DeLiBA: An Open-Source Hardware/Software Framework for the Development of Linux Block I/O Accelerators

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

***Abstract*—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 a number of 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 aims to address 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.**

While the current version of DeLiBA is focused on enabling more productive research instead of on raw performance, even in its current form it achieves 10% higher throughput and up to 2.3x the I/Os per second for a proof-of-concept Ceph accelerator realized using the system. These initial results show the large potential of performing further research in this acceleration domain.

I. 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 [1] for block storage, but also highly scalable parallel file systems such as GPFS [2] and PanFS [3]. Some storage systems, such as the Ceph [4]–[6] 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, there appears to be potential to employ hardware acceleration in the storage I/O stack, in a similar fashion as has been done successfully for a number of years in the high-speed networking I/O space.

However, only limited prior work has been performed in this area (see Section VI 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.

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 up 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 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 signalling, 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 remainder of this paper is organized as follows. In Section II, we give an introduction to the existing Linux block layer and point out some of the performance bottlenecks. Section III describes the software architecture of our DeLiBA framework, while Section IV introduces the hardware interface. As a first use-case for DeLiBA, we have implemented a proof-of-concept of an I/O accelerator for the client side of the Ceph storage protocol. Thus, Section IV discusses some key Ceph operations and their hardware design. Section V presents the results of an initial performance evaluation of the accelerator. We close with a discussion of related work in Section VI and conclude in Section VII, also looking forward to future work.

II. Revisiting Linux Block I/O Layer

The Linux block layer is a kernel subsystem that is responsible for handling block devices, e.g., hard disk drives (HDDs), solid state disks (SSDs), and remote storage (SAN) [7]. Applications submit I/O operations (hereafter: I/Os) via kernel system calls (`sysread()`/`syswrite()`), and are represented by 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

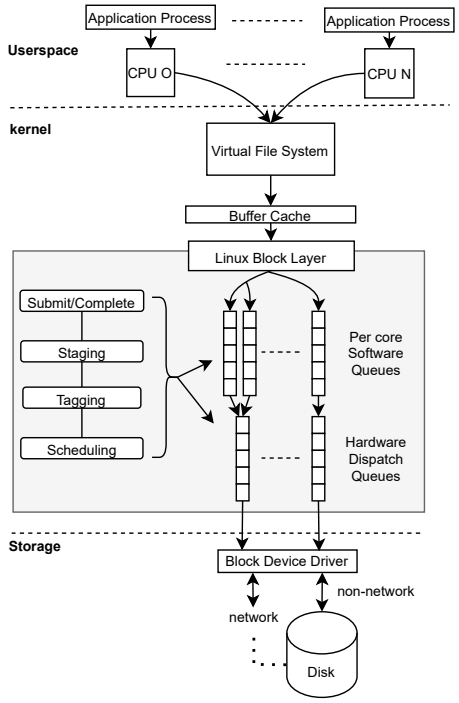


Fig. 1: Existing Linux Block Layer

years, the block layer has undergone a major change to move from a single request queue to a multi-queue model [8], 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 [9]. 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, with the aim to reduce the lock contention with a single request queue. Whereas the *hardware queues* provide a second level of buffering. Using these hardware queues, **`bio`** requests scheduled for dispatch are not sent directly to the device driver, they are instead sent to the 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 PCI Express (PCIe) interface used to communicate with actual hardware (storage or network devices). However, consensus is growing in the storage community that even with these improvements, the block layer has not kept up with novel storage hardware [10], [11]. We will analyze some of the deficiencies and make a case for FPGA acceleration of selected parts of the stack.

A. Limitations of the Current Storage Stack

To start with the latency bottleneck, as Figure 1 shows, there is a deeply layered kernel hierarchy to translate 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 $4\,\mathrm{kB}$ I/O request [12]. In x86 systems, around $50\,\%$ of the total execution time of a single $4\,\mathrm{kB}$ I/O request is spent in submitting and completing I/O requests at the kernel stage [13]. Whereas for 32-bit ARM Cortex A9 processors, the overhead of a $4\,\mathrm{kB}$ I/O reaches $90\,\%$ of the total execution time, with storage device latency only adding $10\,\%$ [14].

As a further complication, the block I/O scheduler(s) [15]–[17] are also sometimes at odds with the CPU scheduler [18]. This leads to I/O bound processes not receiving sufficient CPU time to actually execute at their desired priority [19], 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 [20], and suggest 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 in 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.

III. DeLiBA 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 [21]–[23]. Note that the use of nbd does carry a performance penalty, as the additional user-kernel space switches take time. However, as our 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 hardware acceleration (Fig. 4 and 5). Figure 2 sketches the main architecture of our NBD-based framework. Note that the figure also shows our first use-case for DeLiBA, namely the acceleration of the distributed *Ceph* storage protocol, and thus includes network communication from the Ceph client (left side) to the Ceph server (right side). For purely local use of DeLiBA, this extra complexity would not be required.

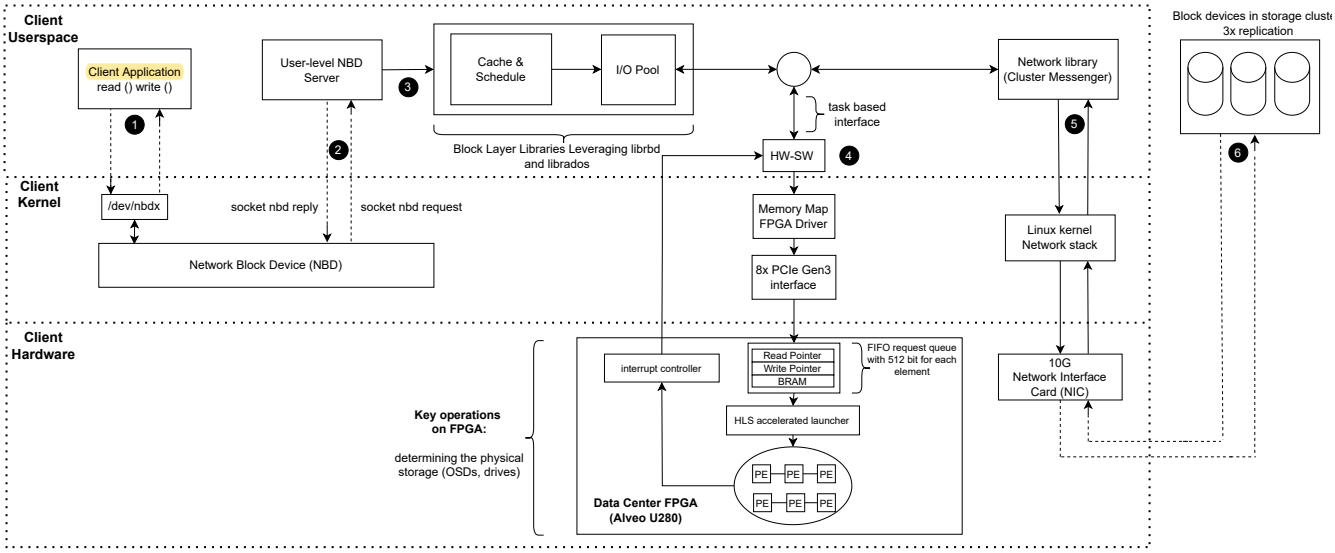


Fig. 2: Design and implementation of our framework

The focus of Figure 2 is an initial overview of DeLiBA and its system integration, later sections will provide more details. In the example shown, a client application generates a read operation ❶, which would often pass through filesystem layers (not shown here), and end up at the driver responsible for the device the data is stored on. For our purposes, that physical storage is represented by the nbd driver, which redirects the I/O requests back up into user-space ❷ using a netlink interface. Ceph then provides a number of software support functions (**`librados`**,**`librbd`**) to perform the initial steps of I/O processing ❸, similar to the approach that would be used in kernel-level. After passing through these Block Layer Libraries, requests would continue to be processed in software. However, with DeLiBA, we open up an alternative, namely the choice to route the requests toward an FPGA accelerator ❹ for further processing. In both cases, as Ceph is a distributed filesystem, we have to communicate over the network ❺ after completing client-side processing. Currently, for both software and hardware-based I/O processing, we rely on the Linux kernel network stack for this operation, which in turn interacts with the actual Network Interface Card (NIC) to communicate with the remote Ceph storage ❻.

A. 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 mechanism to manage and optimize I/O operations. But we can achieve a similar functionality by providing similar mechanisms in user-space. 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.

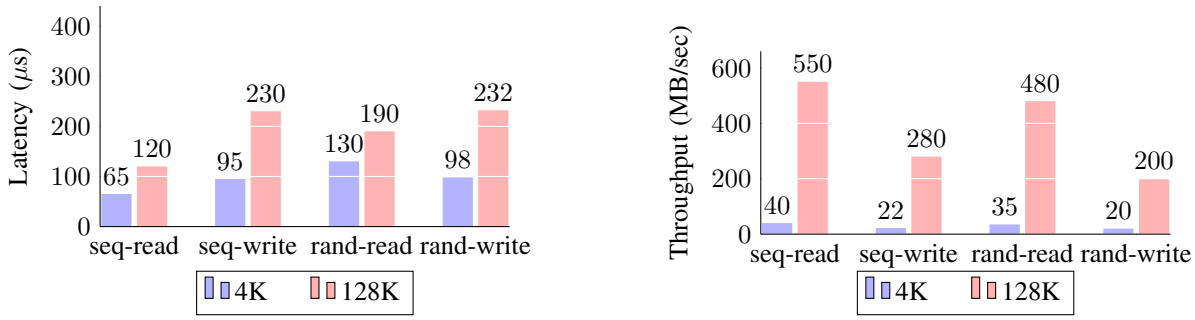
First, as our user-space approach cannot benefit from the kernel's page cache, we employ the **`librbd`** library to realize our own caching facility. We employ the Least Recently Used (LRU) replacement strategy and use a default cache size of 32 MiB. In addition to caching, we also employ the facility 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. In a later step, special care will be taken to transfer all the coalesced requests in a single PCIe access to the FPGA for handling.

B. Advanced Request Handling

After 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 our request cache. To support the asynchronous operations, DeLiBA extends each of the actual I/O primitives with a *completion*, indicating a callback function to be executed when the I/O operation has actually completed and 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 setup 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. This is computationally expensive, and thus (among other Ceph operations) an interesting candidate for hardware acceleration (see Section IV).



(a) Latency

(b) Throughput

Fig. 3: Latency and throughput of the software baseline of 4 kB and 128 kB

To allow the hardware-accelerated execution of storage operations, an I/O Pool is employed that is responsible for both launching the actual hardware tasks corresponding to the I/O operations, as well as tracking their completion status and performing the completion routines when done. Communication with the target FPGA, a Xilinx Alveo U280 card, occurs by memory-mapped I/O (control/status data) and DMA (bulk data) via a PCIe Gen3x8 interface.

As DeLiBA is intended as a research platform for block I/O acceleration, it currently relies on a software-based layer for networking. The I/O requests, coming either from software or from FPGA-accelerated processing, are then formatted into a Ceph-specific protocol called *Messenger*, which uses TCP to communicate with the remote Ceph storage cluster. Internally, the networking functionality supports multi-threaded execution by relying on a thread-pool of worker threads. The lower-level networking operations are then submitted to the kernel networking stack and NIC as usual.

C. Performance of DeLiBA in Software-Only Mode

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 a network testing tool namely, *iperf*. 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. The workloads were generated by the Flexible I/O (*fio*) tool [24]. 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 and 128 kB, we have chosen these two sizes for our experiments. Furthermore, 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 [25]. The tool generates a synthetic workload to benchmark a real data center cluster, and is part of the original source code [26].

a) Latency: Figure 3a shows the latency for block sizes 4 kB and 128 kB. With respect to Figure 2, the latency derives mainly from three stages: ❶ the block I/O request transmission from the client application to ❷ the nbd kernel client, then ❸ the userspace library and at the last stage the block I/O request is placed on the target block device in storage cluster. For comparison, when using Ceph without nbd, the latency measures is between 90 μs and 95 μs.

b) Throughput: Figure 3b shows the throughput for block sizes 4 kB and 128 kB. The maximum throughput achieved is for 128 kB sequential reads with 550 MB/s, and the maximum IOPS (I/O operations per second) are achieved for 4 kB sequential reads at 9,000 IOPS. For comparison, using Ceph without the nbd detour back to userspace achieves roughly 800 MB/s in this setup.

IV. Adding Hardware Acceleration to DeLiBA

The nature of block I/O operations fits well to 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, DeLiBA relies 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. For maximum throughput, DeLiBA employs a version of TaPaSCo extended with the hardware-assisted task launch mechanism described in [28], which aggregates multiple launch requests into a single PCIe transfer. In this manner, more than 6 million tasks can be launched per second, which is far higher than the I/O operation rates we currently require in current DeLiBA framework. As described in Section III-A, we provide the required caching and scheduling facilities in DeLiBA to prepare this aggregation of requests *before* performing the actual hardware task launches.

To evaluate the 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 I
SOFTWARE PROFILING RESULTS, VITIS 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	SLOCs SW	SLOCs HLS
Straw Bucket (pure HLS code)	85 μs	70% - 75%	133 - 135	0.665 μs - 0.675 μs	70 μs	256	148
Straw Bucket (using Vitis $\ln x$ function IP)	85 μs	70% - 75%	177 - 177	0.885 μs - 0.885 μs	70 μs	256	130
List bucket	65 μs	70%	56 - 56	0.280 μs - 0.280 μs	72 μs	197	134
Uniform Bucket	20 μs	50%	46 - 448	0.230 μs - 2.240 μs	25 μs	237	161
Tree Bucket	45 μs	75%	161 - 162	0.805 μs - 0.810 μs	45 μs	241	152

A. Hardware I/O Accelerator for Ceph Clients

The Ceph client was carefully profiled on a bare metal server using tools such as Intel VTune Profiler [29] and Valgrind [30] 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. Since Ceph is a distributed storage system, much effort is spent on guarding against data loss, for which two methods are employed: One is *replication*, where the same data is replicated in a distributed manner across multiple storage nodes, and the second is *erasure coding*, where mathematical techniques are employed to compensate for the partial loss of data. Replication carries a higher storage overhead, while erasure coding has a higher compute and memory cost.

Based on the profiling results in first two columns of Table I, we have focused on the acceleration of the compute-intensive replication algorithms in our proof-of-concept. For replication, Ceph relies on a pseudo-random data distribution algorithm [31]–[33] 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 [31] state 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 [34]. This heavy use of hashing has become even more intense in more recent Ceph versions: Our own 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 also to profile the overall performance choosing 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 actually vary with the configuration of the storage cluster as shown in Table I 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 *grows*, the **`list`** 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 static cluster, the **`straw2`** algorithm had the execution time of 85μs. In a cluster with more deeply nested cluster hierarchies the **`tree`** algorithm had an execution time of 45μs.

Furthermore, Table I shows another profile measurement termed as *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 relative runtime are promising for acceleration. Our goal was to select kernels contributing more than 50% to the overall Ceph runtime.

Both the measurements i.e *SW Execution Time* and *Overall contribution to runtime* are profiled based on a particular cluster size. 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.

Table I also shows the number of source-lines of code (SLOC) of the original software and the HLS implementation.

B. Implementing the Ceph Hardware Kernels

As shown in Table I, for our Ceph proof-of-concept I/O accelerator, *five* CRUSH key operations were moved from software to the hardware. The hardware kernels were implemented using the Xilinx Vitis 2021.2 High-Level Synthesis (HLS) tool [35], which at the time of writing is the latest version of the tool chain. The target FPGA board is a Xilinx Alveo U280, which carries an UltraScale+ FPGA. In HLS, we set a target clock frequency of 300 MHz, and all kernel individually achieve that. For the complete SoC on the FPGA (see Section IV-C), we achieve an $f_{\max}$ of 200 MHz after place-and-route. Table I shows the latency measurements in cycles and seconds for all five individual kernels at $f_{\max}$.

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, its more beneficial in terms of performance and area usage to *statically* tune individual kernels for concrete configuration parameter values, and instead employ separate, differently parameterised hardware kernels for different cluster configurations. All 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 pure HLS code version, we have 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 an 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 a 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 using 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 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 identification that is passed to the hash function as a parameter through the map. Similar to 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 IV-A, the *uniform kernel* performs replicas among (potentially large) groups of *identical* storage devices. The underlying computations are a Fisher–Yates shuffle [36], [37], computing a uniform 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.

In software profiling, we measured kernel runtimes between $20\,\mu s$ and $85\,\mu s$ on the host. Thus, the Vitis HLS estimates in Table I already indicate the potential benefits of accelerating core algorithms of modern storage stacks using reconfigurable computing at the post-P&R $f_{\max}$ of $200\,\mathrm{MHz}$, often being more than two orders of magnitude faster than executing the same algorithms on the host. However, as discussed in Section IV-C, the simplified programming environment provided by DeLiBA currently comes with a performance penalty of a similar order of magnitude, limiting the achievable wall-clock speedups.

C. Hardware Accelerator Integration

A key aspect for 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 $512\,\mathrm{kB}$ and thus is very amenable for a fast DMA transfer to the FPGA. Since the map is static as described in Section IV-A, it is transferred once and reused for the remaining I/O requests. 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 also be processed *independently*. This is very suitable for execution in a task-based model of computation as provided by the TaPaSCo FPGA framework [27].

For *each* of the different kernels, we configured ten hardware instances, called Processing Elements (PEs) in TaPaSCo. Our choice of ten PEs per kernel yields the optimal throughput: Fewer PEs would limit the degree of parallel requests, while more PEs lead to a further slow-down of the clock frequency of the resulting system-on-chip. With 10 PEs per kernel, we achieved a maximal clock frequency of $200\,\mathrm{MHz}$ for the entire FPGA.

We use the TaPaSCo scheduler to distribute the I/O requests from the DeLiBA I/O Pool to the actual hardware PEs, and transfer the results of finished hardware computation tasks back to their originating I/O requests in the I/O Pool in software.

These switches between software-hardware-software execution from the DeLiBA I/O pool, to the Ceph kernels on the FPGA, and back to the Ceph Cluster Messenger, carry significant overhead. Using the software-side C++ timing library **`#include <chrono>`**, we measured a full roundtrip, e.g., for the **`straw2`** kernel, to take around $60\mu s$ and $70\mu s$ (see Table I, last column). Thus, the execution switches almost cancel out the per-kernel speedups. However, since we can use *pipelined* execution with the hardware kernels, even with these high overheads, we *do* achieve the performance gains shown in Section V over pure software execution.

TABLE II
I/O REQUEST LATENCY ON SOFTWARE AND HARDWARE

SW vs HW (4 kB)	Latency [µs]			
	seq-read	*seq-write*	*rand-read*	*rand-write*
SW	65	95	130	98
HW	60	82	93	96

V. EXPERIMENTAL EVALUATION USING HARDWARE ACCELERATOR

As described above, DeLiBA is used to integrate a Xilinx Alveo U280 FPGA card providing accelerators for performing key computations of the Ceph protocol stack into the block I/O stack. The client side host uses an AMD EPYC Rome 7302P 16-core CPU with 128 GB of memory, attached by 10 Gb/s Ethernet to the Ceph server as described in Section III-C. The FPGA card is attached to the client host by PCIe Gen3 x8 and uses a system clock of 200 MHz.

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 DeLiBA 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 [38].

Even with the additional overheads due to 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 similar or even better than using the pure software stack, as shown in Table II. In this manner, the proof-of-concept Ceph accelerator we use to demonstrate DeLiBA in all but one case manages to *exceed* the baseline performance of the pure software implementation.

As shown in Figure 4, similar gains also apply to the throughput measurements for 4 kB and 128 kB block sizes, and sequential and random access patterns. The hardware-accelerated solution manages to speed-up throughput by up to 1.9x for sequential writes of 4 kB blocks, and by 1.2x for random writes of 128 kB blocks. The gains are even more pronounced for the rate of I/Os per second, shown in Figure 5. Here, the largest gain of 2.36x for 4 kB blocks is achieved for random reads. For the larger 128 kB blocks, the I/O rate will naturally be slower. But still, the hardware accelerated stack manages a gain of 1.13x for the random-read case.

TABLE III
TOTAL RESOURCE UTILIZATION

Resource	Used	Utilization (%)
Slices	37,576	35 %
LUTs	≈ 20,000	< 2%
DSPs	18	< 1.5%
BRAMs	48	< 2%

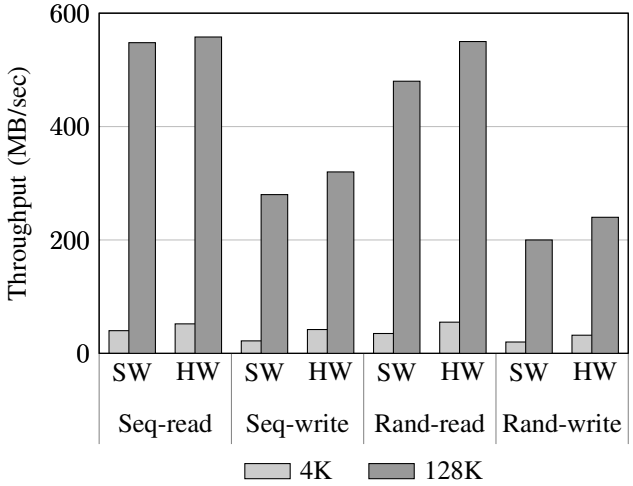


Fig. 4: Block I/O Throughput Hardware-Accelerated vs Software Baseline

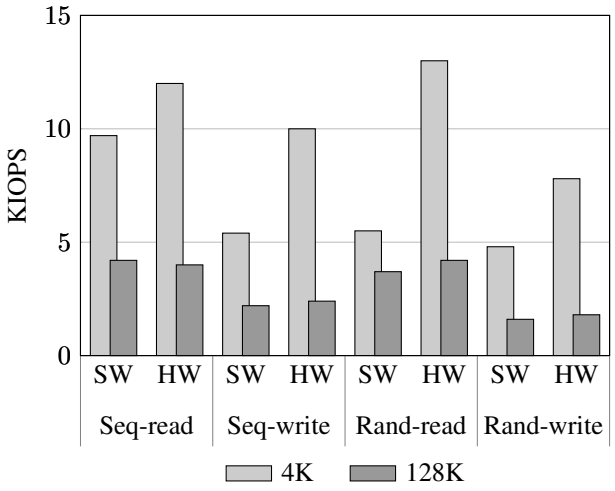


Fig. 5: IOPS on Hardware-Accelerated Stack vs Software Baseline

Table III shows the place-and-routed resource utilization. For a comparison the Alveo U280 card contains an FPGA chip with 1.08 million LUTs. Furthermore, it holds 4.5 MB of on-chip BRAM, 30 MB of on-chip URAM, and 9024 DSP slices. The DSPs are mainly used for division operations, as that is a frequent operation in the **`straw2`** kernel.

VI. RELATED WORK

Comparing our results with related work is somewhat difficult for two reasons: First, most of the related work is solely based on *local* SSDs, while we aim to speed-up a *distributed* system. Second, our proof-of-concept Ceph accelerator requires a specialized set of computations, e.g., replication coding for fault tolerance, that is not examined for acceleration in prior work.

However, we can discuss related work that at least partially overlaps our own efforts. To start with, the work in [39] 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 [40] do include an FPGA-based framework to achieve 5x higher IOPS and up to 71% lower latency. They follow an approach similar to SPDK [41] 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.

The approach in [42] goes even further than [40] 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.

The idea of using the abstraction 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 [43] and [44] 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 [45], 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 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. That DeLiBA is already able to achieve performance gains, though, is a pleasant side-effect of the originally intended purpose.

VII. Conclusion and Future Work

We have presented DeLiBA, an open-source framework for the development of hardware-accelerated block I/O stacks. DeLiBA provides a productive programming environment for the prototyping and functional testing of even complex storage protocols and can use powerful existing FPGA frameworks such as TaPaSCo to ease the integration of the actual hardware accelerators.

While DeLiBA can already enable some performance gains over the standard Linux block I/O stack, further improvements are already being worked on. Specifically, instead of using the nbd based relay, the hardware accelerators will be directly integrated with the *kernel-level* rbd module and communicate with the server via a hardware network stack directly integrated in the FPGA. In this manner, both user-kernel space transitions, as well as PCIe transfers between the host, the FPGA board, and the network interface card can be avoided. Also, hardware accelerators that perform Erasure Coding (EC), as an alternative to the current replication accelerators, will also be integrated.

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

Acknowledgment

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

References

[1] Y. Wu, F. Wang, Y. Hua, D. Feng, Y. Hu, W. Tong, J. Liu, and D. He, "I/o stack optimization for efficient and scalable access in fcoe-based san storage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2514–2526, 2017.

[2] IBM, "General parallel file system 4.1.0.4," https://www.ibm.com/docs/en/gpfs/4.1.0.4, (Last accessed: 2022-01-16).

[3] Panasas, "Panfs parallel file system," https://panasas.com/panfs-architecture/panfs/, (Last accessed: 2022-01-16).

[4] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "The case for custom storage backends in distributed storage systems," *ACM Trans. Storage*, may 2020.

[5] C. storage, "Ceph)," https://github.com/ceph/ceph, (Last accessed: 2021-12-16).

[6] Ceph, "block device rbd kernel driver," https://github.com/torvalds/linux/blob/master/drivers/block/rbd.c, (Last accessed: 2021-12-16).

[7] J. Suse, "Linux block io—present and future," *Proceedings of the Ottawa Linux Symposium 2004*, 01 2004.

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

[9] B. Caldwell, "Improving block-level efficiency with scsi-mq," *CoRR*, 2015.

[10] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. Association for Computing Machinery, 2014, p. 1–14.

[11] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, p. 48–54, mar 2017. [Online]. Available: https://doi.org/10.1145/3015146

[12] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[13] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim

[14] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján, "Fastpath: Towards wire-speed nvme ssds," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 170–1707.

[15] LWN, "Linux blk-mq scheduling framework," https://lwn.net/Articles/708465/, (Last accessed: 2021-12-16).

[16] Linux article, "Kyber multiqueue i/o scheduler," https://lwn.net/Articles/720071/, (Last accessed: 2021-12-16).

[17] The Linux kernel documentation, "Bfq (budget fair queueing)," https://www.kernel.org/doc/html/, (Last accessed: 2021-12-16).

[18] The kernel development community, "Linux scheduler - CFS scheduler," https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c, (Last accessed: 2021-12-16).

[19] K. Kim, S. Hong, and T. Kim, "Supporting the priorities in the multi-queue block i/o layer for nvme ssds," *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE*, 02 2020.

[20] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal, "Rearchitecting linux storage stack for µs latency and high throughput," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, july 2021.

[21] P. Breuer, A. Lopez, and A. G. Ares, "The network block device," 1999.

[22] N. Github, "Network block device)," https://github.com/NetworkBlockDevice, (Last accessed: 2021-12-16).

[23] L. Wang and Y. Wen, "Design and implementation of ceph block device in userspace for container scenarios," in *2016 International Symposium on Computer, Consumer and Control (IS3C)*, 2016.

[24] FIO, "Fio tool source code," https://github.com/axboe/fio, (Last accessed: 2021-12-16).

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

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

[27] C. Heinz, J. Hofmann, J. Korinth, L. Sommer, L. Weber, and A. Koch, "The tapasco open-source toolflow," *Journal of Signal Processing Systems*, May 2021. [Online]. Available: https://doi.org/10.1007/s11265-021-01640-8

[28] C. Heinz, J. A. Hofmann, L. Sommer, and A. Koch, "Improving job launch rates in the tapasco fpga middleware by hardware/software-co-design," in *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2020, pp. 22–30.

[29] Intel, "Intel vtune profiler," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.mkxn87, (Last accessed: 2022-01-16).

[30] Valgrind, "Valgrind framework," https://valgrind.org/, (Last accessed: 2022-01-16).

[31] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 31–31.

[32] R. Honicky and E. Miller, "A fast algorithm for online placement and reorganization of replicated data," in *Proceedings International Parallel and Distributed Processing Symposium*, 2003, pp. 10 pp.–.

[33] ——, "Replication under scalable hashing: a family of algorithms for scalable decentralized data distribution," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 96–.

[34] A. Broder and M. Mitzenmacher, "Survey: Network applications of bloom filters: A survey." *Internet Mathematics*, vol. 1, 11 2003.

[35] Xilinx, "Xilinx vitis hls," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1399-vitis-hls.pdf, (Last accessed: 2022-01-16).

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

[37] R. Durstenfeld, "Algorithm 235: Random permutation," *Commun. ACM*, vol. 7, no. 7, p. 420, jul 1964. [Online]. Available: https://doi.org/10.1145/364520.364540

[38] K. Meth and J. Satran, "Design of the iscsi protocol," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, 2003, pp. 116–122.

[39] A. Faria, R. Macedo, J. Pereira, and J. a. Paulo, "Bdus: Implementing block devices in user space," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR '21. New York, NY, USA: Association for Computing Machinery, 2021.

[40] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján, "Fastpath_mp: Low overhead & energy-efficient fpga-based storage multi-paths," *ACM Trans. Archit. Code Optim.*, nov 2020.

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

[42] J. Zhang, F. Meng, L. Qiao, and K. Zhu, "Design and implementation of optical fiber ssd exploiting fpga accelerated nvme," *IEEE Access*, vol. 7, 2019.

[43] S. Rezaei, K. Kim, and E. Bozorgzadeh, "Scalable multi-queue data transfer scheme for fpga-based multi-accelerators," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 374–380.

[44] S. Rezaei, E. Bozorgzadeh, and K. Kim, "Ultrashare: Fpga-based dynamic accelerator sharing and allocation," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, pp. 1–5.

[45] Xilinx, "Xilinx revolutionizes the modern data center with software defined hardware accelerated alveo smartnics," https://www.xilinx.com/news/press/2021/, (Last accessed: 2021-12-16).