-benchmarks, this work also includes a realistic workload evaluated in cooperation with our industry partner's labs. These non-synthetic workloads (Online Analytical Processing (OLAP) [83] and Online Transaction Processing (OLTP) [84]) cover two real world applications and tasks that are part of a proprietary test suite regularly used by data center users in the industrial research lab. However, for reproducibility, we still include a synthetic workload generated by the Flexible I/O (*fio*) [85] tool. We use the notation *seq* and *rand* here to identify the sequential and random-access workloads, respectively.

We always measure *latency* and *throughput*. Following the methodology recently adopted by the Linux kernel community [86], we have continued to focus our measurement on larger block sizes, including 512 kB. This benefits applications for on-disk databases, particularly those in industry research labs involving full table scans or bulk data loads. However, the latency and throughput results in the figure show only 4 kB and 128 kB due to space constraints. To compare with earlier DeLiBA-2 (D2) work, we benchmark *both* erasure coding (EC) and replication operations, the two methods used in Ceph for data durability and high availability. The results reported here are based on averages measured across five benchmark executions to ensure accuracy and consistency.

2) ***Latency and Throughput (DeLiBA-K vs DeLiBA-2)****:* Figure 3a and Figure 4a shows the latency results of DeLiBA-K in replication and EC modes respectively. As mentioned previously, this comparison highlights the improvements in latency achieved in the DeLiBA-K framework compared to DeLiBA-2 (D2), across both Erasure Coding (EC) and Replication modes. In EC mode, DeLiBA-K significantly reduces latency, with random read latency at 4 kB dropping from 130μs to 85μs and random write latency at 4 kB decreasing from 98μs to 80μs. Similarly, sequential operations have also

TABLE I
COLUMN 2 AND 3 SHOWS THE SOFTWARE PROFILING RESULTS OF CEPH IN DELIBA-K. COLUMN 4 AND 5 SHOWS HARDWARE EMULATED ESTIMATES IN VIVADO. COLUMN 6 SHOWS HW EXECUTION TIMES ON REAL PHYSICAL FPGA (U280). COLUMN 7 AND COLUMN 8 SHOWS THE SOURCE LINE OF CODES FOR CEPH REPLICATION AND EC ALGORITHMS IN SOFTWARE (.C) AND HARDWARE (VERILOG)

Replication and EC Kernels	Profiling SW Execution Time (Ceph-kernel)	Overall contribution to runtime	Vivado 2024 RTL Cycles (min-max)	Vivado 2024 Latency (min-max)	HW Execution on FPGA	SLOCs (C) SW Ceph-kernel	SLOCs (Verilog) HW Ceph-kernel
Straw Bucket	55 μs	80 %	105-105	0.345 μs - 0.355 μs	49 μs	256	880
Straw2 Bucket	48 μs	80 %	155-155	0.315 μs - 0.315 μs	51 μs	256	806
List bucket	35 μs	80 %	40-40	0.161 μs - 0.161μs	56 μs	197	770
Tree Bucket	22 μs	85 %	130-130	0.115 μs - 0.115 μs	31 μs	241	780
Uniform Bucket	9 μs	72 %	40-50	0.180 μs - 0.180 μs	19 μs	237	745
Reed-Solomon Encoder	65 μs	70 %	150-150	0.345 μs - 0.345 μs	85 μs	280	960

seen notable improvements. In Replication mode, DeLiBA-K shows a substantial decrease in random read latency at $4\,\mathrm{kB}$, from 130μs to 85μs, and random write latency at $4\,\mathrm{kB}$ reduced from 98μs to 80μs. Figure 3b and Figure 4b shows the throughput results of DeLiBA-K in replication and EC modes, respectively. In EC mode, the random write throughput at $4\,\mathrm{kB}$ has increased by 2.88x, and random read throughput at $4\,\mathrm{kB}$ has increased by 2.4x.

IV. EXTENDING IN-NETWORK HARDWARE ARCHITECTURE IN DELIBA-K

As shown in Figure 2, the FPGA architecture in the DeLiBA-K framework has been extended to accommodate newly developed application side and Linux kernel libraries on the client (host) side (explained previously in Section III, Section III-A and Section III-B). The client (host) communicates with the XCU280-L2FSVH2892E AMD (earlier Xilinx) Alveo U280 UltraScale+ FPGA data center card via a PCIe Gen3x16 interface. In the new DeLiBA-K framework, we have explicitly focused on fully leveraging the microarchitecture of the AMD Alveo U280 FPGA to achieve maximum performance. To this end, we have utilized an additional number of vendor-specific FPGA tools and IPs (discussed in the following sections) in DeLiBA-K to achieve better results. The following sections explain the *four* critical optimizations in the FPGA architecture of previous DeLiBA-2 (D2) that were essential in developing the microarchitecture of DeLiBA-K, as illustrated in Figure 2. These four key optimizations, which represent significant extensions to the original design, are also highlighted by four black circles (❸,❹,❺, and ❻) in the main Figure 2. Note that the U280 card was chosen due to its high flexibility for R&D. For a deployment-at-scale, much simpler, and correspondingly cheaper, FPGA cards could be used instead.

After implementing the improved DeLiBA-K framework libraries on the client side and establishing a software baseline through system benchmarking, we extensively profiled the DeLiBA-K source code with the Ceph use case on a bare-metal server. To this end, Table I presents the profiling results alongside other findings, which will be discussed in the following section.

A. Leveraging QDMA in FPGA stack of DeLiBA-K

Since the UIFD kernel driver includes the Ceph RADOS block device (RBD) kernel module, and the software baseline established in the previous section is based on benchmarks conducted in the Ceph cluster, the customized Xilinx PCI Express Multi Queue DMA (QDMA) IP implementation in DeLiBA-K adheres to Ceph cluster-level rules defined in the CRUSH map. In Ceph storage, a CRUSH map is a set of rules defined by Ceph application to be enforced when clients read and write data in a cluster. For replication operations, the QDMA IP [79]–[81] is customized to incorporate rules for replicating data across remote nodes. This customization enables the replication hardware accelerator to process descriptors and perform the replication, thereby ensuring data availability across the network. Similarly, the hardware accelerator for erasure coding (EC) is configured to manage data chunks to enable data recovery in case of failure or data loss.

The Figure 2 illustrates the PCI Express (PCIe) QDMA architecture within the DeLiBA-K FPGA stack, which comprises *five* modules ((highlighted by circle ❸ in Figure 2): **R**equester **R**equest (RQ), **D**escriptor **E**ngine (DE), **H**ost-to-**C**ard (H2C), **C**ard-to-**H**ost (C2H), and **C**ompletion **E**ngine (CE).

Data communication among all modules is conducted via AXI-stream, as the DeLiBA-K FPGA stack functions as a data-plane smartNIC platform. In DeLiBA-K, the queues in QDMA have been individually configured according to their interface type, such as replication and erasure coding queues. By assigning these queues as resources to multiple PCIe Physical Functions (PFs) and Virtual Functions (VFs), a single QDMA core and PCI Express interface are utilized for both bare-metal and virtual machine (VM) applications in our industrial research lab.

In its current implementation, the customized QDMA IP supports up to 2048 queue sets for both replication and erasure coding computations. Each queue set consists of a collection of rings that operate together to manage specific DMA operations related to replication and erasure coding. Specifically, each of the 2048 queue sets in the QDMA includes a complete set of three rings: the H2C descriptor ring, the C2H descriptor ring, and the C2H completion ring. This configuration provides the high degree of parallelism required in DeLiBA-K which is

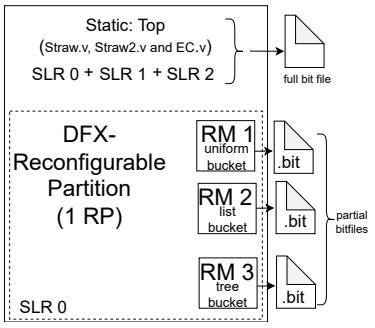


Fig. 5: DFX based Reconfiguration Modules (RMs)

crucial given the parallelism on the client-side kernel libraries.

For our framework, which utilizes 10G Ethernet and an x16 PCIe 3.0 interface, we have selected a 512-bit data bus width to ensure future-proofing and accommodate potential increases in data throughput. However, we have initially implemented a 256-bit data bus width, providing a burst size of 32 bytes per cycle, which meets the current 10G Ethernet requirements and aligns with the Ceph use case. This approach effectively balances performance and resource usage while enabling us to leverage higher bandwidth capabilities and a 64-byte burst size per cycle in the future.

The **RQ** module is responsible for receiving and managing incoming replication and erasure coding (EC) data packets. Whereas, the **DE** serves as a central module. The **DE** contains descriptors, which are data structures managed by the UIFD kernel driver. These descriptors define the *five* main parameters of a DMA operation for both replication and erasure coding: Source Address, Destination Address, Length of Replicated or Encoded Data, Control Information, and Next Descriptor Pointer (NDP). This implies that the descriptor itself does not carry the replication and EC *packet payload* but defines the parameters for the data transfer. In our case the descriptors are 128 bytes in size and it uses UltraRAM (also shown in Figure 2) to store per queue configuration. The total length of all descriptors is less than $64\,\mathrm{kB}$ in our implementation.

The **H2C** and **C2H** RTL modules primarily function as packet payload blocks (data blocks). The **H2C** descriptor rings are used to manage the flow of data from the UIFD driver to the hardware accelerators in DeLiBA-K. The **H2C** module can handle up to 256 read and write I/Os concurrently and includes a re-ordering buffer with a capacity of $32\,\mathrm{kB}$ of data. The **C2H** descriptor rings manage data transfers from the card back to the UIFD driver (client). Consequently, both **H2C** and **C2H** are written by the UIFD driver. **H2C** handles descriptors for DMA read operations from the host (client), while **C2H** handles descriptors for DMA write operations to the host (client).

B. New RTL Accelerators in FPGA stack of DeLiBA-K

As illustrated in Figure 2, the Descriptor Engine, H2C Streaming Engine, C2H Streaming Engine, and Completion Engine modules are equipped with bi-directional AXI-stream interfaces that connect to the replication and erasure coding hardware accelerators (highlighted by circle ❹ in Figure 2). A significant enhancement in DeLiBA-K is the complete redesign of these hardware accelerators from a High-Level Synthesis (HLS) based approach to a Hardware Description Language (HDL) implementation with a fine-granular control.

Although Bluespec [87] was initially considered, Verilog was ultimately selected due to its native compatibility with the user logic shell in QDMA IP. Consequently, the redesign adhered to the Verilog IEEE 1800-2023 standard [88] for cycle accurate Register Transfer Level (RTL) design. This decision was mainly motivated by the need for precise control over clock cycles required for the replication and erasure coding accelerators. Deterministic behavior was essential because these Verilog accelerators operate as finite state machines (FSMs), crucial for managing various states and transitions necessary for efficient data distribution in a Ceph storage cluster. The FSMs maintain their state within the FPGA fabric, utilizing internal memory resources such as Block RAMs (BRAMs) and UltraRAMs (URAMs) available on the AMD Alveo U280 FPGA. Comprehensive simulation and functional verification were conducted for each RTL accelerator to ensure their correctness and reliability. Given the inherent verbosity in HLD like Verilog, it was not an easy undertaking. However, we could explicitly improve two important metrics compared to our HLS accelerators in DeLiBA-2: *Latency* and *RTL* cycles (shown in columns 4 and 5 of Table I)

In the context of an FPGA hardware accelerator for the Ceph CRUSH replication algorithm, cycles refers to the number of clock cycles required to complete four key operations: rule evaluation, hash computation, data mapping, and replication. To define cycles, we measure how many clock cycles these four operations take within the Verilog RTL microarchitecture on the FPGA. Furthermore, latency in this context is the total time it takes for a data object to be fully processed, from its entry into the system to its replication. The optimization of the FPGA hardware accelerator for the Ceph CRUSH replication algorithm has resulted in an overall performance improvement of approximately 38.61% in terms of clock cycles.

The optimization of the FPGA hardware accelerator for the Ceph CRUSH replication algorithm has led to an overall latency reduction of approximately 45.71%, significantly enhancing the speed and efficiency of data processing operations. Both the replication and erasure coding RTL accelerators operate at approximately 235 MHz. The minimum packet length in DeLiBA-K is 64 bytes. In contrast, the maximum packet length is configurable to support the required MTU plus overhead, ranging from 1518 bytes for standard Ethernet to 9018 bytes for Jumbo frames, depending on the cluster's network requirements in the research lab.

C. Partial Reconfiguration in FPGA stack of DeLiBA-K

As shown in Figure 2, alongside the implementation of QDMA and newly redesigned RTL accelerators, another key optimization in DeLiBA-K framework is the utilization of

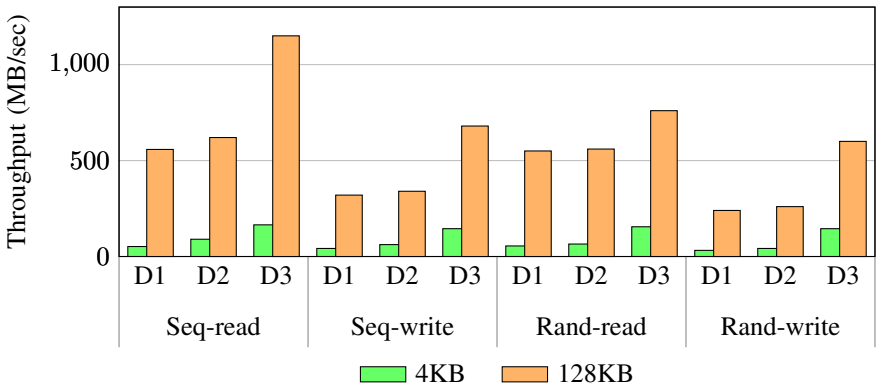


Fig. 6: Replication Mode: Hardware Accelerated I/O Throughput of DeLiBA-K (D3) in comparison to the previous DeLiBA versions i.e., DeLiBA-1 (D1) and DeLiBA-2 (D2)

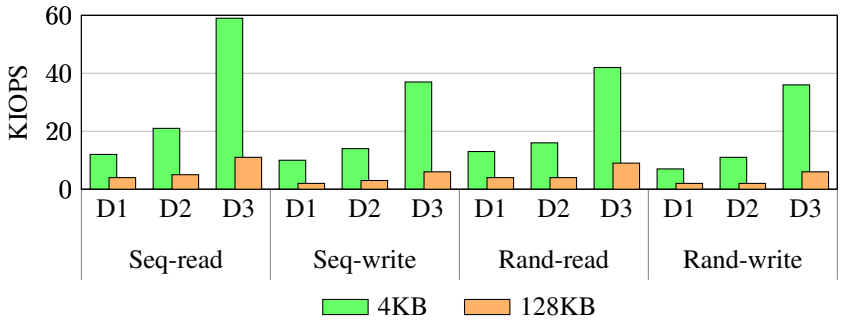


Fig. 7: Replication Mode: Hardware Accelerated KIOPS of DeLiBA-K (D3) in comparison to the previous DeLiBA versions i.e., DeLiBA-1 (D1) and DeLiBA-2 (D2)

FPGA-based partial reconfiguration (highlighted by circle ❺ in Figure 2). This optimization is crucial for addressing the dynamic nature of storage cluster hierarchies. For example, the size of the Ceph storage cluster may fluctuate due to the failure of underlying disks, which reduces the cluster size, or the addition of new disks, which increases the overall cluster size. This variation necessitates time-division multiplexing of the underlying FPGA resources. To accommodate these fluctuations, our redesigned RTL accelerators include three distinct in-network accelerators, each optimized for different cluster sizes. Previously, these accelerators were implemented within the static region of the Alveo U280 FPGA, following a single bitstream configuration, which made it impossible to change functionality on the fly without power cycling the storage server. Building upon this, the DeLiBA-K framework leverages dynamic reconfiguration of specific regions inside Alveo U280 FPGA [89]–[93] without reprogramming the entire FPGA in a live storage cluster, allowing for real-time adaptation to changing cluster sizes. To be precise, DeLiBA-K framework has leveraged one of the three underlying SLR (Super Logic Region) regions of the Alveo U280 FPGA through a technique termed as [4]Dynamic Function eXchange (DFX) [94]–[98]. As shown in main architecture in Figure 2, DeLiBA-K uses DFX through the **M**edia **C**onfiguration **A**ccess **P**ort (MCAP), which provides a dedicated connection to the configuration engine from one specific PCIe block per device. Each SLR in the FPGA is a device die slice that contains a subset of resources, such as CLBs, Block RAMs, DSP tiles, and GTs, structured similarly to non-SSI (stacked silicon interconnect) devices. In our implementation, the RTL accelerators, namely Straw, Straw2, and EC-encoder, are implemented in the static region, spanning across two SLRs (SLR1 and SLR2), to manage the primary, unchanging functions and ensure continuous FPGA operation under a single bitstream configuration. In contrast, dynamic regions (or reconfigurable regions) have to be positioned within a specific SLR based on available space and specific reconfiguration requirements. This ensures that global reset events are properly synchronized across all elements in the reconfiguration module (hereafter RM), and that all super long lines (SLL) are contained within the static portion of the design. Here the global reset events refer to signals that reset the entire FPGA or significant portions of it, including the reconfigurable parts. We selected SLR region 0 for our design, which includes 355K LUTs, 725K CLB registers, 490 Block RAM Tiles, 320 UltraRAM, and 2733 DSPs.

Figure 5 shows the schematic of a single *configuration* with the DFX-based terminologies. A configuration is a complete design that has at least one RM for each reconfigurable partition (hereafter RP). Each configuration generates one full

[4]DFX is AMD's terminology for the methodology that includes partial reconfiguration in FPGA designs

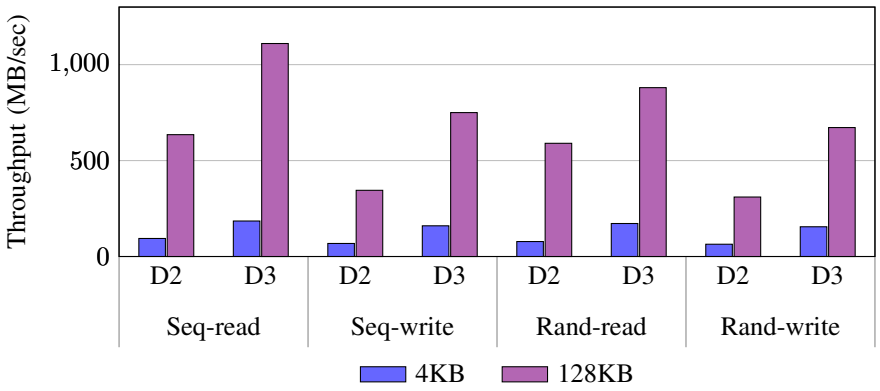


Fig. 8: Erasure Coding (EC) mode: Hardware Accelerated I/O Throughput of DeLiBA-K (D3) in comparison to the previous DeLiBA-2 (D2) version

binary (BIT) file as well as one partial BIT file for each RM. An RP is a main reconfigurable block and it defines a level of hierarchy within which different RMs are implemented. An RM is the netlist implemented within an RP, and multiple RMs can exist for a single RP. As shown in Figure 5, within SLR region 0, we developed a single RP containing three RMs. Each RM generates a partial bitstream and must be carefully floorplanned with physical constraints (Pblocks) to ensure the module can be physically isolated and meet timing requirements. These partial bitstreams are tailored to specific use cases: the Uniform Bucket RTL accelerator is ideal for scenarios where all devices in the storage cluster have identical capacities and performance characteristics, making it ideal for uniform hardware configurations. The List Bucket RTL accelerator's partial bitstream is optimized for expanding clusters where devices are frequently added, effectively managing dynamic environments by seamlessly integrating new devices. Lastly, the Tree Bucket RTL accelerator, which uses a binary search tree structure, is best suited for larger or more complex clusters that require efficient management of many devices or nested buckets. Synthesizing each of these accelerators as RM (partially reconfigurable designs) is based on typical a *bottom-up* synthesis flow in which each module has its own synthesis project. The DFX Configuration Analysis report compares each RM and examines its resource usage, floorplanning, clocking, and timing metrics. Also, we have verified all DFX designs by running a verification utility `pr_verify` on all our configurations.

D. Redesigning TX and RX paths of TCP/IP in RTL

In the previous version of DeLiBA, the storage accelerators (replication and erasure) relied on a High-Level Synthesis (HLS)-based communication library and a HLS-based open-source TCP/IP block. In DeLiBA-K, we have completely remove the HLS-based communication library and TCP/IP stack. To enhance performance, the RX and TX modules, along with the storage accelerators, have also been redesigned in Verilog (highlighted by circle ❻ in Figure 2), addressing the performance limitations inherent in the HLS-based design. Furthermore, the CMAC in DeLiBA-K operates at a frequency of 260 MHz.

V. Hardware Evaluation and Comparison with Previous Versions

The hardware testbed deployed at the industry lab includes a XCU280-L2FSVH2892E 16nm Ultrascale Xilinx Alveo U280 FPGA card attached to the client node. The client node runs Red Hat Enterprise Linux (RHEL) 9.4 (Linux kernel version 6.0.9-1) and has an Intel Sky Lake-E 28-core CPU and 256 GB of memory (6 memory channels per socket).

In line with the software baseline, all hardware evaluations in the research lab were conducted thoroughly using both synthetic and real-world workloads. In this context, real-world workloads refer to actual applications and tasks that are part of a test suite regularly utilized by data center users in our industrial partner's research lab.

a) ***Criteria of Hardware Evaluation****:* In this final FPGA-based hardware evaluation, we compared DeLiBA-K with the previous DeLiBA frameworks, namely DeLiBA-1 (D1) and DeLiBA-2 (D2). In the bar charts (Figure 6, Figure 7, Figure 8 and Figure 9), D1 refers to DeLiBA-1, and D2 refers to DeLiBA-2. We benchmarked both erasure coding (EC) and replication operations. It's important to note that, unlike DeLiBA-2 (D2), DeLiBA-1 (D1) did not include erasure coding accelerators. Therefore, the final FPGA based DeLiBA-K erasure coding results are compared only to DeLiBA-2 (D2) erasure coding results.

b) ***Throughput and Latency****:* The DeLiBA-K FPGA framework demonstrates substantial throughput improvements over DeLiBA-2, particularly in random writes and sequential writes. For random writes, DeLiBA-K achieves the most significant gains at smaller block sizes, with throughput reaching $145\,\mathrm{MB/s}$ at $4\,\mathrm{kB}$ and $170\,\mathrm{MB/s}$ at $8\,\mathrm{kB}$, representing speed-ups of 3.45x and 2.50x, respectively. In sequential writes, notable improvements are also seen, with DeLiBA-K achieving $440\,\mathrm{MB/s}$ at $64\,\mathrm{kB}$ and $680\,\mathrm{MB/s}$ at $128\,\mathrm{kB}$, translating to speed-ups of 2.38x and 2.00x, respectively.

Table II presents the end-to-end latency measurements in DeLiBA-K compared to previous versions of DeLiBA. Since

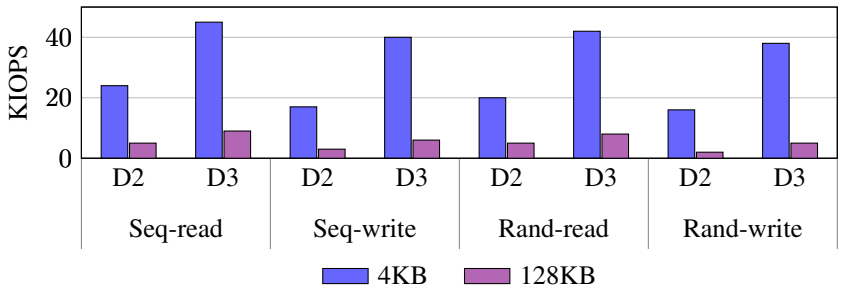


Fig. 9: Erasure Coding (EC) mode: Hardware Accelerated KIOPS of DeLiBA-K (D3) in comparison to the previous DeLiBA-2 (D2) version

TABLE II
I/O REQUEST LATENCY IN DELIBA-K (HARDWARE) COMPARED WITH PREVIOUS FRAMEWORKS DELIBA-1 (D1) AND DELIBA-2 (D2)

Hardware (Replication) (4 kB)	**Latency [µs]**			
	seq-read	***seq-write***	***rand-read***	***rand-write***
DeLiBA-1	65	95	130	98
DeLiBA-2	55	75	85	82
DeLiBA-K	40	52	64	68
Hardware (Erasure Coding) (4 kB)	**Latency [µs]**			
	seq-read	***seq-write***	***rand-read***	***rand-write***
DeLiBA-2	48	70	82	75
DeLiBA-K	38	47	59	60

TABLE III
TOTAL RESOURCE UTILIZATION OF DELIBA-K, INCLUDING THE 10G TCP/IP STACK, QDMA, RTL ACCELERATORS CONFIGURED FOR A SPECIFIC REPLICATION AND EC MODE. RESOURCE PERCENTAGES ARE RELATIVE TO AVAILABLE RESOURCES ON THE ALVEO U280 FPGA.

RTL Kernel + RTL TCP/IP + CMAC + QDMA	**CLB LUTs**		**CLB Registers**		**Block RAM (BRAM)**		**UltraRAM (URAM)**		**DSPs**
	Count	**% Usage**	**Count**	**% Usage**	**Count**	**% Usage**	**Count**	**% Usage**	**Count**
Straw Bucket	78,555	6.2 %	224K	8.59 %	190	9.42 %	26	2.71 %	0
Straw2 Bucket	82,334	6.31%	313K	12.01 %	165	8.18 %	35	3.65 %	0
Reed-Solomon Encoder	92,355	7.08 %	582K	22.32 %	215	10.66 %	52	5.42 %	0
Partial Reconfiguration Modules (RM) in SLR 0 of U280	**CLB LUTs**		**CLB Registers**		**Block RAM (BRAM)**		**UltraRAM (URAM)**		**DSPs**
	Count	**% Usage**	**Count**	**% Usage**	**Count**	**% Usage**	**Count**	**% Usage**	**Count**
RM 1 List Bucket (Replication RTL Accelerator)	52,335	14.74 %	92,456	12.75 %	85	17.35 %	22	6.88 %	0
RM 2 Tree (Replication RTL Accelerator)	56,555	15.93 %	97,523	13.45 %	82	16.73 %	26	8.13 %	0
RM 3 Uniform (Replication RTL Accelerator)	62,456	17.59 %	112K	15.45 %	78	15.92 %	29	8.7%	0

DeLiBA-1 framework did not include Erasure Coding hardware accelerators, the erasure coding results of DeLiBA-K in Table II are only compared with those of DeLiBA-2. In terms of latency, DeLiBA-K reduces random read latency from 85 μs to 64 μs, a 25% decrease compared to its predecessor DeLiBA-2. Similarly, random write latency drops from 82 μs to 68 μs, representing a reduction of reduction of approximately 17%.

*c) **Resource Utilization and Power Measurements**:* Table III shows the place-and-routed resource utilization. The XCU280-L2FSVH2892E AMD Alveo U280 card, built on 16nm UltraScale architecture, features an FPGA chip with 1.3 million LUTs, 2.72 million registers, 9,024 DSP slices, 2,016 Block RAMs (BRAMs), and 960 UltraRAMs (URAMs), offering 4.5 MB of on-chip BRAM and 30 MB of on-chip URAM. Furthermore, Alveo U280 is divided into three Super Logic Regions (SLRs). For DFX based partial reconfiguration of three accelerators we have considered SLR region 0 of the U280 chip. The SLR region 0 consists of SLR 355K LUTs, 725K CLB register, 490 Block RAM Tile, 320 UltraRAM, and 2733 DSPs. The Reconfigurable Modules (RMs) include accelerators like Uniform, List, and Tree in SLR 0. Furthermore, we dedicated significant effort to conducting detailed power measurements. This focus was essential because full-scale power measurements were a critical requirement established by our industrial partner. We categorized the power consumption into two scenarios: full load with no partial reconfiguration and full load with partial reconfiguration. We conducted detailed power estimation measurements using Vivado Report Power and Vivado Power Analysis. We then performed final power measurements using Xilinx xbutil [99] and xbtest [100]. Options like Power Design Manager (PDM) [101] is available; however, to the best of our knowledge, PDM is not applicable to Alveo U280s. In the first scenario, the reported power consumption was approximately 195 watts. With partial reconfiguration, the reported power consumption was 170 watts.

VI. RELATED WORK

In the development of our framework, we dedicated considerable effort to thoroughly understand existing research and

identify gaps where our framework can add value. While it is understood that the some of the related works differ in scope and approach, we have ensured that, wherever applicable, we compare benchmark results that are directly relevant to our findings. To this end, we have classified the related work into three categories: (**1**) FPGA-based I/O frameworks that emphasize OS integration; (**2**) relevant framework utilizing the io_uring API; and **(3)** in-network distributed frameworks, particularly those based on Ceph, alongside industrial accelerators that have advanced research on Ceph into practical, industry-grade solutions.

FPGA-based I/O frameworks: Landgraf *et al.* [102] introduced FSRF (File System for Reconfigurable Fabrics), a framework demonstrating that FPGA I/O can be abstracted through mmap-style file access. Their microbenchmark measures transfer performance for I/O sizes from 4KiB to 2MiB using a random access pattern from a single application. It shows how quickly FSRF can handle warm and cold data, and how efficiently the host reads from the NVMe drive and transfers data to the FPGA via DMA. However, these results are not directly comparable, as DeLiBA-K does not support mmap-style virtual memory on FPGAs.

Korolija *et al.* [15] explored whether traditional operating system abstractions are suitable for FPGAs within hybrid systems. To investigate this, the authors developed a FPGA based framework namely Coyote that integrates with the host OS and offer a comprehensive set of OS abstractions. While our work does not adopt a holistic approach to all operating system abstractions as Coyote does, we consider our work could potentially be integrated with Coyote for a more robust AIO interface.

Ruan *et al.* [21] developed INSIDER, an FPGA-based reconfigurable drive controller as the in-storage computing (ISC) unit based on traditional Linux APIs. Although INSIDER mentions the implementation of partial reconfiguration in its FPGA framework, the open-source version for AWS F1 FPGAs does not support partial reconfiguration. In one of the final evaluations the work reports a latency of 5μs. However, this specific latency is associated with the local in-storage computing read and write I/Os facilitated by their FPGA-based local reconfigurable drive controller. This highlights a fundamental architectural difference between INSIDER and our work: INSIDER is drive-centric and does not account for network I/O, thereby limiting its applicability in distributed storage systems.

io_uring framework: Zhou *et al.* [103] developed a framework to mitigate the high latency and low throughput of Paxos replication protocol by using io_uring to accelerate the consensus process. This is relevant to our DeLiBA-K framework, as we also leverage io_uring to enhance the performance of Ceph-based replication protocols. The work reports a 99th-percentile tail latency of 49μs, compared to our 40μs, though their maximum throughput reaches 65K IOPS, slightly higher than our 59K IOPS.

Ceph: DeLiBA-K is a cloud-driven block storage stack, with Ceph traditionally used in cloud environments. However, I/O performance issues are also prominent in HPC. Gai *et al.* [104] introduced UrsaX, a block storage service for the next-generation Tianhe exascale supercomputer, comparing it to Ceph in HPC settings. While the framework lacks hardware accelerators like FPGA, its performance metrics still offer meaningful comparisons to our framework. For 4KB random read and write I/Os, UrsaX achieves a latency of under 100μs compared to our latency of 59μs.

Since the development of DeLiBA-K has been conducted in collaboration with our industrial partner, it is important to compare our framework with those resulting from industrial research. A commercial Ceph accelerator, known as the Accepherator [105], has been developed to accelerate erasure coding performance in Ceph. Similar to DeLiBA-K, the accelerator features a 10GbE SFP network interface and utilizes a hardware acceleration I/O module with programmable silicon to compute erasure coding. Architecturally, our framework excels in two key areas: First, DeLiBA-K is tightly integrated with the latest Ceph version and is designed to scale with future releases. Second, on the FPGA side, our framework outperforms Accepherator by implementing both erasure coding and replication accelerators, along with an FPGA-based TCP/IP network stack. AMD [106] presents a solution to optimize Ceph block device operations by leveraging Data Processing Units (DPUs) This approach enables the DPU to run Ceph client libraries (library rados block device), effectively virtualizing the Ceph block device and presenting it to the host as a PCIe-connected NVMe disk. Intel [107] has implemented hardware-based compression for Ceph using Intel's QuickAssist Technology (QAT).

VII. Conclusion and Future Work

The development of DeLiBA-K has effectively addressed the limitations identified in DeLiBA-2, particularly by eliminating unnecessary context switches. Furthermore, the improvements in the FPGA network and storage stack within DeLiBA-K have contributed substantially to this performance enhancement. While developing the previous two versions of DeLiBA, and especially the current DeLiBA-K framework, we extensively focused on tracing Ceph and Linux kernel operations related to erasure coding. However, a detailed explanation of the profiling and tracing-related work falls outside the scope of this paper. In future work, we will provide a detailed explanation of the techniques that were used to profile and trace these erasure coding operations.

DeLiBA-K will be soon open-sourced at https://github.com/esa-tu-darmstadt/deliba

Acknowledgment

This work has been co-funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01 IS 19018 B. We would also like to thanks Amazon Research for research credits. We would also like to thank our industry partner, a global leader in DBMSs and cloud operations, who unfortunately cannot be named due to confidentiality reasons.

References

[1] J. R. David Reinsel, John Gantz, "The digitization of the world: From edge to core. data age 2025," https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf,, 2018, (Last accessed: 2024-03-01).

[2] A. M. Deiana, N. Tran, J. Agar, M. Blott, G. Di Guglielmo, J. Duarte, P. Harris, S. Hauck, M. Liu, M. S. Neubauer, J. Ngadiuba, S. Ogrenci-Memik, M. Pierini, T. Aarrestad, S. Bähr, J. Becker, A.-S. Berthold, R. J. Bonventre, T. E. Müller Bravo, M. Diefenthaler, Z. Dong, N. Fritzsche, A. Gholami, E. Govorkova, D. Guo, K. J. Hazelwood, C. Herwig, B. Khan, S. Kim, T. Klijnsma, Y. Liu, K. H. Lo, T. Nguyen, G. Pezzullo, S. Rasoulinezhad, R. A. Rivera, K. Scholberg, J. Selig, S. Sen, D. Strukov, W. Tang, S. Thais, K. L. Unger, R. Vilalta, B. von Krosigk, S. Wang, and T. K. Warburton, "Applications and techniques for fast machine learning in science," *Frontiers in Big Data*, vol. 5, Apr. 2022. [Online]. Available: http://dx.doi.org/10.3389/fdata.2022.787421

[3] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves, "Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications," *ACM Trans. Storage*, vol. 16, no. 4, oct 2020. [Online]. Available: https://doi.org/10.1145/3415580

[4] J. L. Bez, S. Byna, and S. Ibrahim, "I/o access patterns in hpc applications: A 360-degree survey," *ACM Comput. Surv.*, vol. 56, no. 2, sep 2023. [Online]. Available: https://doi.org/10.1145/3611007

[5] B. Khan, C. Heinz, and A. Koch, "Deliba: An open-source hardware/-software framework for the development of linux block i/o accelerators," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 183–191.

[6] ——, "The open-source deliba2 hardware/software framework for distributed storage accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 2, mar 2024. [Online]. Available: https://doi.org/10.1145/3624482

[7] G. Haas and V. Leis, "What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines," *Proc. VLDB Endow.*, vol. 16, no. 9, p. 2090–2102, may 2023. [Online]. Available: https://doi.org/10.14778/3598581.3598584

[8] Z. Ren and A. Trivedi, "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*, ser. CHEOPS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 35–45. [Online]. Available: https://doi.org/10.1145/3578353.3589545

[9] Jens Axboe, "Efficient io with io_uring," *https://kernel.dk/io_uring.pdf*, Oct. 2019, accessed: 2024-03-04. [Online]. Available: https://kernel.dk/io_uring.pdf

[10] J. Corbet, "Redesigned workqueues for io_uring," Oct. 2019, accessed: 2024-07-03. [Online]. Available: https://lwn.net/Articles/803070/

[11] ——, "Automatic buffer selection for io_uring," Mar. 2020, accessed: 2024-07-03. [Online]. Available: https://lwn.net/Articles/815491/

[12] J. Axboe, "Liburing," 2019, accessed: 2024-07-03. [Online]. Available: https://github.com/axboe/liburing

[13] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: an open, general, cpu/fpga platform for systems software research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 434–451. [Online]. Available: https://doi.org/10.1145/3503222.3507742

[14] L. Y. Wong, J. Zhang, and J. J. Li, "Dongle: Direct fpga-orchestrated nvme storage for hls," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3–13. [Online]. Available: https://doi.org/10.1145/3543622.3573185

[15] D. Korolija, T. Roscoe, and G. Alonso, "Do os abstractions make sense on fpgas?" in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[16] M. Jung, "OpenExpress: Fully hardware automated open research framework for future fast NVMe devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 649–656. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/jung

[17] S. Awamoto, E. Focht, and M. Honda, "Designing a storage software stack for accelerators," in *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage '20. USA: USENIX Association, 2020.

[18] R. Schmid, M. Plauth, L. Wenzel, F. Eberhardt, and A. Polze, "Accessible near-storage computing with fpgas," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387557

[19] ——, "Orchestrating near-data fpga accelerators using unix pipes," in *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*, 2019, pp. 125–128.

[20] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda, "The feniks fpga operating system for cloud computing," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3124680.3124743

[21] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing In-Storage computing system for emerging High-Performance drive," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/ruan

[22] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[23] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 107–127. [Online]. Available: http://www.usenix.org/conference/osdi18/presentation/khawaja

[24] K. Nam, B. Fort, and S. Brown, "Fish: Linux system calls for fpga accelerators," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[25] S. Guha, W. Wang, S. Ibraheem, M. Balakrishnan, and J. Szefer, "Design and implementation of open-source sata iii core for stratix v fpgas," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 237–240.

[26] J. Ahn, D. Kwon, Y. Kim, M. Ajdari, J. Lee, and J. Kim, "Dcs: a fast and scalable device-centric server architecture," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 559–571. [Online]. Available: https://doi.org/10.1145/2830772.2830794

[27] A. Kaitoua, H. Hajj, M. A. R. Saghir, H. Artail, H. Akkary, M. Awad, M. Sharafeddine, and K. Mershad, "Hadoop extensions for distributed computing on reconfigurable active ssd clusters," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, jul 2014. [Online]. Available: https://doi.org/10.1145/2608199

[28] C. Gorman, P. Siqueira, and R. Tessier, "An open-source sata core for virtex-4 fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 454–457.

[29] P. Lehmann, T. Frank, O. Knodel, S. Köhler, T. B. Preußer, and R. G. Spallek, "Weasel: A platform-independent streaming-optimized sata controller," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–4.

[30] J. Zhang, X. Jiang, J. Wu, and J. Shan, "Netstoragefpga—a prototyping platform for building high-performance transmission and storage systems using field programmable gate array (fpga) hardware," in *2014 19th IEEE-NPSS Real Time Conference*, 2014, pp. 1–3.

[31] A. A. Mendon, B. Huang, and R. Sass, "A high performance, open source sata2 core," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 421–428.

[32] L. Woods and K. Eguro, "Groundhog - a serial ata host bus adapter (hba) for fpgas," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 220–223.

[33] A. Ismail and L. Shannon, "Fuse: Front-end user framework for o/s abstraction of hardware accelerators," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 170–177.

[34] A. Kivity, "I/o access methods in scylla," Oct. 2017, accessed: 2024-07-03. [Online]. Available: https://www.scylladb.com/2017/10/05/io-access-methods-scylla/

[35] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use mmap in your database management system?" in *CIDR 2022, Conference on Innovative Data Systems Research*, 2022.

[36] J. Choi, J. Kim, and H. Han, "Efficient memory mapped file I/O for In-Memory file systems," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, Jul. 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/choi

[37] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped I/O for fast storage devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 813–827. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/papagiannis

[38] N. Y. Song, Y. J. Yu, W. Shin, H. Eom, and H. Y. Yeom, "Low-latency memory-mapped i/o for data-intensive applications on fast storage devices," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 766–770.

[39] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "Dimmap—a scalable memory-map runtime for out-of-core data-intensive applications," *Cluster Computing*, vol. 18, pp. 15–28, 2015.

[40] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004, vol. 352.

[41] 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 *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 603–616.

[42] B. C. LaHaise, "An aio implementation and its behaviour," in *Ottawa Linux Symposium*, 2002, p. 260.

[43] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous i/o support in linux 2.5," in *Proceedings of the Linux Symposium*. Citeseer, 2003, pp. 371–386.

[44] S. Bhattacharya, J. Tran, and M. Sullivan, "Linux aio performance and robustness for enterprise workloads," in *Proceedings of the Linux Symposium*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:196418937

[45] Z. Brown, "Asynchronous system calls," in *Proceedings of the Ottawa Linux Symposium (OLS)*. Citeseer, 2007, pp. 81–85.

[46] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting {Quasi-Asynchronous}{I/O} for better responsiveness in mobile devices," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 191–202.

[47] C. Hall, B. Mortensen, P. Bonnet, H. Tuuri, and P. Zaitsev, *Do Linux Asynchronous I/0 Really Matters?* DIKU, 2004.

[48] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, p. 387–400. [Online]. Available: https://doi.org/10.1145/2150976.2151017

[49] 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 *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'16. USA: USENIX Association, 2016, p. 41–45.

[50] 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)*, 2017, pp. 154–161.

[51] SPDK, "Spdk source code," https://github.com/spdk/spdk, (Last accessed: 2021-12-16).

[52] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[53] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the sync," *ACM Trans. Comput. Syst.*, vol. 26, no. 3, sep 2008. [Online]. Available: https://doi.org/10.1145/1394441.1394442

[54] D. Seo, Y. Joo, and N. Dutt, "Improving virtualized i/o performance by expanding the polled i/o path of linux," in *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 31–37. [Online]. Available: https://doi.org/10.1145/3655038.3665944

[55] J. Lee, G. Oh, and S.-W. Lee, "Boosting compaction in b-tree based key-value store by exploiting parallel reads in flash ssds," *IEEE Access*, vol. 9, pp. 56 344–56 353, 2021.

[56] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "Posix abstractions in modern operating systems: The old, the new, and the missing," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901350

[57] E. Zadok, D. Hildebrand, G. Kuenning, and K. A. Smith, "POSIX is dead! long live... errr... what exactly?" in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, Jul. 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/zadok

[58] D. McCracken, "Posix threads and the linux kernel," in *Ottawa Linux Symposium*, 2002, p. 330.

[59] S. R. Walli, "The posix family of standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[60] Stack Overflow, "Difference between posix aio and libaio on linux," Dec. 2011, accessed: 2024-07-03. [Online]. Available: https://stackoverflow.com/questions/8768083/difference-between-posix-aio-and-libaio-on-linux

[61] A. Aghayev, M. Shafaei, and P. Desnoyers, "Skylight—a window on shingled disk operation," *ACM Transactions on Storage (TOS)*, vol. 11, no. 4, pp. 1–28, 2015.

[62] Y. Cassuto, M. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic, "Indirection systems for shingled-recording disk drives," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–14.

[63] M. H. Kryder, E. C. Gage, T. W. McDaniel, W. A. Challener, R. E. Rottmayer, G. Ju, Y.-T. Hsia, and M. F. Erden, "Heat assisted magnetic recording," *Proceedings of the IEEE*, vol. 96, no. 11, pp. 1810–1835, 2008.

[64] N. Tehrany and A. Trivedi, "Understanding nvme zoned namespace (zns) flash ssd storage devices," *arXiv preprint arXiv:2206.01547*, 2022.

[65] K. Han, H. Gwak, D. Shin, and J. Hwang, "Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 147–162.

[66] T. Stavrinos, D. S. Berger, E. Katz-Bassett, and W. Lloyd, "Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 144–151.

[67] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: Introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*, ser. SYSTOR '13. New York, NY, USA: Association for Computing Machinery, 2013.

[68] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, pp. 1–70, 2014.

[69] J. Liedtke, "On micro-kernel construction," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 237–250, 1995.

[70] K. Dang Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell, "Microkernel hypervisor for a hybrid arm-fpga platform," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013, pp. 219–226.

[71] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?" *Queue*, vol. 11, no. 11, pp. 30–44, 2013.

[72] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library op-

erating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.
[73] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 291–304.
[74] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "The case for custom storage backends in distributed storage systems," *ACM Trans. Storage*, vol. 16, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3386362
[75] C. storage, "Ceph)," https://github.com/ceph/ceph, (Last accessed: 2021-12-16).
[76] Ceph, "block device rbd kernel driver," https://github.com/torvalds/linux/blob/master/drivers/block/rbd.c, (Last accessed: 2021-12-16).
[77] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 31–31.
[78] Ceph, "Ceph kernel client (kernel modules)," 2024. [Online]. Available: https://github.com/ceph/ceph-client
[79] A. (Xilinx), *QDMA Subsystem for PCI Express: Product Guide*, https://www.xilinx.com/products/intellectual-property/qdma.html, AMD (Xilinx), San Jose, CA, 2024, https://www.xilinx.com/support/documentation/ip_documentation/qdma.pdf.
[80] X. (now AMD), "Amd opennic project," https://github.com/Xilinx/open-nic, 2021, (Last accessed: 2024-08-5).
[81] ——, "Qdma linux drivers," https://github.com/Xilinx/dma_ip_drivers/tree/master/QDMA, 2021, (Last accessed: 2024-08-5).
[82] M. H. Hajkazemi, V. Aschenbrenner, M. Abdi, E. U. Kaynar, A. Mossayebzadeh, O. Krieger, and P. Desnoyers, "Beating the i/o bottleneck: a case for log-structured virtual disks," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 628–643. [Online]. Available: https://doi.org/10.1145/3492321.3524271
[83] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *SIGMOD Rec.*, vol. 26, no. 1, p. 65–74, mar 1997. [Online]. Available: https://doi.org/10.1145/248603.248616
[84] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 981–992. [Online]. Available: https://doi.org/10.1145/1376616.1376713
[85] FIO, "Fio tool source code," https://github.com/axboe/fio, (Last accessed: 2021-12-16).
[86] J. Edge, "Moving the kernel to large block sizes," *LWN.net*, September 2023, accessed: June 29, 2024. [Online]. Available: https://lwn.net/Articles/945646/
[87] B. Inc, "Bluespec compiler," https://github.com/B-Lang-org/bsc, (Last accessed: 2024-08-5).
[88] IEEE, "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pp. 1–1354, 2024.
[89] A. Xilinx, "Alveo u280 data center accelerator card data sheet (ds963)," https://docs.xilinx.com/r/en-US/ds963-u280/Alveo-Product-Details, 2023, (Last accessed: 2024-08-5).
[90] Xilinx, "Alveo u280 data center accelerator card user guide ug1314 (v1.2.1) november 20, 2019," https://www.mouser.com/pdfDocs/u280userguide.pdf, 2019, (Last accessed: 2024-08-5).
[91] A. Xilinx, "Alveo data center accelerator card platforms user guide ug1120 (v2.0.1) october 11, 2023," https://docs.amd.com/r/en-US/ug1120-alveo-platforms, 2023-10-11, (Last accessed: 2024-08-5).
[92] ——, "Getting started with alveo data center accelerator cards," GettingStartedwithAlveoDataCenterAcceleratorCardsUserGuide(UG1301), 2022-12-23, (Last accessed: 2024-08-5).
[93] ——, "Alveo data center accelerator card test user guide ug1361 (v5.0) february 11, 2021," https://docs.amd.com/v/u/en-US/ug1361-alveo-card-validation-test-solution, 2023-10-11, (Last accessed: 2024-08-5).
[94] ——, "Vivado design suite user guide: Dynamic function exchange (ug909)," https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration, 2024-06-12, (Last accessed: 2024-08-5).
[95] AMD, "Vivado design suite tutorial dynamic function exchange ug947 (v2024.1) june 12, 2024," https://docs.amd.com/r/en-US/ug947-vivado-partial-reconfiguration-tutorial, 2024-06-12, (Last accessed: 2024-08-5).
[96] ——, "Technology advancements for dynamic function exchange in vivado ml edition wp534 (v1.0) july 28, 2021," https://docs.amd.com/v/u/en-US/wp534-technology-advancements-dfx, 2021, (Last accessed: 2024-08-5).
[97] X. (now AMD), "Fast partial reconfiguration over pci express xapp1338 (v1.0)," https://docs.amd.com/r/en-US/xapp1338-fast-partial-reconfiguration-pci-express, 2019, (Last accessed: 2024-08-5).
[98] ——, "Isolation design flow + dynamic function exchange example," https://docs.amd.com/r/en-US/xapp1338-fast-partial-reconfiguration-pci-express, 2019, (Last accessed: 2024-08-5).
[99] X. Inc., *xbutil User Guide*, https://www.xilinx.com/products/boards-and-kits/xbutil.html, Xilinx Inc., San Jose, CA, July 2024, https://www.xilinx.com/support/documentation/sw_manuals/xbutil.pdf.
[100] X. (now AMD), "Alveo card validation test (xbtest)," https://www.xilinx.com/products/acceleration-solutions/xbtest.html#u280, (Last accessed: 2024-08-5).
[101] A. (Xilinx), *Power Design Manager (PDM) User Guide*, https://www.xilinx.com/products/design-tools/power-design-manager.html, AMD (Xilinx), San Jose, CA, 2024, https://www.xilinx.com/support/documentation/sw_manuals/pdm.pdf.
[102] J. Landgraf, M. Giordano, E. Yoon, and C. J. Rossbach, "Reconfigurable virtual memory for fpga-driven i/o," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 556–571. [Online]. Available: https://doi.org/10.1145/3582016.3582048
[103] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating distributed protocols with eBPF," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1391–1407. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/zhou
[104] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong, "Ursa: Hybrid block storage for cloud-scale virtual disks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303967
[105] SoftIron, "Softiron unveils hardware-based ceph erasure coding acceleration," https://softiron.com/blog/softiron-unveils-hardware-based-ceph-erasure-coding-acceleration/, (gesucht und gelesen: 2023-05-30).
[106] V. Kari, "Ceph block device optimization using dpus," https://ceph2023.sched.com, (gesucht und gelesen: 2023-05-30).
[107] H. FENG, "Offloading compression and encryption in ceph using intel® quickassist technology," https://www.intel.com/content/www/us/en/developer/articles/technical/offloading-compression-and-encryption-in-ceph.html, (Last accessed: 2024-08-5).