The Open-Source DeLiBA2 Hardware/Software Framework for Distributed Storage Accelerators

BABAR KHAN, CARSTEN HEINZ, and ANDREAS KOCH, Embedded Systems and Applications Group, Technische Universität Darmstadt, Germany

With the trend towards ever larger "big data" applications, many of the gains achievable by using specialized compute accelerators become diminished due to the growing I/O overheads. While there have been several research efforts into computational storage and FPGA implementations of the NVMe interface, to our knowledge there have been only very limited efforts to move *larger* parts of the Linux block I/O stack into FPGA-based hardware accelerators. Our hardware/software framework DeLiBA initially addressed this deficiency by allowing high-productivity development of software components of the I/O stack in user instead of kernel space and leverages a proven FPGA SoC framework to quickly compose and deploy the actual FPGA-based I/O accelerators. In its initial form, it achieves 10% higher throughput and up to 2.3× the I/Os per second (IOPS) for a proof-of-concept Ceph accelerator running in a real multi-node Ceph cluster. In DeLiBA2, we have extended the framework further to better support *distributed* storage systems, specifically by directly integrating the block I/O accelerators with a hardware-accelerated network stack, as well as by accelerating more storage functions. With these improvements, performance grows significantly: The cluster-level speed-ups now reach up to 2.8× for *both* throughput and IOPS relative to Ceph in software in synthetic benchmarks, and achieve end-to-end wall clock speed-ups of 20% for the real workload of building a large software package.

CCS Concepts: • **Hardware → Hardware-software codesign**.

Additional Key Words and Phrases: High-Level Synthesis, FPGA architecture, FPGA acceleration, Linux, Application and Architecture, Programming Tools, Open-Source

ACM Reference Format:
Babar Khan, Carsten Heinz, and Andreas Koch. 2022. The Open-Source DeLiBA2 Hardware/Software Framework for Distributed Storage Accelerators. *ACM Trans. Reconfig. Technol. Syst.* 37, 4, Article 111 (August 2022), 32 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the trend towards ever larger "big data" applications, many of the gains achievable by using specialized compute accelerators become diminished due to the growing I/O overheads. Often, the required storage capacities can only be realized by distributed storage clusters, disaggregated from the compute clusters. Such systems include traditional SANs [77] for block storage, but also highly scalable parallel file systems such as HDFS [63], GPFS [31], PVFS [12] and PanFS [51]. Some storage systems, such as the Ceph [1, 14, 67] solution examined later in this work, combine different storage approaches, such as file storage, block storage, and object storage in a single system. But as the protocols for interacting with these systems become ever more complex as well, e.g., to address fault tolerance and highly parallel operations, many opportunities to employ hardware acceleration

Authors' address: Babar Khan; Carsten Heinz; Andreas Koch, {khan,heinz,koch}@esa.tu-darmstadt.de, Embedded Systems and Applications Group, Technische Universität Darmstadt, Hochschulstr. 10, Darmstadt, Hessen, Germany, 64289.

1936-7406/2022/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

in the storage I/O stack become apparent. This follows a development similar to the increased use of acceleration (offload) functions for high-speed networking.

However, only limited prior work has been performed in this area (see Section 7 for a discussion). And new efforts are hampered both by the complexity of the existing solutions, e.g., the Linux block storage stack with its more than 64K lines of code, as well as the challenging development environment: Many storage stacks are implemented in the operating system *kernel*, which imposes a number of limitations on the usual development, profiling and debugging techniques that can be employed when adding hardware accelerators to an application, as many software development tools do not (fully) work in kernel space.

The DeLiBA framework introduced here is a proposal to alleviate these difficulties for easier research. It lifts key functionality of the modern, multi-queue-based part of the Linux block I/O stack into user-space, enabling the use of a wide spectrum of programming tools and techniques. On the FPGA side, it seamlessly interfaces with a powerful SoC design and integration framework that encapsulates and automatically generates many of the low-level aspects of FPGA accelerators (e.g., PCIe interfacing, DMA, interrupt-based completion signaling, parameter passing etc.), and makes them accessible from abstract APIs. By tackling the I/O acceleration problem from *both* ends, significant gains in development productivity can be achieved. The first use of the framework in DeLiBA1 [37] allowed the hardware-acceleration of compute-intensive storage algorithms, the refined version DeLiBA2 presented adds hardware-acceleration for the network communication operations.

The remainder of this paper is organized as follows. In Section 2, we give an introduction to the existing Linux block layer for distributed storage and point out some of the performance bottlenecks. Section 3 describes the software architecture of our DeLiBA2 framework, while Section 4 introduces the hardware interface. As an initial use-case for DeLiBA2, we have implemented a proof-of-concept of an I/O accelerator for the client side of the Ceph storage protocol. Thus, Section 4 discusses some key Ceph operations and their corresponding hardware design. Since we are addressing the bottleneck in the I/O stack of *distributed* storage, Section 4 discusses the integration of hardware accelerated storage functions with a hardware TCP/IP stack, which is one of the key contributions of this work. Section 6 presents the results of an initial performance evaluation of the accelerators and network interface. We close with a discussion of related work in Section 7 and conclude in Section 8, also looking forward to future work.

2 THE LINUX BLOCK I/O LAYER AND ITS LIMITATIONS

In order to motivate our design choices for the DeLiBA2 framework, we give a brief overview of the Linux block I/O subsystem, specifically when used in a *distributed* storage environment.

The Linux block layer is a kernel subsystem that is responsible for handling block devices, e.g., hard disk drives (HDDs), solid-state disks (SSDs), and remote storage (SAN) [2]. Applications submit I/O operations (hereafter: I/Os) via kernel system calls (`sysread()`/`syswrite()`), which are described in a data structure called a block I/O (`bio`). Each `bio` contains information such as address, size, modality (`read()` or `write()`), or type (synchronous/asynchronous). Over the years, the block layer has undergone a major change to move from a single request queue to a multi-queue model [5], 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. In order to add support for the block multi-queue to a storage protocol like SCSI, the scsi multi-queue (scsi-mq) work was also merged in the 3.17 kernel [11].

As Figure 1 shows, in its present form, the Linux block layer provides per-core request queues called *software queues*. These software queues are configured based on the number of CPU cores in the system, to reduce the lock contention with a single request queue. Below the software queues,

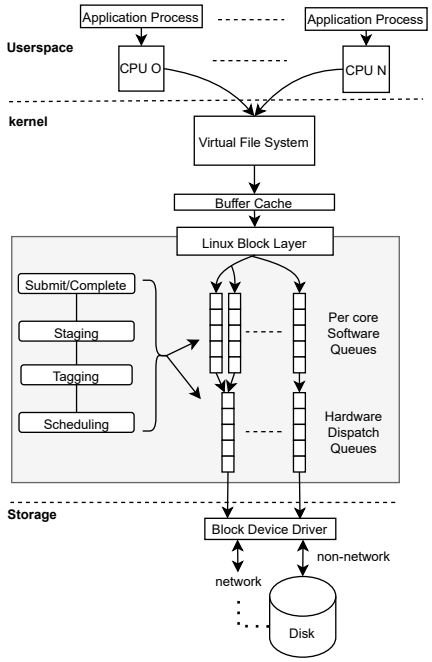


Fig. 1. Existing Linux Block Layer

per-device *hardware queues* provide a second level of buffering: `bio` requests scheduled for dispatch are not sent directly to the device driver, they are instead sent to the corresponding hardware dispatch queue. The number of hardware queues will typically match the number of hardware contexts supported by the device driver. Device drivers may choose to support anywhere from one to 2,048 queues, as supported by the message-signaled interrupts (MSI-X) standard. In contrast to the legacy I/O stack, the new block layer has higher flexibility to optimize access to the underlying PCI Express (PCIe) interface used to communicate with actual hardware.

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 remote distributed storage. as these also can send or receive packets in multiple hardware queues. These NIC-based multiple queues were introduced to improve virtualization in Linux network stack, but they also proved useful for load balancing and dispatching in multi-core systems [3].

However, a consensus is growing in the storage community that even with these improvements, the block layer has not kept up with novel storage hardware [4, 42]. We will analyze some of the deficiencies and make a case for FPGA acceleration of selected parts of the stack.

Many of current Linux core operations are slower than they were some years ago, e.g., due to the need to cope with ever more complex systems architectures [56]. This has impacted the Linux storage stack as well, e.g., with regard to latency: As Figure 1 shows, a deeply layered kernel hierarchy is used to translate from I/O requests to the actual storage operations. This layered architecture adds a significant overhead along the entire request path. Measurements have shown that it takes between 18,000 and 20,000 instructions to send and receive a *single* fundamental 4 kB I/O request [13]. In x86 systems, around 50% of the total execution time of a single 4 kB I/O request is spent in submitting and completing I/O requests at the kernel stage [38]. Whereas for

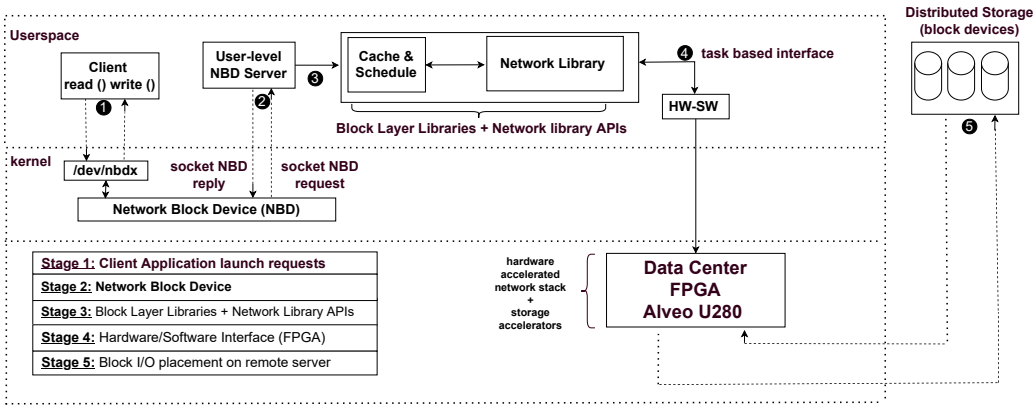


Fig. 2. Architecture overview of DeLiBA2, showing the new hardware network stack integration

32-bit ARM Cortex A9 processors, the overhead of a 4 kB I/O reaches 90 % of the total execution time, with storage device latency only adding 10 % [68]. As a further complication, the block I/O scheduler(s) [45, 46, 71] are also sometimes at odds with the CPU scheduler(s) [70]. This leads to I/O bound processes not receiving sufficient CPU time to actually execute at their desired priority [39], sometimes leading to workarounds where the non-I/O limited parts of a workload are artificially slowed down by `sleep()` system calls to "bump up" the throughput of I/O intensive processes. However, this gain is only achieved at the cost of increased latency.

Balancing I/O latency and throughput is a long-standing problem. A recent study shows that conventional approaches do not achieve both goals simultaneously [30], and suggests to re-architect the existing storage stack again. As an alternative to these software-centric efforts, we propose to examine the use of hardware acceleration at higher levels of the stack. However, research in this area is hindered by the lack of high-level frameworks and the somewhat "hostile" nature of the kernel-space programming environment, in which many programming, debugging, and profiling techniques work only with restrictions, or not at all. Conventional storage software stacks are unable to meet the needs of high-performance storage. In terms of I/O request rates (IOPS), storage devices are an order of magnitude slower than network devices. For example, the fastest SSD devices operate at around 1M IOPS (I/O operations per second) per device, whereas a state-of-the-art NIC is capable of handling more than 70M packets per second. This slower storage I/O rate means that several legacy OS improvement efforts for storage handling are *still* considered worthwhile, even on modern hardware. Furthermore, in distributed storage the bottleneck is further aggravated by the underlying Linux network stack [9, 10], where even state-of-the-art designs spend 70% to 80% of CPU cycles in just handling TCP connections [35].

3 FRAMEWORK ARCHITECTURE

To enable easier experimentation with hardware-accelerated block I/O stacks, our framework moves the software-side processing up from the kernel-space back into user-space. To this end, we rely on the Network Block Device (NBD) to bidirectionally transfer requests between user and kernel space. NBD is a Linux based block device protocol that allows to export a block device to a client application. It has been a part of the standard Linux kernel since version 2.1.67 [6, 23, 74]. Note that the use of NBD does carry a performance penalty, as the additional user-kernel space switches take time. However, as our initial focus for DeLiBA is to enable research into this area, we are willing to

accept the overhead. Also, we will show later, that even with these overheads, performance gains can actually be achieved using FPGA acceleration. With the update to DeLiBA2 described later in this text, we improve the speed-ups even further, but still retain the flexibility in the front-end of the block I/O pipeline achieved by enabling easier development in user-space. Figure 2 sketches the main architecture of the improved framework DeLiBA2, which integrates storage accelerators with a hardware network stack. This integration reduces the latency compared to the original DeLiBA [37] as it avoids both user/kernel context switches, as well as hardware/software execution-mode switches, e.g., from software to NIC, in the tail-end of the distributed block I/O storage pipeline.

In the example shown in Figure 2, a client application issues a read operation ❶, which passes through various filesystem layers (not shown here), and ends up at the driver responsible for the device the data is stored on. For our purposes, that physical storage is provided by the nbd driver, which redirects the I/O requests back up into user-space ❷ using a netlink interface.

With the I/O processing now being handled in user-space, we have to provide similar functionality to what would be available in the kernel, which is unavailable to user-space code. Fortunately, most of what is required comes with Ceph, which provides several software support libraries (`librados`, `librbd`) to assist the I/O processing ❸, similar to what would be used at kernel-level. By reusing the original Ceph implementations in our user-space framework, we can avoid having to develop the algorithms from scratch, and always remain compatible with the original code.

With the DeLiBA framework, any step(s) of the user-space I/O processing pipeline could be offloaded to accelerated hardware ❹ (see Section 4). In the original DeLiBA, the network communication taking place *after* the local I/O processing steps (both software and hardware-accelerated) had completed was also initiated from user-space software. This incurred one switch back from hardware to software execution (returning from the storage hardware accelerators), one user-kernel context switch (from the network operation to the kernel network stack), and one more switch from kernel software execution to the NIC hardware. In the updated DeLiBA2 described here, these last three switches are eliminated, since the storage hardware accelerators now directly interact with a hardware network stack.

The hardware network stack, in turn, interacts with the physical network to communicate with the distributed storage (here: Ceph) server ❺.

3.1 Cache and Scheduler

In our scenarios, the nbd driver just redirects the I/O requests back into user-space. It does not participate in any of the kernel's support mechanisms to manage and optimize I/O operations. But we can achieve the required functionality by providing similar mechanisms in user-space. As describe above, to realize the DeLiBA Block Layer Libraries, we do not have to start from scratch but can leverage parts of the functionality provided by Ceph's `librbd` and `librados` libraries to realize these interfaces. We employ these to realize two key functions.

First, as our user-space approach cannot benefit from the kernel's page cache, we employ the `librbd` library to realize our own caching facility. We employ the Least Recently Used (LRU) replacement strategy and use the default cache size of 32 MiB, which is the same size used by the in-kernel Ceph rados-block device (RBD) driver.

In addition to caching, we also employ the facility as a first step towards coalescing multiple I/O requests for improved throughput. The next step of request coalescing takes place in a custom I/O *scheduler* we created. We use a self-tuning algorithm to delay I/O requests by up to *half* the currently observed average I/O latency. Requests arriving in that time window will be coalesced together for further processing. The self-tuning algorithm uses `boost::accumulators` as its core data structure, which are controlled by the `ROLLING_WINDOW parameters`. Currently, these are configured for `rolling_count` and `rolling_sum` operations, yielding the current delay of up to

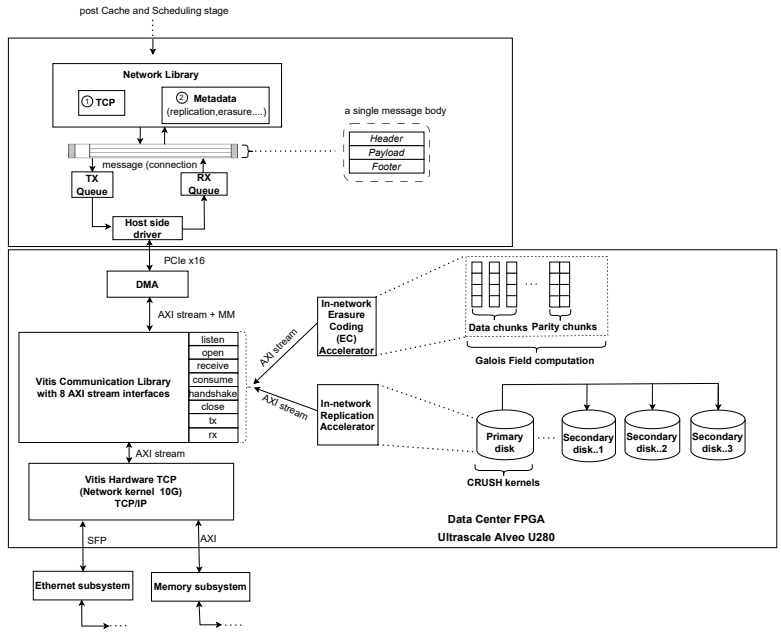


Fig. 3. Integration of User-Space Networking Library with FPGA-based TCP/IP stack

half. In a later step, care will be taken to transfer all the coalesced requests in a single PCIe burst transfer to the FPGA for handling.

However, when moving on to DeLiBA2, this approach has two disadvantages: First, the additional execution mode switches from FPGA hardware back to software had a non-negligible overhead (e.g., due to interrupt-based signaling and result/status transfers using programmed I/O), especially for high packet rates. Also the use of the Linux network stack necessitated yet another user-kernel context switch, as well as another software-hardware execution mode switch when using the actual NIC.

Second, the software I/O pool is realized using worker threads [48–50], which in turn are executed as processor hardware threads managed using the POSIX `pthreads` API. This is a heavy-weight interface, e.g., in terms of required context switch overhead, but also due to the frequent inter-thread synchronisation required by the I/O pool to coordinate access to shared data structures. Together, these issues carry a relatively high overhead, as discussed in [48–50].

3.2 User-Space Network Library

Figure 3 shows the integration of user-space network library with the FPGA-based TCP/IP stack. Following caching and scheduling, the operations are ready to be issued in the form of the three I/O primitives `read`, `write`, and `flush`. Both `read` and `write` are asynchronous operations, while `flush` operates synchronously on the request cache. To support the asynchronous operations, DeLiBA had to extend each of the current I/O primitives with a *completion*, indicating a callback function to be executed when the I/O operation has actually been completed and to carry status information. E.g., for a `read`, the return value of the completion is the number of bytes read on success, for a `write`, the return value of the completion will be 0 on success. Negative error codes can be used to express the reasons for a failed request.

At this stage, all of the basic housekeeping has been performed and the requests have been set up properly (after caching and coalescing) for more advanced processing. E.g., for Ceph, a key operation is determining the physical storage (OSDs, drives) in the distributed storage cluster where the data indicated in the I/O requests is actually located. Contributing more than 50-60% to the overall workload, these are computationally expensive operations, and thus an interesting candidate for hardware acceleration.

To allow the hardware-accelerated execution of storage operations, an I/O Pool is used in DeLiBA that is responsible for aforementioned advanced processing. As shown in Figure 3, this pool is responsible for launching the actual hardware tasks corresponding to the I/O operations, as well as tracking their completion status, and performing the completion routines when an operation has finished. The I/O Pool is used to perform four operations: request/reply buffer memory management on the FPGA, setting-up the network communications, preparing the block I/O request (and handling a reply), and launching of the actual hardware tasks.

The host-side I/O requests, coming in either from the host (software) or from FPGA-accelerated processing, are packaged into a Ceph-specific network protocol called *Messenger* to communicate with the remote Ceph storage cluster. Such a message contains a header (source and destination addresses / ports, message type, Ceph session numbers etc.), a footer (mainly holding CRC values), a so-called *front-message* body and possibly a data payload. During the replication phase, the key contents of the front-message body will be the *parameters* related to the required replication accelerator on the FPGA. Similarly, during the erasure-coding phase, the front-message body will contain parameters related to *parity chunks* and *data chunks*. Note that the Messenger layer has no knowledge of the actual distributed storage algorithms or the lower-level TCP communication formats of the various involved service daemons. For DeLiBA, the actual network communications were performed in software on the Linux kernel networking stack and NIC as usual.

However, when moving from the initial DeLiBA1 on to DeLiBA2, this approach has two disadvantages: First, the additional execution mode switches from FPGA hardware back to software had a non-negligible overhead (e.g., due to interrupt-based signaling and result/status transfers using programmed I/O), especially for high packet rates. Also, the use of the Linux network stack necessitated yet another user-kernel context switch, as well as another software-hardware execution mode switch when using the actual NIC. Second, as described above in Section 3.1, the software I/O pool with its use of heavy-weight processor threads, also becomes a bottleneck.

DeLiBA2 addresses both of these issues. First, the network operations are now also performed in the FPGA, avoiding the communications-induced switching overhead. Second, the multi-threaded I/O pool has been replaced by a much lighter-weight single-threaded task-based interaction with the FPGA, which uses a simple FIFO to just pass the requests on and performs the request handling now completely in hardware, avoiding all of the software based synchronization overheads. On the software side, the previously observed heavy lock contention was avoided by the excision of the inter-thread shared data structures.

As will be shown in Section 6, this approach yields the largest gains for high-request rate scenarios, e.g., when operating on small storage block sizes.

Communication of these operations with the hardware accelerators on the target FPGA, a Xilinx Alveo U280 card, occurs via memory-mapped I/O (control/status data) and DMA (bulk data) via a PCIe Gen3x8 interface. The details of these low-level operations are abstracted using the TaPaSCo hardware/software framework [27], which offers a task-oriented view of the accelerators that fits very well with the request-based nature of the I/O operations.

3.3 Performance of DeLiBA in Software-Only Mode

As a baseline, we begin our performance measurements by examining the original DeLiBA architecture in *software-only mode*. Thus, we employ the NBD-based redirection to user-space, but do *not* use any hardware acceleration.

Our testbed consists of a single client with 6 cores and a cluster with 8 drives. All nodes on the cluster run Linux kernel version 5.2 on CentOS. The client node runs on Ubuntu 18.04. The underlying hardware uses an AMD EPYC Rome 7302P 16-core CPU with 128 GB of memory. In order to verify the network speed among the nodes, we have used the *iperf* network testing tool. For our 10 GbE network, the bandwidth performance range we achieved is 9.2 Gb/s. We use the notation *seq* and *rand* here to identify the sequential and random-access workloads, respectively.

We show results for both a synthetic as well as a real workload. The synthetic workload is generated by the Flexible I/O (*fio*) tool [21] traditionally used for storage testing. With its fine control, we can easily create the necessary scenarios for detailed measurements.

The real workload consists of compiling the entire Ceph system using GCC from source stored on a Ceph block-storage device with a CephFS filesystem on top, created with default `ceph fs volume create` parameters. For this real workload, we do not have fine control over block sizes and will just give aggregate performance numbers (wall clock execution time) for each of the two test scenarios. The Ceph source code compiled here consists of a total of 3,956 source files having a total size of 47 MB and containing ca. 1,55 million lines of code. The build process itself generates a total of 30 GB of output, which is also stored on the CephFS volume.

Furthermore, we have quantified the bottlenecks into two forms, i.e. latency inflation and throughput degradation. Considering the most popular request sizes of many applications are 4 kB, 8 kB, 64 kB, 128 kB and 512 kB, we show bar charts for these values. For the "other" block sizes of 16 kB, 32 kB, and 256 kB, we have to present the measurements in a condensed fashion as numerical tables, since their inclusion in the bar charts would reduce legibility.

For the *Messenger*, which executes using the multi-threaded I/O pool in software, we stayed with the default option of 1:1 core:thread affinity. Since Ceph is a complex system, we have additionally verified our testbed and profiling results by using the exact testbed configurations as created by Ceph's widely used deployment tool [16]. The tool generates a synthetic workload to benchmark a real data center cluster and is part of the original source code [17]. Since Ceph is a distributed storage system, much effort is spent on guarding against data loss, for which three mutually exclusive methods are employed: One is *replication*, where the same data is replicated in a distributed manner across multiple storage nodes, the second is *erasure coding*, where mathematical techniques are employed to compensate for the partial loss of data, and the third is *data scrubbing* [22], which is a server-side technique that is not relevant to our use-case.

The main challenge for replication is to quickly to determine in a highly scalable manner where a certain piece of data is physically stored among the (potentially many) storage nodes. Erasure coding encodes the user data in a manner similar to Forward Error Correction (FEC), but is specialized to handle *missing* data (e.g., due to failed storage media or nodes), instead of erroneous/corrupted data, and can reconstruct the original data from its encoded representation.

At a system level, the storage overhead of replication is proportional to the replication scheme used. For instance, a 3-replication scheme will carry an overhead of around 300% because each copy of data contributes 100% of storage overhead. In contrast to the storage overhead of replication, erasure coding has a higher compute and memory cost [59, 75].

3.3.1 ***Latency.*** Figure 4b and Figure 5b give the latency for the commonly used block sizes 4 kB to 512 kB, for replication and erasure coding, respectively. Here, the latency has the three main sources marked in Figure 2: ❶ the block I/O request transmission from the client application to

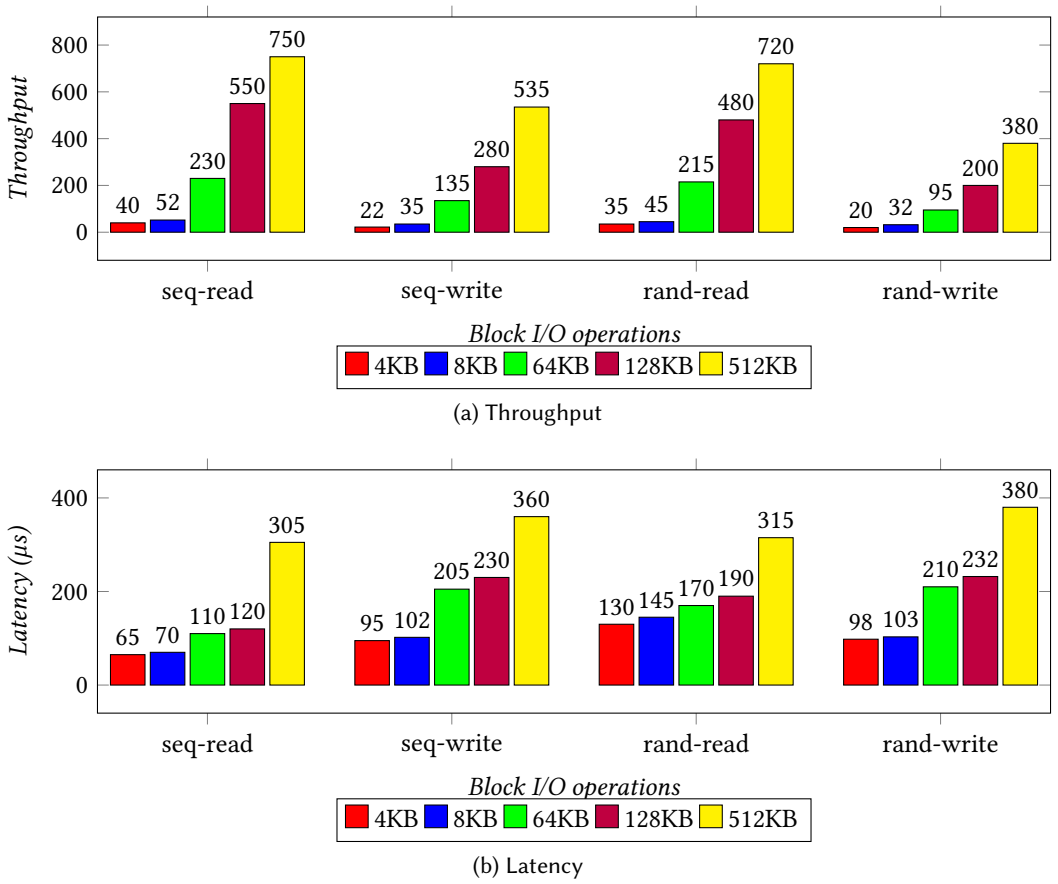


Fig. 4. Latency and throughput results for the software baseline for replication-based storage with a Network Block Device (NBD) in-the-loop

stage ❷ the nbd kernel client, then stage ❸ the user-space library. Finally, at the last stage the block I/O request is placed on the target block device in the storage cluster. For comparison, when using Ceph natively *without* nbd, our measurements showed the latency between 90 μs and 95 μs for 128 kB in case of 128 kB *reads*. Note that these latency measurements are subject to the jitter typically occurring in a non-realtime environment such as Linux.

Since the end goal is to use low-latency FPGA accelerated network stack, it was important to explicitly profile the end-to-end latency of a single block IO request in *software-only* mode. This implies profiling latency from stage ❶ (client application) till last stage ❺ (distributed storage) of Figure 2. To this end we have used a reliable profiling method, namely the enhanced Berkeley Packet Filter (eBPF) tool. The Linux kernel has supported BPF since version 2.5. A key reason to use eBPF was its ability to easily inject C user-space code into the kernel part of nbd that resides between stage ❶ and stage ❷ without changing the kernel source code. This allowed the detailed latency measurements shown here.

The bar chart given in Figure 6 shows the various block I/O request patterns and their impact on overall latency, as introduced in Figure 2. The graph is divided into four stacked bars, each

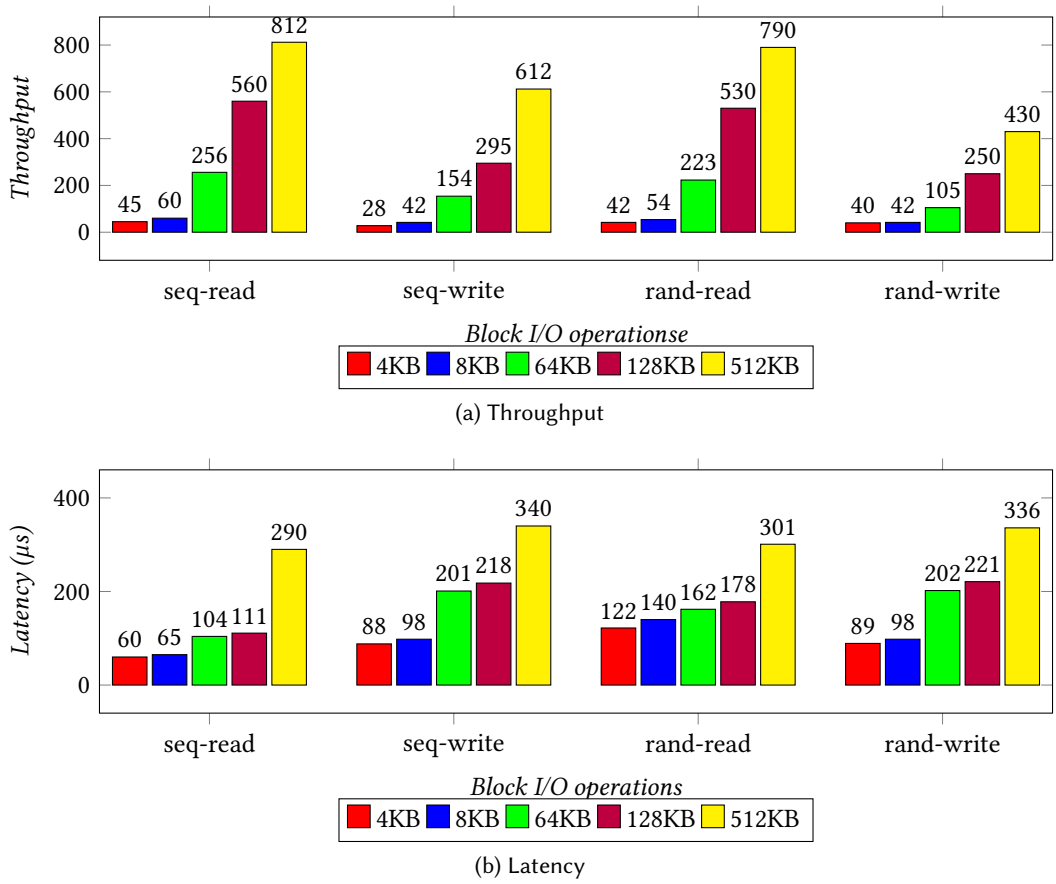


Fig. 5. Latency and throughput results for the software baseline for erasure coding-based storage with a Network Block Device (NBD) in-the-loop

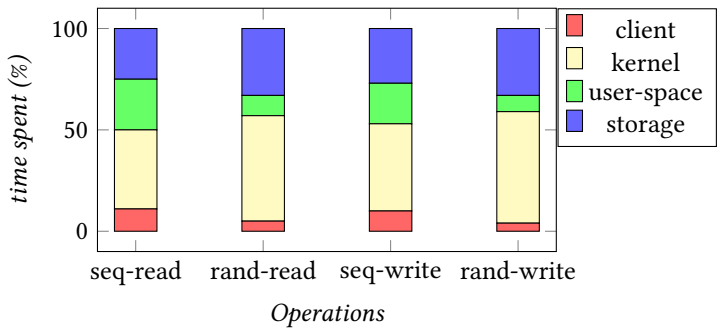


Fig. 6. Latency breakdown for a 4 KB block I/O request w.r.t to main DeLiBA architecture

representing a specific stage of the main DeLiBA2 framework architecture in Figure 2. We captured the times for a specific scenario involving a *single* 4 KB block I/O request across four distinct

Table 1. Throughputs of software baseline for"other" block sizes

Block Size	Request Patterns			
	seq-read	seq-write	rand-read	rand-write
16 kB (replication)	68 MB/s	55 MB/s	62 MB/s	48 MB/s
32 kB (replication)	155 MB/s	102 MB/s	140 MB/s	78 MB/s
256 kB (replication)	660 MB/s	455 MB/s	515 MB/s	360 MB/s
16 kB (EC)	85 MB/s	60 MB/s	78 MB/s	50 MB/s
32 kB (EC)	175 MB/s	106 MB/s	152 MB/s	92 MB/s
256 kB (EC)	680 MB/s	502 MB/s	590 MB/s	450 MB/s

non-direct (buffered) block I/O request patterns: sequential read, sequential write, random read, and random write.

We begin in stage ❶ of Figure 2, which is represented by the stacked bar labeled *client* in Figure 6. This stage involves the generation of the block I/O request by the client application, and the subsequent transition from user mode to kernel mode. Regardless of the specific pattern, the time taken in this stage remains almost constant for sequential read/writes and random reads/writes, averaging at around 7% of the overall latency.

Moving forward, stage ❷ of Figure 2 represented as stacked bar *kernel* shows the duration spent *within* the nbd kernel library, bridging the gap between stage ❶ and stage ❷. It is measure to be the most time-consuming part across all access patterns. On average, this stage accounts for 47.25% of the overall latency, indicating its substantial impact on the end-to-end latency.

The stacked bar *user-space* represents the time spent in user-space libraries at stage ❸ of Figure 2. Across the different patterns, this stage contributes an average of 15.75% to the overall latency. It is worth noting that during this stage, the most significant latency contributors are the cache and scheduler libraries, particularly when dealing with *sequential* read or write block I/O requests. This finding suggests that the efficiency of the cache and scheduler becomes crucial for optimizing the system's performance during these specific I/O operations. For random read or write block I/O requests, the time spent in user-space cache and scheduler is relatively shorter, which is most likely due to the expected cache misses associated with random access patterns, resulting in reduced time spent in the user-space cache and scheduler.

Finally, the *storage* stacked bar represents the time spent at the storage end, occurring between stage ❹ and stage ❺, *including* the network latency. On average, this stage contributes 29.5% to the overall latency. It encompasses the operations related to the storage system, including data retrieval or storage processes, and plays a significant role in determining the final latency experienced by the DeLiBA2 framework.

3.3.2 ***Throughput and IOPS***. The Figure 4a and Figure 5a shows the throughput for block sizes 4 kB, 8 kB, 64 kB, 128 kB and 512 kB. For replication mode, the maximum throughput achieved is for 512 kB sequential reads with 750 MB/s, and the maximum IOPS (I/O operations per second) are achieved for 4 kB sequential reads at 9,000 IOPS. EC has similar latencies, but achieves higher throughputs. Table 1 condenses the measurements for the "other" block sizes that would not fit into the bar charts.

For comparison, using Ceph natively, *without* the nbd relay to user-space, achieves roughly 800 MB/s in this setup. Note that the penalty for the NBD kernel-user space redirection we employed in DeLiBA1 thus mostly affects latency and IOPS, and has only a limited effect on *throughput* for the larger block sizes, being, e.g., just $\approx$ 7% slower for 512 kB requests in replication mode, and even reaching non-NBD throughputs for EC mode.

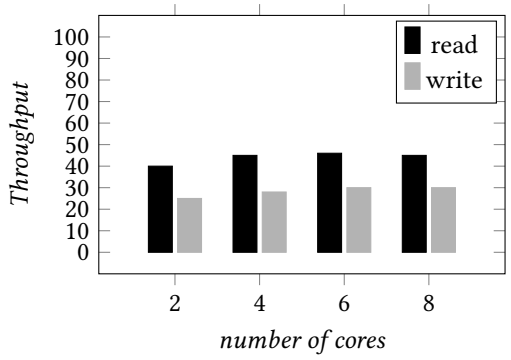


Fig. 7. Scaling-up core counts in software-only mode with a Network Block Device (NBD)

3.3.3 ***Wallclock Performance on Real Workload.*** For our real world workload of compiling Ceph itself on a remote CephFS filesystem held on our software baseline configuration (software Ceph block storage with NBD-in-the-loop), the average wall clock execution across ten full-system compilations time is 132 seconds (min. 128s, max 140s), with the file system cache being cleared between runs.

3.3.4 ***Impact of Scaling-Up Core Counts.*** Given the discussion of overhead sources in Section 3.2, we examine to what extent we can raise the performance by scaling up the number of CPU cores used to issue more I/O in parallel requests in a client. To this end, we used a different server configuration that attaches a single fast NVMe SSD to the Ceph OSD storage daemon, replacing the HDDs used in the other experiments. The SSD with its many internal queues should easily be able to keep up with the higher request rates that might be enabled by using more cores on the client to issue them. The actual requests are again generated by *fio*, which this time is configured to create a parallel workload using multiple queues across the cores. Specifically, we configured *fio* to perform 4 KB sequential reads and 4 KB sequential writes at an *iodepth* of 64.

The results of increased core counts to generate requests are shown in Figure 7. It is clear that more cores generating requests do not linearly improve system throughput even when using SSD storage on the server. The maximum throughput for 4K read stalls at 45 MB/s for the single disk. Likewise, the maximum throughput for 4K writes tops out at 30 MB/s. Thus, in the software baseline we evaluate here, the Ceph client-side is unable to keep up with more I/O requests, as it expends too much effort in its own thread pool-based processing architecture.

4 ADDING HARDWARE ACCELERATION TO DELIBA

The nature of block I/O operations fits well with the task-based computational model, where each I/O request can be mapped to a task to be executed on the FPGA accelerator. Thus, it makes sense to apply an existing framework for task-based FPGA computing to the I/O scenario. Specifically, both the original as well as the new version of DeLiBA rely on the *Task-Parallel System Composer* (TaPaSCo) [27] as a middleware to dispatch the I/O requests to the actual FPGA accelerator for processing and to perform the required high-performance DMA transfers. To evaluate the improved practicability and performance of DeLiBA, we have used it to construct a block I/O accelerator for the Ceph distributed filesystem. The next sections will discuss the key Ceph kernels and their hardware acceleration.

Table 2
Software Profiling results, High-Level Synthesis (HLS) Estimates, and HW execution times for Ceph kernels

Kernel	SW Execution Time	Overall contribution to runtime	Vitis HLS Cycles (min-max)	Vitis HLS Latency (min-max)	HW Execution w/ network	SLOCs SW	SLOCs HLS
Straw Bucket (pure HLS code)	85 μs	70% - 75%	133 - 135	0.665 μs - 0.675 μs	62 μs	256	148
Straw Bucket (using Vitis $\ln x$ function IP)	85 μs	70% - 75%	177 - 177	0.885 μs - 0.885 μs	62 μs	256	130
List bucket	65 μs	70%	56 - 56	0.280 μs - 0.280 μs	65 μs	197	134
Tree Bucket	45 μs	75%	161 - 162	0.805 μs - 0.810 μs	37 μs	241	152
Uniform Bucket	20 μs	50%	46 - 448	0.230 μs - 2.240 μs	22 μs	237	161
Reed-Solomon Encoder	120 μs	> 50%	240 - 245	2.56 μs	105 μs	280	173

In addition to speeding-up the storage functions, DeLiBA2 also integrates their accelerators with an an open-source hardware-accelerated TCP/IP stack [26, 60, 64, 73] to reduce both the number of user/kernel as well software/hardware execution mode switches.

4.1 Hardware I/O Accelerator for Ceph Clients

The Ceph client with its complete source code was carefully profiled on a bare metal server using tools such as Intel VTune Profiler [33] and Valgrind [72] to determine compute-intensive processing operations and collect the run-time call graphs. An important metric in profiling was to measure the contribution of the software kernels to the overall load.

The original DeLiBA only provided hardware accelerators for the Ceph CRUSH replication scheme [76], while DeLiBA2 discussed here also supports accelerators for erasure coding. The third option would be scrubbing, which scans all data and compares it with the other replicas to detect and fix inconsistencies. While it is a very I/O-intensive operation, it occurs under server control and is out-of-scope for this work, which focuses on client side acceleration.

4.2 Profiling Replication

Based on the profiling results in the first two columns of Table 2, we have initially focused in DeLiBA on the acceleration of the compute-intensive replication algorithms in our proof-of-concept.

A key aspect of Ceph is the avoidance of a central directory for keeping track of the physical locations of the replicas of a piece of data. Such a central directory would quickly become a bottleneck when scaling the distributed storage system to (many) more nodes. Instead, Ceph employs algorithms that enable each client to *locally* compute the physical replica storage locations by itself, using just a logical integer ID for the piece of data (similar to a block number), and a description of the available storage resources in the Ceph cluster, called a *cluster map*.

For replication, just like other replication algorithms [7, 24, 36, 43, 66] in distributed storage, Ceph relies on a pseudo-random data distribution algorithm [28, 29] named CRUSH (Controlled Replication Under Scalable Hashing) that distributes replicas across block devices. CRUSH defines *four* different kinds of computations to represent internal nodes in the storage cluster hierarchy: `uniform`, `list`, `tree` and `straw2`. In addition to the parameters of the current storage operation, they receive a "map" of the entire storage cluster describing the different physical storage resources.

Prior profiling reports of the Ceph system [68] determined that approximately 45% of execution time is expended on the CRUSH mapping function, which mainly performs *hashing* operations using a non-cryptographic Jenkins hash function [8]. This heavy use of hashing has become even more intense in more current Ceph versions: Our profiling of the recent Ceph version Octopus 15.2.16 indicates that the client now spends *more than 70%* of its execution time in the CRUSH mapping functions.

We thus heavily focused on the optimized implementation of the hashing operation in the five kernels Ceph uses for data replication. E.g., the 32 bit mixer step has been inlined for all kernels. One of the main goals of profiling was to profile the overall performance at different cluster sizes. Each of the four CRUSH kernels is based on a different internal data structure and computes a different function for selecting nested storage nodes during the replica placement on the block devices.

Their CRUSH kernel execution times vary with the configuration of the storage cluster as shown in Table 2 under *SW Execution Time*. E.g., when the storage cluster has disks with *identical* sizes, for instance, 20 GiB for each disk, the `uniform` kernel execution time was 20μs. When the cluster size frequently *grows*, the `list` function would be used, whose kernel had the execution time of 65μs, as it had to consider the optimal data movement to the new storage nodes as they were added. For a dynamically changing cluster, with both additions and removal of storage capacity, the `straw2` algorithm is suitable. It had an execution time of 85μs. In a cluster with a large number n of storage sets, called *buckets* in Ceph, the `list` approach with its run-time complexity of $O(n)$ becomes too slow. Here, the `tree` algorithm allows deeply *nested* cluster hierarchies, which reduces the complexity to $O(\log n)$, and shortens the execution time to 45μs in the sample scenario.

Table 2 also shows another profile measurement, namely *Overall contribution to runtime*, which lists the percentage of each kernel's contribution to the total Ceph application run time. Only kernels that have a high relative runtime are promising for acceleration. In our work, we looked for kernels contributing at least 50% to the overall Ceph runtime when choosing acceleration candidates. Both the measurements i.e *SW Execution Time* and *Overall contribution to runtime* are profiled based on a particular cluster size. As can be seen, in terms of their software execution times, all of the replication kernels are suitable candidates for hardware acceleration. The column *HW Execution w/ network* gives the execution time of the kernel computation *together* with the network communication. Note that in many cases, the hardware is able to perform *both* operations together in similar or shorter times than software requires just for the computation itself.

The kernels realize multiple algorithms to perform different data replication strategies in a Ceph cluster. Which specific algorithm is used for a given cluster is statically configured in the Cluster map. Only the kernel providing that single algorithm will then actually execute for an I/O request. To give an indication of the implementation complexity for each algorithm, Table 2 also shows the number of source lines of code (SLOC) of the original software code and our HLS implementation.

4.3 Profiling Erasure Coding (EC)

As a new contribution in DeLiBA2, we now also consider the Erasure Coding (EC) functionality for hardware acceleration.

EC is implemented in Ceph as a plugin-in using open-source libraries, specifically Jerasure [15, 52, 53] or Intel Intelligent Storage Acceleration (ISA-L) [32]. The Jerasure library is integrated

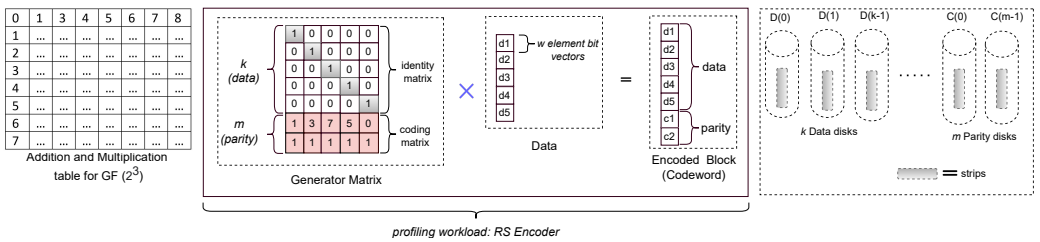


Fig. 8. Erasure Coding encoder using Reed-Solomon for k=5, m=2 and $GF(2^3)$

as the default EC provider in Ceph and it runs on all processors, unlike ISA-L which is specifically optimized for Intel processors. Jerasure provides multiple EC algorithms, however, one of the most commonly used ones is *Reed-Solomon (RS)* [55]. Figure 8 depicts a common example of a typical RS encoding process. As shown, for a given $RS(k, m)$ code, there exist k *data* packets and m *parity* packets, sometimes also called *coding* packets. These codes represent a storage system as a set of linear equations, with the arithmetic of these equations being based on *Galois Field arithmetic* described as as $GF(2^w)$, where all elements of the system are based on w-bit words. The required arithmetic involves addition, multiplication and division. Considering the 32-bit and 64-bit machine word boundaries, the values of w commonly employed are 8,16,32 in Ceph. For encoding, each disk is partitioned into w-bit word *strips*, with a collections of $k + m$ strips being called a single *stripe*. The figure shows $k = 5$, $m = 2$, and $w = 3$ for $GF(2^w)$. This particular example will thus yield 8 disks in the cluster. Similarly, for $w = 4$ and $w = 8$, the cluster will have 16 and 256 disks, respectively.

As we currently focus on *client-side* acceleration for distributed storage, we only perform EC *encoding* in DeLiBA2, as the computed parity data for each write will be sent to the storage nodes together with the payload data. In contrast, EC decoding will in general only be required on the *server-side* when reconstructing (repairing) failed storage media or nodes, and will involve retrieving data (payload and parity) from multiple other nodes to recompute the missing pieces of data.

When profiling the software-based EC encoding, we examined multiple (k, n) encoder configurations. E.g., Table 2 shows the times for an $RS(6, 3)$ code. In all practically relevant configurations, the relative software run-time exceeded our acceleration threshold of 50% of the Ceph client-side total execution time.

5 IMPLEMENTING THE STORAGE ACCELERATORS AND THEIR NETWORK INTEGRATION

Based on these profiling results, we can now proceed to create hardware accelerators for the various replication and the EC encoding storage functions. In contrast to DeLiBA, we will tightly integrate the storage accelerators with the networking functionality in DeLiBA2, avoiding the extra overheads caused by more user/kernel context switches and the need for interacting with a separate NIC. Instead, we rely on a customized version of an open-source hardware TCP/IP stack [60, 64] for communication. A key challenge here is the layering of the Ceph Messenger protocol on top of the TCP connection provided by the stack.

For both kinds of storage accelerators, we will first explain their hardware implementation and then their integration with the hardware networking layer.

5.1 Target Hardware and SoC Architecture

Figure 3 shows the schematic view of the FPGA architecture in DeLiBA. The target FPGA board is a Xilinx Alveo U280, which carries an UltraScale+ FPGA. The software on the host communicates with the Alveo U280 FPGA over a PCIe Gen3x8 interface. A DMA engine is provided by the FPGA device for fast memory transfers. Accelerators for in-network Replication and Erasure Coding are integrated with the hardware TCP/IP network stack, which itself is organized as two main subsystems.

The stack's Ethernet subsystem implements both the MAC sublayer and upper parts of the physical layer, and uses one or more of the FPGA's GTY high-speed serial transceivers, which implement the lower part of the physical layer for 10G Ethernet. This subsystem exchanges Ethernet frames with upper-layer components using two separate 64-bit AXI Stream interfaces for the Rx (receive) and Tx (transmit) directions.

The stack's memory subsystem manages read and write requests to the TCP send and receive buffers. In order to integrate the network stack with our replication and EC accelerators using AXI

streams, we have used the communication library that is already part of the original hardware network stack [73].

In its current form, the packet frames for replication and EC workload are defined ahead of compile-time i.e, there is no complex run-time packet parsing of replication and EC packets required on the FPGA. Both transmission and receiving can be performed by just reading/writing static offsets within the packet buffers.

5.2 FPGA-Accelerated In-Network Replication Operation

At the core of our in-network replication accelerator is the CRUSH algorithm [76]. In our earlier work [37], we have demonstrated that FPGAs can be used to efficiently execute the replication algorithms over two orders of magnitude faster than software on a fast CPU. In this work, we directly integrate the same replication algorithms (also shown in Table 2) into an in-network processing pipeline, as depicted in Figure 3. For our Ceph proof-of-concept I/O accelerator, five key CRUSH operations were moved from software to hardware execution. The hardware kernels were implemented using the Xilinx Vitis 2021.2 High-Level Synthesis (HLS) tool [79]. For HLS, we set a target clock frequency of 300 MHz, which is achieved for all of the individual kernels. Table 2 shows the latency measurements in cycles and seconds for all five of the replication kernels kernels.

At the SoC level, including the TCP/IP stack and all other supporting IP blocks, we still reach an $f_{\max}$ of 200 MHz after place-and-route (see Section 5.4).

Replication Accelerators using HLS. As usual for HLS, the original software code had to be heavily altered for hardware synthesis. Key areas that had to be modified were the heavy use of dynamically allocated memory and variable loop bounds. The original code employed these to support the many degrees of freedom available for configuring a Ceph cluster. However, on an FPGA, it is more beneficial in terms of performance and area usage to *statically* tune individual kernels for concrete configuration parameter values, and instead employ separate, differently parameterized hardware kernels for different cluster configurations. All replication kernels have been pipelined and achieve II=1.

In terms of functionality, `straw2` allows other *buckets* to fairly "compete" against each other when determining replica placement through a process analogous to a “drawing of straws”. The original kernel has to compute a 64-bit natural logarithm using a lookup table. For the pure HLS code version, we initially implemented this approach as a single-port BRAM using a Vitis `RESOURCE` directive. However, for certain values of bucket parameters, such as *id* and *weight*, the table-based computation leads to stalls in the pipeline. To guarantee constant latency, we implemented a second version of `straw2` using the $\ln x$ IP block from the Vitis floating-point library as an alternative. In addition to the logarithm computation, the second key operation in the inner loop of the `straw2` kernel is the computation of pseudo-random numbers using the Jenkins hash function. Both versions of the kernel achieve II=1 when pipelined.

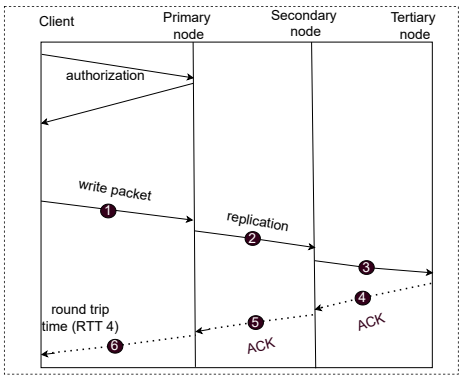
For the case when the configuration of a Ceph cluster is frequently changed, e.g., by adding more disks, the *List bucket* algorithm is used, which is based on linked lists of arbitrary weights. The software version of the kernel begins by computing the parametric hash of the given data. After the hash computation, the kernel performs a *scan* of the list, beginning with the first element holding the most recently added item, and compares its weight to the *sum* of all remaining items weights in the map before returning the block disk *id*. The lists are usually short, e.g., having a maximum of 64 entries here, which is the same as the bucket size in the map. For our hardware implementation, this allows us to avoid the indirect (linked) data structure used in software. Instead, we employ a combination of arrays and FIFOs that allows pipelining with II=1 for the algorithm’s main loop.

Tree Buckets is also used for frequently expanding clusters with more than 64 entries. However, in contrast to *List bucket*, the clusters processed here have more *deeply nested* storage hierarchies. Each node in the tree is aware of the total weight of its left and right subtrees, as well as a unique identifier that is passed to the hash function as a parameter through the map. Similar to the uniform kernel, the tree kernel also does not involve pointers for traversing the tree nodes. Instead, we have implemented its parameters *node weights* and *items* as a combination of array and FIFOs. The software version requires five function calls in each of its loop iterations, two for hashing and three for tree management. For the HLS version, all function calls were inlined for better resource sharing and cross-call optimizations.

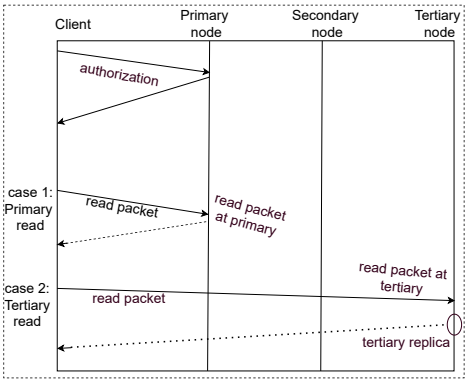
As explained in Section 4.1, the *uniform kernel* performs replicas among (potentially large) groups of *identical* storage devices. The underlying computations are a Fisher-Yates shuffle [19, 40], computing a uniformly random permutation of *weights*, which is then applied to the bucket members. The hardware kernel again uses inlining to combine these operations into a single function, having three nested loops that could all be pipelined to achieve II=1.

Replication Operation in the Network. The next step is to integrate the five HLS replication accelerators with the network stack on the FPGA. For replication, the communication model underlying Ceph assumes ordered transmission of messages i.e. all sent messages will be delivered in the order they were sent. This is most easily achieved my operating the hardware network stack in TCP mode. Hence, all replication accelerators are mainly integrated with the TCP modules of the network stack.

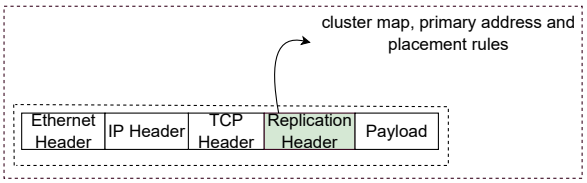
We begin with a description of the general replication flow, shown in Figure 9. Replication is usually performed into two ways in distributed storage: Either the client directly sends the requests



(a) Life of a write packet where final ACK is sent from the Primary node



(b) Life of a read packet in both cases



(c) Write request packet for 3x replication

Fig. 9. Packet Flow for Erasure Coding, example shown for for k=1 and m= 4

to *all* replicas (all copies) or the client sends the request to a *single* replica (single copy), which then replicates the request to the remaining replicas. Ceph implements a variation of the latter, called *primary-copy* replication, where it is the main (primary) storage node that sends the replicas to other storage nodes (secondary and tertiary, for the 3-replication discussed here). To this end, as shown in Figure 9 a typical request flow in *primary-copy* replication consists of following *five* steps:

1. The client divides the block (payload) into multiple packets, and sends these packets to the primary node.
2. The primary node receives the packets, which it forwards to the secondary node.
3. The secondary node receives the packets, which it forwards to the tertiary node.
4. The secondary node waits for I/O completion acknowledgment from tertiary node, which is forwarded with the secondary's own acknowledgment to the primary node.
5. The primary node waits for the I/O completion acknowledgment for the secondary and tertiary nodes.
6. Once *all* replicas are updated and report completion in this manner, the primary node sends a single acknowledgment (ACK) to the client.

Together with so-called *placement rules*, the replication accelerators discussed above are used in this scheme to determine which *specific* nodes to employ as replicas to for an individual write request.

Initially in Ceph, read requests were just handled at the primary node (shown in Figure 9.c, Case 1), not to the replicas. However, newer Ceph optimizations [44] perform *load balancing* to avoid stressing just the primary node with reads, and dynamically select other replicas for reading as well (shown in Figure 9.c, Case 2, by reading from the tertiary replica).

5.3 In-Network Erasure Coding (EC) using FPGA

As shown in Figure 3, the in-network EC accelerator provides erasure *encoding*. It was developed using the Vitis 2021.2 HLS tool. Similar to the replication accelerators, the HLS target clock frequency for the encoder was 300 MHz. Table 2 shows the profiling, HLS and final hardware latency results of the in-network EC accelerator. For the complete SoC on the FPGA (see Section 5.4) with 10 GbE, we achieve a post-place-and-route $f_{\max}$ of 166 MHz.

Next, we explain the HLS transformations performed on the software version of the RS encoder, and their effect on the microarchitecture, integrating the block with the FPGA-based TCP/IP stack.

EC HLS Encoder Accelerator. For maximum compatibility, the EC hardware accelerator has been developed using HLS based on the Jerasure library, which is also used in the software implementation of Ceph. A key change required was replacing the dynamic memory allocations used in the software version with statically allocated arrays for hardware synthesis.

As previously explained in profiling results (Section 4.3), the most compute-intensive operations in the RS encoding is the Galois Field ($GF(2^w)$) operations. These region operations involve constructing matrices that perform the compute-intensive multiplication ($\times$) and bit-wise XORing of operands ($\oplus$) for GF addition (or subtraction).

At the core of our EC accelerator in DeLiBA, we use an 8-bit (255 symbols) *ALU* (algorithmic logic unit) block that implements these regional operations and generates the encoded codeword (parity). To allow better synthesis-time optimization, we make the symbol width a *static* compile time parameter. This implies that the EC accelerator cannot dynamically change field polynomial or symbol width during the operation. Any change would require modifying the FPGA design and recompiling, with Dynamic Partial Reconfiguration being a possible way to work around this limitation. However, this design decision will not impair our Ceph use-case, as the EC parameters will generally not change once configured for a storage cluster.

For the proof-of-concept implementation, the encoder was configured to support *data* packets with $k \in \{6, 8, 10, 12\}$ and *parity* packets with $m = 4$. These packet values are defined as macros in the library. Larger numbers of parity packets, i.e. $min\{8, 12, 16\}$, are also supported by the encoder, but are only rarely used in actual Ceph deployments.

Three techniques are often employed to achieve fast GF multiplication operations: First, using pre-calculated multiplication tables for smaller values of $w = 3$, i.e. $GF(2^3)$. Second, logarithm tables or incremental shifters for larger values such as $w = 8$ and $GF(2^8)$. Third, replacing the expensive multiplications with additional XOR operations. Since the reference Jerasure RS encoder used in our profiling does not apply XOR operations for the multiplication, we have refrained from XOR optimizations in the HLS accelerator as well.

Of these three methods, we thus employed only the first two to stay compatible with the Jerasure approach, as the third option would require completely different matrices. Due to their small sizes of just 48 elements, and to preserve BRAMs, we implemented the encoding matrix as distributed RAMs.

We also employ *Pipeline*, *Memory Partitioning* and *Loop Unrolling* directives to lay out a fully optimized and pipelined RS encoder. Loop pipelining is mainly used to exploit the intrinsic parallelism between successive loop iterations inside the GF matrix multiplication. Furthermore, with the pipelined loops, it is beneficial to implement parallel accesses (data-level parallelism) to the memory inside the nested loops. This is achieved by using a set of memory partition directives that creates multiple memory banks.

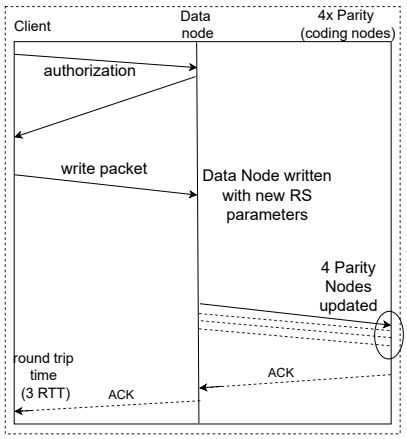
Integration with Network stack. The next step is to integrate the EC accelerator with the hardware network stack operating in TCP mode. Compared to the replication-based data protection discussed earlier, a different scheme of interaction is used. First, EC-based storage uses static storage locations, thus there is no longer the need for something like the CRUSH algorithm to determine storage locations based on a dynamically changing cluster map.

For the EC example of $k = 1, m = 4$ shown in Figure 10, the client interacts only with a single storage node, called a *data node* in this context. When writing (Figure 10.b), the data node also accepts the EC parity packets (Figure 10.a) from the client, and distributes them in parallel to the parity nodes, sometime also called *coding nodes*. Once all the coding nodes have confirmed the successful writing of the parity data, and the data node has stored the payload data, it confirms the completion of the entire write back to the client by sending an acknowledgment (ACK).

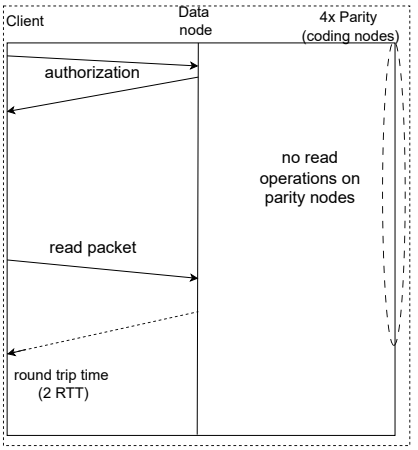
From the client perspective, reading (Figure 10.c) is performed similarly to the replicated-storage scenario, where the accessed data node(s) are determined by load balancing mechanisms. The key point is that during regular operations, the parity data on the coding nodes does *not* need to be read. It will be accessed by the Ceph server-side only if data nodes have failed and the data stored there needs to be reconstructed using the parity data.

5.4 SoC Integration of Storage Accelerators

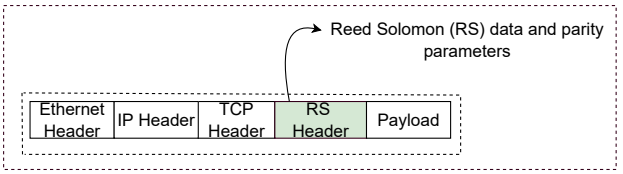
The TaPaSCo hardware/software middleware already supports the use of 10G Ethernet communications from Processing Elements (PEs) such as our storage accelerators. This feature is now used to attach the hardware TCP/IP stack, which is thus inserted between the PEs and the GTY transceiver used for the physical network connection. Note that the stack as well as the Ethernet interface would already support 100G Ethernet, as demonstrated in [25], but in our current work we just use 10G due to the capabilities of our Ceph testbed, as well as to reduce tool run times, which would lengthen even further for the wider data paths required for 100G operation. However, the actual system architecture should indeed scale up to 100G operation with the available FPGA capacities. Also, while the U280 board does support the use of HBM, the Ceph workload is not memory-bound



(a) Life of a write packet where final ACK is sent from Data node after updating 4 parity nodes



(b) Life of a read packet where stored data with previously written RS parameters is read



(c) Write request packet for Encoding

Fig. 10. Packet Flow for Erasure Coding, example shown for for k=1 and m= 4

and can easily use the traditional off-chip DDR4 SDRAM to hold client-side data for the replication and erasure coding kernels shown in Table 2.

A key aspect of the use of accelerators is the communication overhead required for interacting with the host. Fortunately, for the Ceph accelerator, the size of the parameters is relatively small. The largest one is the "map" of the storage cluster, which even for larger configurations rarely exceeds 512 kB and thus is very amenable for a fast DMA transfer to the FPGA. Since the map is static as described in Section 4.2, it is transferred once and reused for the remaining I/O requests. Likewise, for the EC encoder accelerator, the relatively small sizes of *parity* $m = 4$ and *data* chunks $k \in \{6, 8, 10, 12\}$ were also quick for DMA to transfer.

The nature of the computations is also amenable to acceleration: All of the different kernels discussed above execute *independently*. Also, individual I/O requests can be processed *independently*. This is very suitable for execution in a task-based model of computation as provided by the TaPaSCo FPGA framework. Interestingly, with the optimizations performed since our original work [37], the ten accelerator instances used there could be replaced in DeLiBA2 by *single* instances each without performance loss. This is similar to replacing the original multi-threaded software I/O Pools with higher performing single-threaded algorithms (Section 3.2) that no longer require synchronization.

Hence each FPGA accelerator used for the evaluation is realized as a TaPaSCo composition containing exactly only one PE. Even though DeLiBA2 has successfully removed the DeLiBA redirection of *network* communications to software, the overhead of the NBD kernel-user relay and

the user-space libraries remains. Using the software-side C++ timing library `#include <chrono>`, we determined the hardware execution time, e.g., for the `straw2` kernel, to take around 62μs (see third last column in Table 2). When comparing with the numbers given in [37], please consider that the hardware execution times reported here now *include* the network communications until the time the ACKs from the nodes have been received back at the client.

Compared to the original DeLiBA work, the remaining execution mode switches still almost cancel out the per-kernel speedups, with just minor speedups achieved by integrating the hardware network stack. However, since we can now use *pipelined* execution from the hardware kernels down to the network, even with these still-high overheads, we *do* achieve better overall performance gains as shown in Section 6 over the pure software execution.

To realize the full potential of the approach, though, we will have to move our programming framework back *down* into the kernel, while retaining compatibility with the algorithms developed in the "pleasant and productive" user-space programming environment of our current solution. This is the goal of ongoing future work (see Section 8).

6 EXPERIMENTAL EVALUATION OF THE CEPH HARDWARE ACCELERATOR

As described above, DeLiBA2 is used to integrate a 16nm Ultrascale Xilinx Alveo U280 FPGA card providing both storage and network accelerators for the Ceph protocol stack into the Linux block I/O subsystem. The client-side host uses an AMD EPYC 7443P 24-Core Processor with 128 GB of memory, attached by 10 Gb/s Ethernet to the Ceph cluster as described in Section 3.3. The FPGA card is attached to the client host by PCIe Gen3 x16 and uses a system clock ($f_{\max}$) of 166 MHz. The setup also includes a 10G Ethernet network configured to run on the Celestica Seastone DX010 switch. Furthermore, the client has a kernel version 6.0.9-1, with an operating system Rocky Linux 8.7 (Green Obsidian). Similar to the earlier software baseline benchmarks, the hardware evaluation was performed with synthetic (*fio*) workloads for all five major block I/O request sizes i.e. 4 kB, 8 kB, 64 kB, 128 kB and 512 kB. All three major measurements i.e. throughput, KIOPS and latency are average measurements of four multiple runs. Also, as above, we examine the wall clock execution times for the real-world workload of compiling the Ceph sources on a CephFS volume, this time with DeLiBA2 hardware acceleration enabled on the client.

The setup passes all compatibility and functional tests for interactions between the Ceph client and server nodes. The user-space-based programming environment provided by DeLiBA2 considerably simplified the development of the proof-of-concept, as a Ceph-based distributed storage system employs far more complex data structures, administrative computations, and protocols than one, for example, using the far simpler iSCSI (Internet Small Computer Systems Interface) protocol [47].

Although there is still a considerable scope of acceleration, the successful integration of the FPGA accelerated storage and network stack in DeLiBA2 has already improved the overall performance over the original DeLiBA. In our evaluation, we consider both the replication- as well as erasure-coding-based data protection mechanisms, and examine latency, throughput, and I/O rate (as I/Os per second, IOPS).

Latency. With DeLiBA2, we aimed to leverage the data processing capabilities of the FPGA-based network accelerators to speed-up latency of major block I/O operations. In a direct comparison for replication-mode 4 kB operation as shown in Section 6, a major reduction of latency was observed in random-read operation i.e. the latency is reduced from 130 µs to 85 µs, a reduction by 35%. For EC, DeLiBA2 achieves a latency reduction of 20% for sequential-reads from 60 µs to 48 µs.

For operations with the large block size of 128 kB in replication mode, sequential-read has the shortest latency of 108 µs. This is true for EC mode as well, here sequential reads of 128 kB are the

SW vs HW (4 kB)	Latency [µs]			
	seq-read	***seq-write***	***rand-read***	***rand-write***
SW (replication)	65 (SD:0.782)	95 (SD:0.519)	130 (SD:0.631)	98 (SD:0.914)
HW (replication)	55 (SD:0.511)	75 (SD: 0.437)	85 (SD: 0.535)	82 (SD: 0.535)
SW (EC)	60 (SD:0.981)	88 (SD:0.913)	95 (SD:0.823)	89 (SD:0.752)
HW (EC)	48 (SD:1.118)	70 (SD:0.889)	82 (SD:0.696)	75 (SD:0.487)
SW vs HW (128 kB)	**Latency [µs]**			
	seq-read	***seq-write***	***rand-read***	***rand-write***
SW (replication)	120 (SD:0.498)	230 (SD:0.839)	190 (SD:0.487)	232 (SD:0.973)
HW (replication)	108 (SD: 0.4803)	212 (SD: 0.500)	172 (SD: 0.623)	220 (SD: 1.003)
SW (EC)	111 (SD:0.871)	218 (SD:0.564)	178 (SD:0.639)	221 (SD:0.788)
HW (EC)	98 (SD: 0.976)	201 (SD: 0.832)	165 (SD: 0.794)	198 (SD: 0.959)

Table 3

I/O request latency on Software and DeLiBA2 Hardware. Times are averages measured across four benchmark executions and are shown with their standard deviations (SD)

fastest, too, taking 98 µs. All of these times are the *average* measurements over four benchmark runs, Table 3 also lists the standard deviations for the captured numbers.

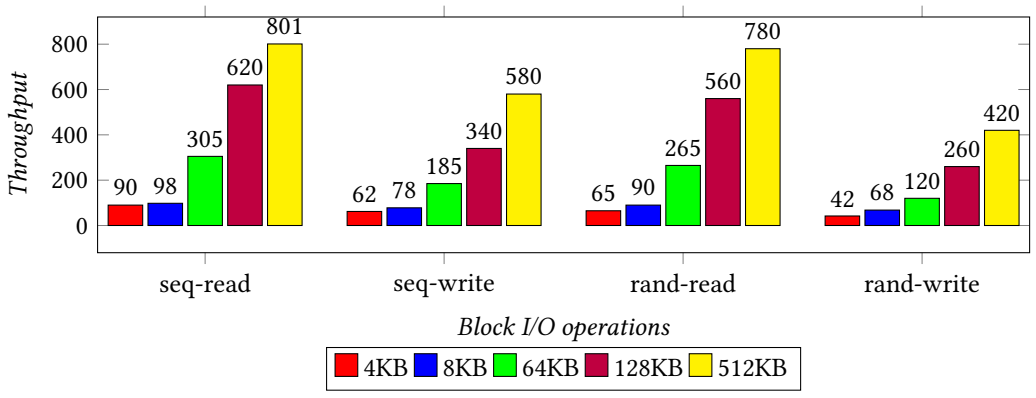
Overall, the FPGA-based network stack in DeLiBA2 has achieved a maximum of 5 µs reduction compared to our earlier DeLiBA1 work where the main framework had no FPGA-based network accelerators. Even with the remaining overheads due to still using nbd as a user-kernel space relay, and the additional PCIe round-trips involved to the FPGA, the latency of using the hardware accelerator is now *always better* than using the native non-NBD software stack, as shown in Section 6.

We also investigated the existing bottleneck in the network data path of DeLiBA2 that prevent even higher improvements of the latency results. Ceph block I/O requests are translated into network operations that are eventually executed on the OSD server in an often log-structured storage backend. Similar to other log-structured data structures as used in databases, this backend requires a costly compaction (a.k.a. garbage collection) process to remove outdated data.

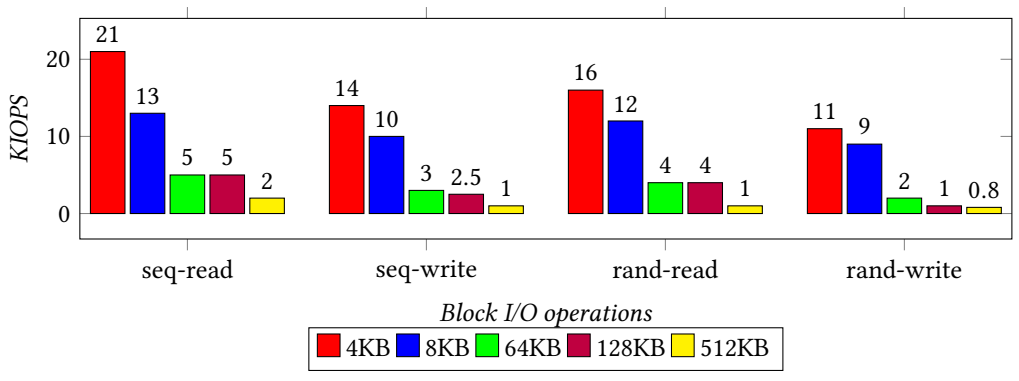
During this garbage collection, the performance of the OSD drops significantly, especially for workloads that include small block I/O requests such as 4 kB, as discussed in Section 6. Moreover, Ceph performs numerous compare-and-swap (read-modify-write) operations that both read a location and write a new value into it simultaneously, either with a completely new value, or as some function of the previous value. Each operation in read-modify-write requires an extra PCIe round-trip communication to read the current value, perform the modification, and write the new value back to the specific location. The time taken for these network interactions has an impact on the overall latency results shown in Section 6.

Throughput Speedups using Hardware Acceleration. As Figure 11.a and Figure 12.a show, throughputs are also improved, showing the 4 kB, 8 kB, 64 kB, 128 kB and 512 kB block sizes as examples. For *replication-mode* operation, the hardware-accelerated DeLiBA2 solution manages to speed-up the throughput by up to 2.81x for sequential writes of 4 kB blocks, and by 1.16x for sequential writes of 128 kB blocks. For *erasure-coding* mode, the highest throughput gains of 2.42x are also achieved for sequential writes of 4 kB blocks, while the larger 128 kB blocks at best realize a throughput improvement of 1.24x for random writes. As for the software case, Table 4 condenses the measurements for the "other" block sizes that would not fit into the bar charts. Table 5 shows the standard deviations for all of these throughput measurements.

I/O Rate Speedups using Hardware Acceleration. The gains are similar for the rate of I/Os per second, shown in Figure 11.b and Figure 12.b. For *replication*, the largest gain of 2.8x for 4 kB blocks



(a) Throughput



(b) KIOPS

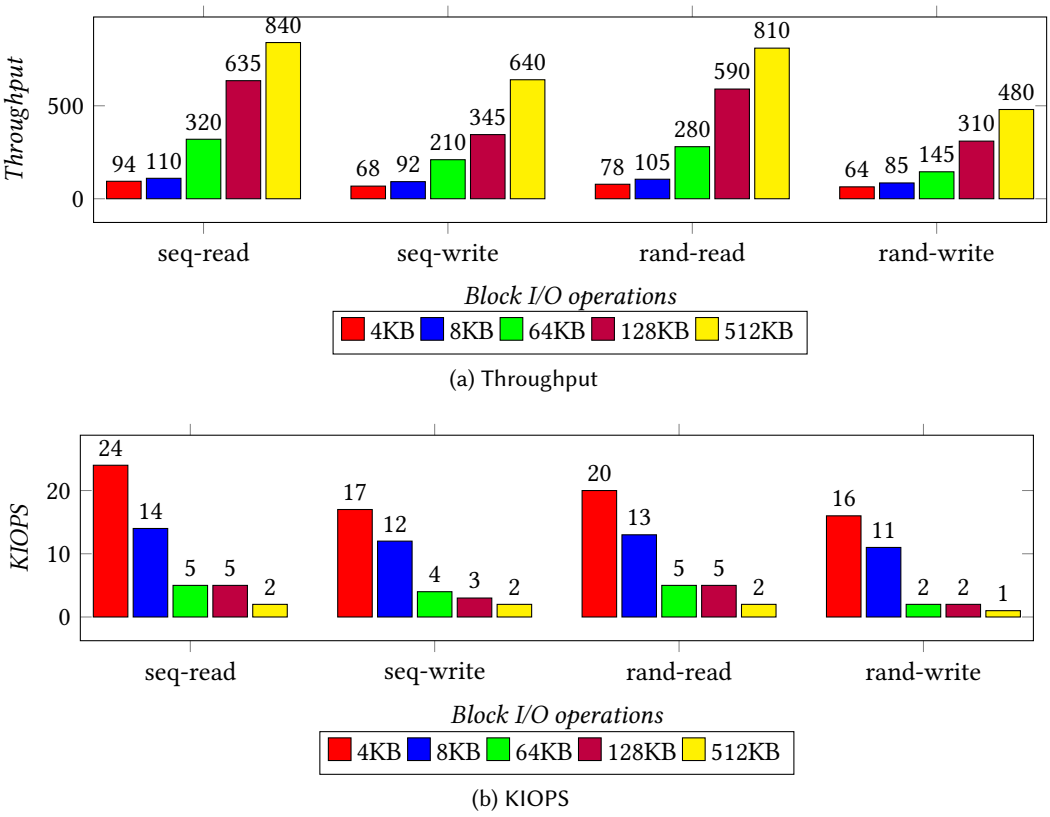
Fig. 11. Throughputs and I/O rates for Ceph replication mode with DeLiBA2 hardware acceleration

Table 4. Throughputs achieved using DeLiBA2 acceleration for "other" block sizes

Block Size	Request Patterns			
	seq-read	seq-write	rand-read	rand-write
16 kB (replication)	121 MB/s	88 MB/s	75 MB/s	64 MB/s
32 kB (replication)	273 MB/s	172 MB/s	218 MB/s	94 MB/s
256 kB (replication)	719 MB/s	623 MB/s	567 MB/s	432 MB/s
16 kB (EC)	132 MB/s	89 MB/s	106 MB/s	68 MB/s
32 kB (EC)	252 MB/s	145 MB/s	182 MB/s	101 MB/s
256 kB (EC)	884 MB/s	603 MB/s	661 MB/s	491 MB/s

is achieved for sequential writes. For the larger 128 kB blocks, the I/O rate will naturally be slower due the longer time required to actually transfer the larger payload over the network. But still, the hardware accelerated stack manages a gain of 1.5x for the sequential-write case.

For *EC* operation, the I/O rate for 4 kB is improved by 2.42x for sequential-writes. Similar to replication mode, IOPS gains are limited for the larger 128 kB blocks. At best, a factor of 1.5x is



(a) Throughput

(b) KIOPS

Fig. 12. Throughput and I/O rates for Ceph in erasure-coding mode with DeLiBA2 hardware acceleration

achieved, again for sequential writes. For both throughput and I/O rate, the speedup factors are quite similar.

Overall throughput and IOPS gains for synthetic fio workloads. Figure 13a and Figure 13b visualize the overall throughput speedup graph of each block I/O size against each block I/O request pattern in DeLiBA2. The highest speedups were achieved in small block IO sizes i.e. 4 kB and 8 kB. Larger block I/O sizes 64 kB, 128 kB and 512 kB had smaller speedups, but never slow downs.

Currently DeLiBA2 employs an LRU caching mechanism to enhance seq read/write performance. As we concentrated on hardware acceleration, we did not attempt to fine-tune these software caching/eviction policies. However, it is likely that the lower performance, especially for larger block sizes, can be improved by better optimization of the policies, especially w.r.t. block eviction.

Wall Clock Performance on Real Workload. Using our real workload of compiling Ceph itself on a CephFS volume, the use of DeLiBA2 hardware acceleration leads to an average wall clock execution time of 110.2 seconds (min. 108s, max. 117s), again flushing the filesystem cache between executions. Compilation is a mix of read/write operations and consists mostly of random accesses, as only the linking step performs larger sequential accesses. Using our hardware acceleration even for this real mix of compute (GCC) and I/O (Ceph) leads to an average wall clock speedup of roughly 20%.

Table 5. Standard deviations of all throughputs for DeLiBA2 accelerated *fio* tests

Block I/O Size	Block I/O Request Patterns			
	seq-read	seq-write	rand-read	rand-write
4 kB (replication)	0.564	0.685	0.982	0.776
8 kB (replication)	0.823	0.661	0.945	0.823
16 kB (replication)	0.999	0.443	0.865	0.899
32 kB (replication)	0.752	0.812	0.934	0.555
64 kB (replication)	0.499	0.687	0.912	0.367
128 kB (replication)	0.911	0.333	0.589	0.989
256 kB (replication)	0.543	0.794	0.912	0.845
512 kB (replication)	0.888	0.435	0.765	0.686
4 kB (EC)	0.678	0.579	0.871	.812
8 kB (EC)	0.712	0.651	0.912	0.912
16 kB (EC)	0.829	0.688	0.811	0.788
32 kB (EC)	0.814	0.919	0.811	0.601
64 kB (EC)	0.515	0.699	0.888	0.912
128 kB (EC)	0.812	0.715	0.888	0.912
256 kB (EC)	0.699	0.912	0.712	0.666
512 kB (EC)	0.899	0.997	0.881	0.779

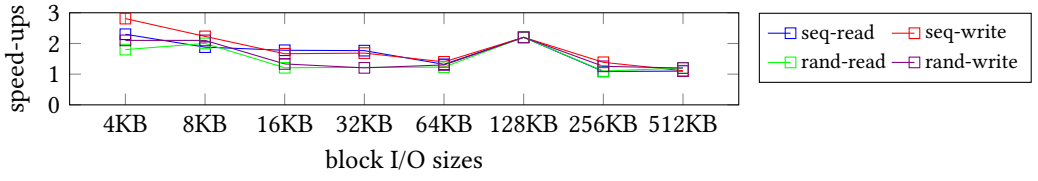
Table 6
Total Resource Utilization of DeLiBA2, including the 10G TCP/IP Stack, configured for a specific replication or EC mode

kernel + TCP/IP + CMAC	**LUTs**	**Block RAM (BRAM)**	**UltraRAM (URAM)**	**DSPs**
Straw Bucket	68,467	112 (5.5%)	0	0
Straw2 Bucket	68,544	102 (5.1%)	0	0
List bucket	64,688	90 (4.4%)	0	0
Tree bucket	67,982	98 (5.9%)	0	0
Uniform bucket	69,143	98 (5.9%)	0	0
Reed-Solomon Encoder	70,111	119 (5.9%)	0	0

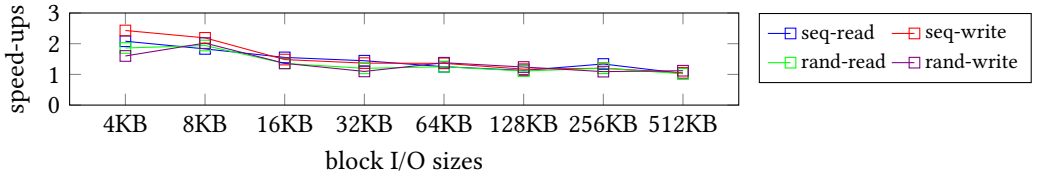
FPGA Resource Requirements. Table 6 shows the post place-and-route resource utilization of each hardware kernel listed in Table 2. For comparison [80], the 16nm Ultrascale Alveo U280 card contains an FPGA chip with 1.3 million LUTs and 2.72 million registers. With 2,016 Block RAMs (BRAMs) and 960 UltraRAMs (URAM), it holds 4.5 MB of on-chip (BRAM) and 30 MB of on-chip URAM. Furthermore, it has 9024 DSP slices. As can be seen, the complete system allowing the interaction with Ceph distributed storage in both replication as well as Erasure Code modes requires only very little FPGA area for 10G operation. It is thus highly promising to consider scaling the system up to *100G* operation, as sufficient capacity to handle the wider data busses and the additional storage I/O accelerator instances *is* available on the chip. This is one of the next goals discussed in Section 8.

7 RELATED WORK AND DISCUSSION

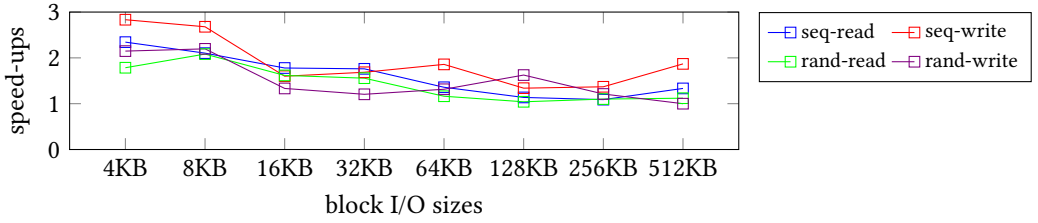
We organize our discussion of related work into two main topics. he first one takes a look at *distributed systems* and encompasses the "SmartNIC-based In-Network Processing" approach we follow with DeLiBA2, especially on prior work for replication and EC operations. The second one



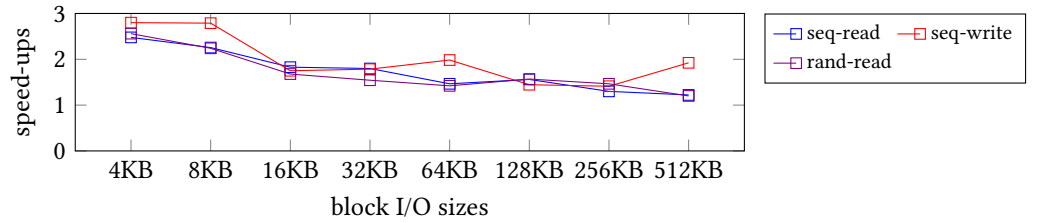
(a) throughput speedup using DeLiBA2 for each block I/O size in replication mode



(b) throughput speedup using DeLiBA2 for each block I/O size in erasure coding mode



(c) IOPS speedup using DeLiBA2 for each block I/O size in replication mode



(d) IOPS speedup using DeLiBA2 for each block I/O size in erasure coding mode

Fig. 13. DeLiBA2 framework throughput and IOPS speedups for synthetic *fio* workloads on Alveo U280 FPGA

deals with generally speeding up accesses to block storage, focusing on local (non-distributed) storage.

In-Network Replication and EC Acceleration. To begin with, the work in [18] is most closely related to DeLiBA2. It shows how storage policies can be offloaded to fully programmable SmartNICs. To this end, they have considered operations like replication and EC as use cases. However, there are two fundamental differences in their work as compared to ours. First, to accelerate the data path they have used remote direct memory access (RDMA) instead of TCP. Second, their work is more focused on file system operations instead of block storage. Apart from it, the replication schemes used in [18] are also different from the replication schemes used in DeLiBA2. Moreover, in their

experimental methodology, they have configured a toolkit to simulate operations on a 400 Gb/s network, as compared to our use of 10 Gb/s in DeLiBA2.

In [34] the authors have demonstrated that replication can be removed from the critical path of performance by moving it to FPGA hardware. As a proof of concept, they have implemented a consensus protocol roughly equivalent to Paxos, namely Zookeeper's atomic broadcast at the 10 Gb/s network level. However, this work falls more in the category of server side acceleration as they use key-value store in conjunction with the atomic broadcast module.

Furthermore, TriEC [61] and INEC [62] propose a new EC NIC offload strategies that overcome many limitations of current-generation NIC-offloaded schemes for EC. However, in contrast to our current client-side focus, both again fall in the server-side acceleration category, accelerating a memcached-based key-value store.

NetEC [54] is a relatively current work in the area of EC acceleration. But they have used an ASIC to accelerate the EC *decoder*, as compared to our FPGA-accelerated *encoder* proposed in DeLiBA2.

The work in [41] is the only work that has conducted a detailed study on the replication and EC operations in Ceph storage from a complete *system* perspective, similar to our own scope. Although it provides useful measurements, this work does not present any new framework to actually accelerate the slow operations in Ceph.

Local SSD acceleration. As a good introduction to the field, the work in [20] has performed an in-depth evaluation of all existing NBD frameworks and offers a good baseline for software-only benchmark. However, their scenario is far simpler than ours: They just use NBD to communicate with local drives via SCSI. Neither hardware acceleration, nor network communication, nor distributed storage protocols play a role in their work.

The authors in [69] do include an FPGA-based framework to achieve 5x higher IOPS and up to 71% lower latency. They follow an approach similar to SPDK [65] in that they combine a user-space NVMe stack with an NVMe target implemented in the FPGA. That NVMe target then forwards the requests to an NVMe SSD directly attached to the FPGA. The FPGA can then be used to perform optimized scheduling/re-queuing of the operations received from the user-space stack before handing them off to the actual storage device. Note that by necessity, the acceleration factors for the distributed storage scenario in our work will always be smaller than those possible for accelerating local storage accesses, as we will always incur the overhead of network transmissions, which can only rarely be reduced (e.g., by possibly moving from TCP/IP to QUIC, but which is out-of-scope for our work).

The approach in [81] goes even further than [69] in removing layers from the I/O stack. Here, for the purpose of high-speed data recording, the host and any software is avoided *completely*. Instead, an FPGA is used to receive multiple data streams arriving via high-speed optical ports, and then to realize an NVMe initiator in soft-logic that can write out that data with minimum latency to NVMe drives directly attached to the FPGA. However, the approach in [81] is NVMe specific, while DeLiBA2 interacts with the *generic* block-mq layer. [69] does not include network optimizations, it just focuses on local SSD operation.

The idea of using multiple queues, which also lies at the heart of the Linux block-mq I/O subsystem, to communicate with FPGAs has been examined by other authors as well. Both [58] and [57] employ it to interact with general-purpose FPGA accelerators, but not for realizing the complete storage solution we aim for.

As Ceph is a widely used storage system for HPC scenarios, there has also been commercial interest in providing accelerated solutions. A recent one is an offering by Xilinx [78], but the publicly available information does not show any benchmarks.

Also, all of these prior research efforts focused right away on end-to-end performance, often with excellent results. But the main design goal of DeLiBA and DeLiBA2 was different: Our system

was designed from the start as an *enabler* for basic research into the acceleration of more complex storage protocols, such as Ceph, than the computationally simpler SCSI and NVMe-based systems examined in prior work. The fact that DeLiBA, even in its initial version DeLiBA1, is already able to achieve performance gains, further improved by DeLiBA2, though, is a pleasant side-effect of the originally intended purpose.

8 CONCLUSION AND OUTLOOK

This work presents the most refined version of the DeLiBA framework for accelerating Linux block I/O operations by moving them to FPGA-based hardware. In its initial version DeLiBA1 [37], the framework was used to move the compute-intensive algorithms for replication-based storage in the Ceph system to FPGA execution. The new version DeLiBA2 presented here also allows erasure-coding based storage, and in addition addresses the *distributed* nature of a Ceph-based storage system by including communications acceleration on the FPGA. The latter encompasses both the Ceph-specific *Messenger* protocol, as well as the underlying TCP/IP layers and Ethernet interface.

By avoiding the overhead of extra context and HW/SW execution switches for the networking functions, we have improved both the throughputs as well as the latencies over the original DeLiBA1 implementation. But we continue to be able to provide a comfortable development environment for more complex storage algorithms by "lifting" their processing from the far more restrictive kernel programming level up into user space, where all of the techniques and tools familiar to software developers are fully applicable.

However, when aiming for maximum performance, this lifting from kernel to user-space becomes too costly. It is currently implemented by employing the Network Block Device (NBD) feature of the Linux kernel, which easily provides the required functionality, but causes slow context switches between the Linux kernel and the user-space processing for each of the I/O requests.

The next step is thus to *completely* avoid these unnecessary user-kernel mode switches we still incur, namely by removing the NBD-based indirection, and instead entirely operating at the kernel-level block I/O interface. For Ceph, this would be achieved by attaching the hardware-based processing (storage and network) to the existing Rados block device (RBD) module. In this manner, both user-kernel space transitions, as well as PCIe transfers between the host and the FPGA board can be reduced even further.

Since we want to preserve the capability of DeLiBA to act as a general-purpose development framework for adding hardware acceleration to the block I/O stack, we will aim to maintain compatibility between the DeLiBA2 user-space development environment, and the in-kernel execution of a future "DeLiBA3". This would enable storage experts to perform productive development in user-space as long as possible, and then descend into the kernel space with minimal extra effort when tuning for maximum performance.

Once the last of the unnecessary context and execution mode switches has been removed, further areas to be investigated is scaling-up the system to 100G operation and to also consider accelerating server-side operations.

DeLiBA is open source and available online at https://github.com/esa-tu-darmstadt/deliba

ACKNOWLEDGMENTS

This work has been co-funded by the German Federal Ministry for Education and Research (BMBF) for the project SODDAS with the funding ID 01 IS 19018 B.

REFERENCES

[1] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2020. The Case for Custom Storage Backends in Distributed Storage Systems. *ACM Trans. Storage* 16, 2, Article 9 (May 2020), 31 pages. https://doi.org/10.1145/3386362

[2] Jens Axboe. 2004. Linux block IO—present and future. In *Ottawa Linux Symp.* Linux Symposium, Ottawa, Ontario Canada, 51–61.

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).* IEEE, USA, 5–16. https://doi.org/10.1109/ANCS.2015.7110116

[4] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (mar 2017), 48–54. https://doi.org/10.1145/3015146

[5] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '13).* Association for Computing Machinery, New York, NY, USA, 10 pages.

[6] Peter Breuer, Andrés Marín-López, and Arturo Ares. 2000. The network block device. *Linux Journal* 73 (05 2000).

[7] André Brinkmann, Kay Salzwedel, and Christian Scheideler. 2000. Efficient, Distributed Data Placement Strategies for Storage Area Networks (Extended Abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures* (Bar Harbor, Maine, USA) *(SPAA '00).* Association for Computing Machinery, New York, NY, USA, 119–128. https://doi.org/10.1145/341800.341815

[8] Andrei Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1 (11 2003). https://doi.org/10.1080/15427951.2004.10129096

[9] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21).* Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[10] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *SIGCOMM '22: Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22).* Association for Computing Machinery, New York, NY, USA, 767–779. https://doi.org/10.1145/3544216.3544230

[11] Blake Caldwell. 2015. Improving Block-level Efficiency with scsi-mq. *CoRR* abs/1504.07481 (2015), 06 pages. arXiv:1504.07481 http://arxiv.org/abs/1504.07481

[12] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. 2000. PVFS: A Parallel File System for Linux Clusters. In *4th Annual Linux Showcase & Conference (ALS 2000).* USENIX Association, Atlanta, GA, 1–12. https://www.usenix.org/conference/als-2000/pvfs-parallel-file-system-linux-clusters

[13] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, Atlanta, GA, USA, 385–395. https://doi.org/10.1109/MICRO.2010.33

[14] Ceph. 2009. block device RBD kernel driver. https://github.com/torvalds/linux/blob/master/drivers/block/rbd.c. (Last accessed: 2022-11-29).

[15] Ceph. 2016. ERASURE CODE. https://docs.ceph.com/en/latest/rados/operations/erasure-code. (Last accessed: 2022-11-29).

[16] Ceph. 2022. Ceph tool. https://docs.ceph.com/en/latest/dev/dev_cluster_deployement/. (Last accessed: 2021-12-16).

[17] Ceph development. 2022. vstart tool. https://github.com/ceph/ceph/blob/master/src/vstart.sh. (Last accessed: 2021-12-16).

[18] Salvatore Di Girolamo, Daniele De Sensi, Konstantin Taranov, Milos Malesevic, Maciej Besta, Timo Schneider, Severin Kistler, and Torsten Hoefler. 2022. Building Blocks for Network-Accelerated Distributed File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)* (Dallas, Texas) *(SC '22).* IEEE Press, USA, Article 10, 14 pages.

[19] Richard Durstenfeld. 1964. Algorithm 235: Random Permutation. *Commun. ACM* 7, 7 (jul 1964), 420. https://doi.org/10.1145/364520.364540

[20] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. 2021. BDUS: Implementing Block Devices in User Space. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '21).* Association for Computing Machinery, New York, NY, USA, 11 pages.

[21] FIO. 2022. FIO tool source code. https://github.com/axboe/fio. (Last accessed: 2021-12-16).

[22] Nick Fisk. 2017. *Mastering Ceph.* Packt Publishing Ltd., Livery Place 35 Livery Street Birmingham B3 2PB, UK.

[23] NBD Github. 1999. Network Block Device). https://github.com/NetworkBlockDevice. (Last accessed: 2021-12-16).

[24] A. Goel, C. Shahabi, S.Y.D. Yao, and R. Zimmermann. 2002. SCADDAR: an efficient randomized technique to reorganize continuous media blocks. In *Proceedings 18th International Conference on Data Engineering*. IEEE, USA, 473–482. https://doi.org/10.1109/ICDE.2002.994760

[25] Marco Hartmann, Lukas Weber, Johannes Wirth, Lukas Sommer, and Andreas Koch. 2021. Optimizing a Hardware Network Stack to Realize an In-Network ML Inference Application. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, USA, 21–32. https://doi.org/10.1109/H2RC54759.2021.00008

[26] Z. He, D. Korolija, and G. Alonso. 2021. EasyNet: 100 Gbps Network for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE Computer Society, Los Alamitos, CA, USA, 197–203. https://doi.org/10.1109/FPL53798.2021.00040

[27] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. 2021. The TaPaSCo Open-Source Toolflow. *Journal of Signal Processing Systems* 93 (02 May 2021), 545–563. https://doi.org/10.1007/s11265-021-01640-8

[28] R.J. Honicky and E.L. Miller. 2003. A fast algorithm for online placement and reorganization of replicated data. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, USA, 10 pp.–. https://doi.org/10.1109/IPDPS.2003.1213151

[29] R.J. Honicky and E.L. Miller. 2004. Replication under scalable hashing: a family of algorithms for scalable decentralized data distribution. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*. IEEE, USA, 96–. https://doi.org/10.1109/IPDPS.2004.1303042

[30] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Berkeley, California, United States, 113–128. https://www.usenix.org/conference/osdi21/presentation/hwang

[31] IBM. 2022. General Parallel File System 4.1.0.4. https://www.ibm.com/docs/en/gpfs/4.1.0.4

[32] Intel. 2016. Intel(R) Intelligent Storage Acceleration Library. https://github.com/intel/isa-l. (Last accessed: 2022-12-03).

[33] Intel. 2022. Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. (Last accessed: 2022-01-16).

[34] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, USA, 425–438.

[35] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 489–502.

[36] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA) *(STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[37] Babar Khan, Carsten Heinz, and Andreas Koch. 2022. 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)* (Belfast, United Kingdom). IEEE, New York City, USA, 9 pages. https://www.esa.informatik.tu-darmstadt.de/assets/publications/materials/2022/2022_FPL_BK.pdf

[38] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. 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)*. USENIX Association, Denver, CO, 41–45. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim

[39] Kyusik Kim, Seungkyu Hong, and Taeseok Kim. 2020. Supporting the Priorities in the Multi-queue Block I/O Layer for NVMe SSDs. *Journal of Semiconductor Technology and Science: JSTS* 20 (02 2020), 55–62. https://doi.org/10.5573/JSTS.2020.20.1.055

[40] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA.

[41] Sungjoon Koh, Jie Zhang, Miryeong Kwon, Jungyeon Yoon, David Donofrio, Nam Sung Kim, and Myoungsoo Jung. 2019. Exploring Fault-Tolerant Erasure Codes for Scalable All-Flash Array Clusters. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (2019), 1312–1330. https://doi.org/10.1109/TPDS.2018.2884722

[42] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670988

[43] Runhui Li, Yuchong Hu, and Patrick P. C. Lee. 2017. Enabling Efficient and Reliable Transition from Replication to Erasure Coding for Clustered File Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2500–2513. https://doi.org/10.1109/TPDS.2017.2678505

[44] Zhike Li and Yong Wang. 2022. An adaptive read/write optimized algorithm for Ceph heterogeneous systems via performance prediction and multi-attribute decision making. *Cluster Computing* 26, 2 (2022), 22. https://doi.org/10.1007/s10586-022-03764-3

[45] Linux Weekly News (LWN). 2017. Kyber multiqueue I/O scheduler. https://lwn.net/Articles/720071/.

[46] Linux Weekly News (LWN). 2016. Linux BLK-MQ scheduling framework. https://lwn.net/Articles/708465/

[47] K.Z. Meth and J. Satran. 2003. Design of the iSCSI protocol. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.* IEEE, USA, 116–122. https://doi.org/10.1109/MASS.2003.1194848

[48] Myoungwon Oh, Jugwan Eom, Jungyeon Yoon, Jae Yeun Yun, Seungmin Kim, and Heon Y. Yeom. 2016. Performance Optimization for All Flash Scale-Out Storage. In *2016 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, USA, 316–325. https://doi.org/10.1109/CLUSTER.2016.11

[49] Myoungwon Oh, Jiwoong Park, Sung Kyu Park, Adel Choi, Jongyoul Lee, Jin-Hyeok Choi, and Heon Y. Yeom. 2021. Re-architecting Distributed Block Storage System for Improving Random Write Performance. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS).* IEEE, USA, 104–114. https://doi.org/10.1109/ICDCS51616.2021.00019

[50] Myoungwon Oh, Sejin Park, Jugwan Eom, Seungmin Kim, Sangjae Kim, Kang-won Lee, and Heon Y Yeom. 2018. LALCA: Locality-Aware Lock Contention Avoidance for NVMe-Based Scale-out Storage System. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, USA, 1143–1152. https://doi.org/10.1109/IPDPS.2018.00123

[51] Panasas. 2022. PanFS Parallel File System. https://panasas.com/panfs-architecture/panfs/

[52] James Plank, Scott Simmerman, and Catherine Schuman. 2008. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications Version 1.2. *Technical Report CS-08-627, University of Tennessee, 2008* 0 (01 2008), 128.

[53] J. S. Plank, K. M. Greenan, and E. L. Miller. 2013. *A Complete Treatment of Software Implementations of Finite Field Arithmetic for Erasure Coding Applications.* Technical Report UT-CS-13-717. University of Tennessee.

[54] Yi Qiao, Menghao Zhang, Yu Zhou, Xiao Kong, Han Zhang, Mingwei Xu, Jun Bi, and Jilong Wang. 2022. NetEC: Accelerating Erasure Coding Reconstruction With In-Network Aggregation. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2022), 2571–2583. https://doi.org/10.1109/TPDS.2022.3145836

[55] I. S. Reed and G. Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304. https://doi.org/10.1137/0108018

[56] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. 2019. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19).* Association for Computing Machinery, New York, NY, USA, 554–569. https://doi.org/10.1145/3341301.3359640

[57] Siavash Rezaei, Eli Bozorgzadeh, and Kanghee Kim. 2019. UltraShare: FPGA-based Dynamic Accelerator Sharing and Allocation. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig).* IEEE, USA, 1–5.

[58] Siavash Rezaei, Kanghee Kim, and Eli Bozorgzadeh. 2018. Scalable Multi-Queue Data Transfer Scheme for FPGA-Based Multi-Accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD).* IEEE, USA, 374–380.

[59] Rodrigo Rodrigues and Barbara Liskov. 2005. High Availability in DHTs: Erasure Coding vs. Replication. In *Peer-to-Peer Systems IV*, Miguel Castro and Robbert van Renesse (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 226–239.

[60] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL).* IEEE, Barcelona, Spain, 286–292. https://doi.org/10.1109/FPL.2019.00053

[61] Haiyang Shi and Xiaoyi Lu. 2019. TriEC: Tripartite Graph Based Erasure Coding NIC Offload. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19).* Association for Computing Machinery, New York, NY, USA, Article 44, 34 pages. https://doi.org/10.1145/3295500.3356178

[62] Haiyang Shi and Xiaoyi Lu. 2020. INEC: Fast and Coherent in-Network Erasure Coding. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20).* IEEE, USA, Article 66, 17 pages.

[63] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST).* IEEE, Incline Village, NV, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[64] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines.* IEEE, Vancouver, BC, Canada, 36–43. https://doi.org/10.1109/FCCM.

2015.12
[65] SPDK. 2022. SPDK source code. https://github.com/spdk/spdk. (Last accessed: 2021-12-16).
[66] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (San Diego, California, USA) *(SIGCOMM '01)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/383059.383071
[67] Ceph storage. 2022. Ceph. https://github.com/ceph/ceph
[68] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2018. FastPath: Towards Wire-Speed NVMe SSDs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Dublin, Ireland, 170–1707.
[69] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2020. FastPath_MP: Low Overhead & Energy-Efficient FPGA-Based Storage Multi-Paths. *ACM Trans. Archit. Code Optim.* 17, 4 (nov 2020), 23 pages. https://doi.org/10.1145/3423134
[70] The kernel development community. 2007. Linux Scheduler - CFS scheduler. https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c. (Last accessed on 2022-11-29).
[71] The Linux kernel documentation. 2014. BFQ (Budget Fair Queueing). https://www.kernel.org/doc/html/latest/block/bfq-iosched.html. (Last accessed on 2022-11-29).
[72] Valgrind. 2022. Valgrind Framework. https://valgrind.org/. (Last accessed: 2022-01-16).
[73] Vitis. 2020. Vitis_with_100Gbps_TCP-IP. https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP. (Last accessed: 2022-12-03).
[74] Li Wang and Yunchuan Wen. 2016. Design and Implementation of Ceph Block Device in Userspace for Container Scenarios. In *2016 International Symposium on Computer, Consumer and Control (IS3C)*. IEEE, Xi'an, China, 383–386. https://doi.org/10.1109/IS3C.2016.105
[75] Hakim Weatherspoon and John Kubiatowicz. 2002. Erasure Coding Vs. Replication: A Quantitative Comparison. In *First International Workshop on Peer-to-Peer Systems (IPTPS '01)*. Springer-Verlag, Berlin, Heidelberg, 328–338.
[76] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (Tampa, Florida) *(SC '06)*. Association for Computing Machinery, New York, NY, USA, 122–es. https://doi.org/10.1145/1188455.1188582
[77] Yunxiang Wu, Fang Wang, Yu Hua, Dan Feng, Yuchong Hu, Wei Tong, Jingning Liu, and Dan He. 2017. I/O Stack Optimization for Efficient and Scalable Access in FCoE-Based SAN Storage. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2514–2526. https://doi.org/10.1109/TPDS.2017.2685139
[78] Xilinx. 2021. Xilinx revolutionizes the modern data center with software defined hardware accelerated alveo smartnics. https://www.xilinx.com/news/press/2021/xilinx-revolutionizes-the-modern-data-center-with-software-defined-hardware-accelerated-alveo-smartnics.html. (Last accessed: 2022-12-03).
[79] Xilinx. 2022. Xilinx Vitis HLS. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1399-vitis-hls.pdf. (Last accessed: 2022-01-16).
[80] AMD Xilinx. 2023. Alveo U280 Data Center Accelerator Card Data Sheet (DS963). https://docs.xilinx.com/r/en-US/ds963-u280/Alveo-Product-Details. (Last accessed: 2023-06-01).
[81] Jingchao Zhang, Fankuo Meng, Liyan Qiao, and Kaihui Zhu. 2019. Design and Implementation of Optical Fiber SSD Exploiting FPGA Accelerated NVMe. *IEEE Access* 7 (2019), 152944–152952. https://doi.org/10.1109/ACCESS.2019.2947181