BDUS: Implementing Block Devices in User Space

Alberto Faria, Ricardo Macedo, José Pereira, João Paulo
INESC TEC & University of Minho

ABSTRACT

Modern general-purpose operating systems implement major parts of their storage stacks in the kernel. Although this bolsters performance, it also complicates development and stifles innovation for today's increasingly complex storage systems. In contrast, implementing system services at the user level eases development and maintenance, and leads to improved portability, reliability, fault tolerance, and security. This has motivated the widespread use in both academia and industry of frameworks such as FUSE, which enable the implementation of file systems in user space.

In this paper, we consider user-level development at the storage stack's block layer. As many applications directly or indirectly rely on block devices to store data, this can promote and accelerate the construction of widely- and transparently-applicable storage solutions. Thus, we propose BDUS, a framework that enables the development of block device drivers in user space, and provide a fully-functional, open-source implementation for Linux.

An extensive evaluation of BDUS and comparable approaches shows that the former incurs less overhead on throughput and latency, while also reducing CPU utilization. File system stacks featuring BDUS are likewise shown to outperform stacks that employ the FUSE framework, considerably so under metadata-intensive workloads. BDUS is thus also a valuable tool for developing functionalities that may be built at both the block and file system layers.

This work was financed by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UTA-EXPL/CA/0080/2019 (Alberto Faria) and PhD grant SFRH/BD/146059/2019 (Ricardo Macedo), and realized within the scope of project BigHPC – POCI-01-0247-FEDER-045924 (João Paulo), funded by the ERDF – European Regional Development Fund, through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through FCT, I.P. within the scope of the UT Austin Portugal Program.

SYSTOR '21, June 14–16, 2021, Haifa, Israel

ACM ISBN 978-1-4503-8398-1/21/06…$15.00
https://doi.org/10.1145/3456727.3463768

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; • **Information systems** → **Information storage systems**.

KEYWORDS

Block storage, user-space drivers

ACM Reference Format:
Alberto Faria, Ricardo Macedo, José Pereira, João Paulo. 2021. BDUS: Implementing Block Devices in User Space. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3456727.3463768

1 INTRODUCTION

Storage services in modern general-purpose operating systems are generally implemented as part of the kernel. Compared to user-space services, this avoids the overhead of additional context switching and memory copies [27]. However, kernel-level development is notoriously complex and confined to an environment with limited functionality, which stifles research and innovation for the increasingly complicated storage systems of today. In contrast, building such services at the user level has several advantages: ease of development and maintenance, greater portability, access to more and higher-level languages and libraries, and improved reliability, fault tolerance, and security [20, 26, 27, 31, 32, 36, 37]. This has motivated the use of frameworks such as FUSE [13], which enables the implementation of file systems in user space, being widely employed both for experimentation and for developing full-fledged production file systems [32, 36].

In this paper, we consider user-level development at the storage stack's block layer of Unix-based systems. This layer provides access to block-based storage via the *block device* abstraction, which is ubiquitously used to expose local and remote storage devices (the latter through protocols such as iSCSI [19]). Many storage functionalities, such as compression, deduplication, thin provisioning, encryption, erasure coding, and replication, can also be implemented at this layer (as is done by, for instance, Device-mapper [1], DRBD [12], Dmdedup [33]). Since many applications either directly or indirectly (*e.g.*, through a file system) rely on block devices for storing data, storage solutions that provide a block device interface are transparently and widely applicable.

For a storage system to expose a block device interface, it must implement a block device *driver*. Although these reside

in the kernel, it is possible to delegate their development to user space by exploiting Network Block Device (NBD) [2], which provides access to remote storage devices through a local block device, or using Target Core Module in User space (TCMU) [9], which enables the implementation of custom SCSI targets in user space. However, unlike FUSE, whose performance has been extensively studied [29, 32, 36], these approaches have never been throughly evaluated.

Thus, as a first contribution, we describe the architecture and characterize the performance of four Linux frameworks that enable the development of block device drivers in user space: BUSE [6], nbdcpp [5], nbdkit [14], and tcmu-runner [16]. We find that, although existing solutions can maintain good performance under highly-batched or lower-throughput workloads, they incur significant overhead on both throughput and latency when under higher load. Moreover, they significantly increase CPU utilization under many workloads. We argue that a categorical solution for the user-level development of block device drivers should not hinge on protocols for remote storage access or storage stacks designed for other purposes, and that a clean-slate approach can unlock new optimizations and improvements.

Therefore, as a second contribution, we propose BDUS, a framework built specifically to enable the user-level development of block device drivers, and provide a fully-functional implementation for Linux. Its design strives to curtail memory copies and system calls, and unlike previous solutions it does not rely on protocols for remote storage access or on the SCSI stack. It also enables straightforward deployment of devices, and easy replacement of running drivers and recovery of failed drivers with no downtime. Further, evaluation of this framework shows that it consistently incurs less overhead on both throughput and latency than existing solutions, while consuming less CPU resources. BDUS is available as an open-source project at **https://github.com/albertofaria/bdus**.

In more detail, BDUS and the aforementioned frameworks are evaluated under a total of 41 micro and macro workloads operating both directly on block devices and on block device-backed file systems. Experiments are conducted under consumer-grade SATA SSDs installed in commodity machines and under enterprise NVMe SSDs installed in many-core server systems.

File system stacks featuring BDUS are further shown to outperform stacks that employ the FUSE framework. This is particularly evident under metadata-intensive workloads, where FUSE can increase latencies by an order of magnitude. Together with the fact that solutions exposing a block device interface are more widely applicable, and considering that this abstraction is notably simpler than the POSIX file system interface, these results affirm that BDUS is also a valuable tool for developing storage functionalities that may be constructed at both the block and file system layers.

2 USER-SPACE BLOCK DEVICE DRIVERS

We begin here by overviewing the block device interface and its internals under Linux, and then describe existing solutions for the user-level development of block device drivers.

2.1 Block devices

The block device interface is an abstraction present in Unix-based systems that provides access to storage devices and systems that transfer randomly accessible data in fixed-size blocks, presenting data as a contiguous byte sequence of predetermined size. Block devices can be used by operating systems services, such as file systems and paging, or directly by user-level applications through block special files typically made available under the `/dev` directory. In either case, clients operate on a block device by submitting requests to its driver, which implements the device's behavior. Requests can be of several types, three of which we describe here.

The most basic types of request are READ and WRITE, which allow clients to retrieve and store data at specific offsets in the device. The offset and size of all requests served by a block device must be aligned to its *logical block size*, the smallest size that the driver is capable of addressing.

The page cache, which caches data read from files and acts as a write-back cache for writes, applies to block devices. Thus, READ requests submitted by clients may be served without intervention of the block device driver, and WRITE requests may be delayed and coalesced. This has potential performance benefits and also enables applications to submit requests that are not aligned to the device's logical block size. Applications may nevertheless open block special files with the `O_DIRECT` flag to bypass this cache.

Block devices themselves may also feature internal write-back caches to improve performance, as is the case with many HDDs and SSDs. WRITE requests submitted by clients may thus be completed before the written data is persisted, even if the page cache is not being used. To ensure that written data is persistently stored and will survive crashes, clients may submit a FLUSH request.

Most drivers do not receive requests directly from clients. Instead, requests are first inserted into a per-core or per-NUMA node *software staging queue*, in which they may be reordered and merged prior to being moved to one of possibly several *hardware dispatch queues*, where they remain until serviced and completed by the driver [18].

2.2 Network Block Device

Although block device drivers reside in the operating system kernel, it is currently possible to exploit Network Block Device (NBD) [2], which provides access to remote storage devices through the block device interface, to effectively implement them in user space. NBD consists of (i) an in-kernel

client implementing a block device driver and (ii) a user-level server. Requests submitted to an NBD device are transmitted through a socket to the corresponding server, which then serves them from a local device. Thus, by building a server with custom request processing logic and deploying it in the same host as the client, one can effectively implement a block device driver in user space. Moreover, NBD can make use of Unix domain sockets instead of TCP sockets for its communication protocol, with the intent of improving the efficiency of such single-host deployments [2]. However, sockets always impose an extraneous memory copy to a kernel buffer of data transferred between client and server.

The BUSE [6], nbdcpp [5], and nbdkit [14] frameworks enable the creation of said custom NBD servers. BUSE in particular has the specific objective of enabling the development of user-level block device drivers, encapsulating the step of binding the client and server components that would otherwise be performed by the user. Servers built using nbdkit can process requests concurrently, and allow the client driver to establish several connections to parallelize request submission. In contrast, BUSE and nbdcpp are limited to processing requests sequentially through a single connection.

2.3 Target Core Module in User space

The SCSI storage protocol revolves around the concepts of a *target*—a service capable of handling SCSI commands—and an *initiator*—a client that submits such commands. Target Core Module (TCM) [15] is an in-kernel, Linux SCSI target consisting of (i) a *backstore* layer and (ii) a *fabric* layer. Backstores handle SCSI commands according to their specific implementation, such as by submitting them to a local SCSI device or serving data from a volatile RAM disk. Fabrics expose backstores to initiators through some particular protocol, such as iSCSI [19]. A *loopback* fabric is also provided, which exposes a backstore directly as a local block device.

TCM in User space (TCMU) [9] is a TCM backstore that delegates processing of SCSI commands to a user-level process. Communication between kernel and user space is accomplished through the UIO framework [10]. The tcmu-runner toolkit [16] encapsulates the user-level code associated with this mechanism, allowing the developer simply to define the logic to process each type of request.

By developing and instantiating a custom TCM backstore using tcmu-runner, and by exposing it locally as a block device using the loopback fabric, one can effectively use this infrastructure to develop block device drivers in user space.

3 BDUS: A FIRST LOOK

Our evaluation of the user-level block device driver development solutions described above, the results of which are presented in §6, shows that they incur significant overhead on throughput, latency, and CPU utilization under several workloads. Moreover, their dependence on the NBD protocol and Linux's SCSI target restricts possible improvements and optimizations. This prompted us to develop BDUS, a framework built from scratch to enable the implementation of block device drivers in user space. This section presents and explains the main functionalities provided by BDUS. Then, in §4, we detail BDUS' design and implementation.

Driver development interface. Using BDUS, block device drivers can be implemented as regular user-space applications. The driver development interface is provided by the *libbdus* library, a component of BDUS written in C99. To make a device available, a driver application invokes the `bdus_run()` function, which receives a value of type `struct bdus_ops` and another of type `struct bdus_attrs`.

In the first, the driver can specify pointers to functions that handle each type of block device request. As long as block device semantics are respected, these functions can be implemented in any desired manner (*e.g.*, by accessing a remote storage system, or reading and storing encrypted data on an underlying device). Drivers can also specify a function for handling custom `ioctl()` commands. If no handler for FLUSH requests is given, the driver is assumed to only complete WRITE requests after data is persistently stored. For other request types, by not specifying the respective function, the driver declares that it does not support them.

In the value of type `struct bdus_attrs`, the driver specifies the size and logical block size of the device, and optionally other attributes such as the maximum size of each type of request and the maximum number of threads that *libbdus* may employ when invoking request processing functions.

Device creation and destruction. When a driver is executed and invokes `bdus_run()`, BDUS creates a new device, assigns it a numerical identifier, and adds a block special file under `/dev` that refers to it (*e.g.*, `/dev/bdus-0`, for identifier `0`). From there on, whenever a request reaches the device, BDUS invokes the appropriate driver function to process it.

The `bdus` command-line tool can be used to destroy a device and terminate its driver. When, for instance, `bdus destroy /dev/bdus-0` is run, BDUS (i) submits a FLUSH request to the device, (ii) instructs the device's driver to terminate, at which point `bdus_run()` returns, and (iii) destroys the device and removes the associated block special file. A device is also automatically destroyed if the driver terminates abnormally (*e.g.*, by crashing).

Driver replacement and recovery. BDUS allows the driver of a device to be replaced without interruption of service, *i.e.*, without failing to handle any request. This can be useful to upgrade or reconfigure drivers of systems with strict availability requirements. To do this, the new driver can use the `bdus_rerun()` function, which behaves much like

`bdus_run()` but also accepts the identifier of an existing device. When called, BDUS flushes the specified device and terminates its current driver gracefully (as if through `bdus destroy`), but attaches the new driver to the device without destroying it. The new driver then begins processing requests as normal, and BDUS ensures that every request is handled by either the previous or the new driver.

A driver may also declare itself to be *recoverable*, in which case BDUS does not destroy its device if the driver terminates abnormally. Another driver may then attach to the existing device using the `bdus_rerun()` function. Requests not completed by the failed driver or submitted while there is no attached driver do not immediately fail, but instead block until a new driver is attached and processes them or until a configurable timeout elapses. Thus, if a driver is relaunched sufficiently quickly (*e.g.*, by a supervisor process), the device can recover transparently from its failure, without failing to handle any request. This feature is useful for deployments with stringent reliability and fault tolerance requirements.

4 BDUS: DESIGN AND IMPLEMENTATION

In the following, we detail BDUS' architecture, the device life cycle and how the driver replacement and recovery features are implemented, and the path taken by requests from their submission by a client to their handling by a user-level driver.

4.1 Architecture

As illustrated in Figure 1, BDUS consists of two main components: (i) the *kbdus* kernel module and (ii) the *libbdus* user-space library. *libbdus* relies on *kbdus*, and all communication between the two is accomplished through a *character device* created by *kbdus* and accessible via the `/dev/bdus-control` character special file. A third component, the `bdus` command-line tool (not shown in the figure), provides device management functionalities. It relies on *libbdus*, which offers the same functionalities programmatically.

The *libbdus* library provides a C99 interface, and bindings for other languages can be developed as libraries that rely on *libbdus* or that communicate directly with the *kbdus* module.

4.2 Device life cycle

A driver developed using BDUS creates a new device by invoking *libbdus'* `bdus_run()` function. When this occurs, *libbdus* first opens `/dev/bdus-control`. Then, it performs an `ioctl()` call on the resulting file description with the `CREATE_DEV` command. The configuration for the device is given as an argument to this command, and includes attributes specified by the driver in `struct bdus_attrs` and the set of supported request types. *kbdus* then validates this configuration, creates a new device, associates the aforementioned file description with it, and returns control to *libbdus*.

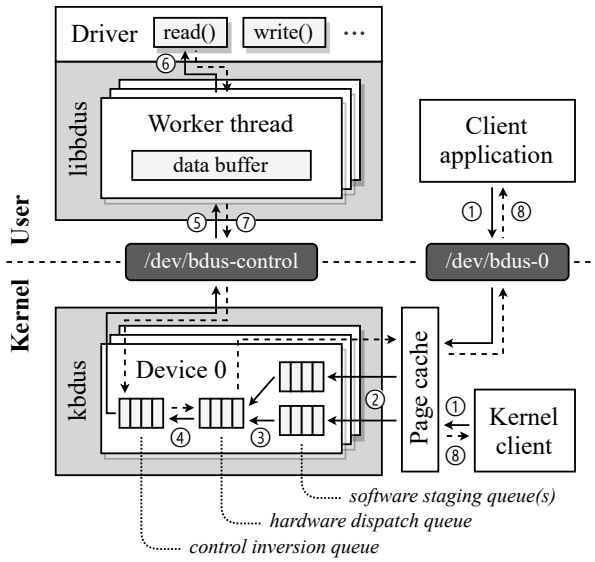

Figure 1: BDUS' architecture. Arrows show the path taken by READ requests submitted by user-level or kernel clients.

From this point on, *libbdus* continually retrieves requests through the same file description until it receives an indication to terminate. This mechanism and the path taken by requests, from when they are submitted by clients to when they are processed by the driver, are detailed further ahead.

When the driver is prompted to terminate, *libbdus* (i) instructs it to perform any necessary cleanup, (ii) performs an `ioctl()` with command `SET_SUCCESSFUL` on the aforementioned file description, and (iii) closes it, causing *kbdus* to destroy the device. The second step is relevant for recoverable drivers: if the driver crashes, the file description is implicitly closed without performing that step, in which case *kbdus* does not destroy the device but leaves it without a driver, allowing another to later attach to it. If a non-recoverable driver crashes, *kbdus* always destroys its device.

The procedure followed by `bdus_rerun()` is similar, with the exception that it performs the `ioctl()` call with command `ATTACH_TO_DEV`. Like `CREATE_DEV`, it takes as an argument the configuration for a device, and additionally the identifier of the device to attach to. *kbdus* then ensures that the given configuration is compatible with the existing device. Next, if the device already has a driver attached to it, *kbdus* submits a FLUSH request to that driver and instructs it to terminate. Once it does, or if no driver is attached to the device, *kbdus* associates with it the file description on which `ioctl()` was called and finally returns control to *libbdus*.

4.3 Request path

We now detail the path taken by requests submitted to block devices created by BDUS. For concreteness, we consider the steps taken by a READ request generated by an application,

as illustrated by the arrows in Figure 1 and identified with circled numbers. However, apart from different interactions with the page cache and direction of data transfers, requests follow the same sequence of steps regardless of their type.

First, an application performs a `read()` on the block special file of a BDUS device (*e.g.*, `/dev/bdus-0`), generating a READ request (①). If the client does not bypass the page cache and it contains the requested data, the request is served from that cache, otherwise it is submitted to the device's software staging queue for the current core or NUMA node (②).

After possible merging and reordering by the block scheduler configured for the device (if any), the request is moved to the device's single hardware dispatch queue, where it remains until completed, and the Linux kernel prompts *kbdus* to process it (③). Since the decision of when to move and begin processing a request is made by the Linux kernel, and because *kbdus* must wait for *libbdus* to be ready to receive it, a reference to the request is then inserted into a *control inversion queue* to be later retrieved by *libbdus* (④).

One of *libbdus'* worker threads performs an `ioctl()` on `/dev/bdus-control` with command `TRANSFER`, prompting *kbdus* to send the request at the head of the control inversion queue (⑤). The worker thread delegates processing of the request to the appropriate driver-specified function, which fills in a per-thread buffer with the requested data according to the driver's own logic and returns a success indicator (⑥).[1]

The worker thread performs another `ioctl()` with command `TRANSFER`, informing *kbdus* of the request's completion and blocking until the next request is received (⑦). Finally, *kbdus* copies the data in the thread's buffer to its final location (or to the page cache, if not bypassed) and informs the Linux kernel that the corresponding request in the hardware queue is completed, which in turn notifies the client (⑧).

In closing, we note that BDUS strives to minimize data copying between the client, kernel components, and user-level driver, as such operations can have a significant impact on performance and CPU utilization. Specifically, requests and replies are copied directly between the clients' buffers and *libbdus'* per-worker thread buffers. Further, by both submitting a reply and retrieving the next request with a single `TRANSFER` command, BDUS reduces the number of system calls required to process a request to only one.

5 METHODOLOGY

Having discussed the architecture of existing frameworks and BDUS' design and implementation, we now detail the methodology used to conduct their performance evaluation.

Evaluated systems. First, resorting to BUSE, nbdcpp, nbdkit, tcmu-runner, and BDUS, we developed pass-through block device drivers that simply redirect requests to an underlying block device (using direct I/O to avoid redundant caching) corresponding to a dedicated partition in a physical storage device. The nbdkit device was implemented as a C plugin and configured with 16 connections (truncated to the number of logical cores in the system) and 16 threads per connection (nbdkit cannot share threads among connections). Since BUSE and nbdcpp are single-threaded, they employed a single connection. In all cases, connections were established through Unix domain sockets. The TCMU device was implemented as a custom tcmu-runner handler employing 16 threads and exposed using the loopback fabric. The BDUS device was similarly set up with 16 worker threads.

We then evaluated the performance of both the underlying device and of each pass-through device under a variety of workloads, allowing us to measure the overhead imposed by each framework. Experiments were conducted on consumer-grade SATA SSDs installed in commodity machines and on enterprise NVMe SSDs in many-core server systems. All pass-through devices were configured without a block scheduler and the underlying device with its default scheduler (`mq-deadline` for SATA SSDs and none for NVMe SSDs).

Block device workloads. With the first of two groups of workloads, we evaluated the performance of operating directly on both the underlying block device and on each of the pass-through devices. These workloads employ either 1 or 16 threads, each reading or writing, sequentially or randomly, in blocks of 4 or 128 KiB, for at most 15 minutes. They operate on the first 64 GiB of the device on SATA SSDs and on the first 256 GiB on NVMe SSDs, with each workload thread performing I/O on a dedicated subrange of that area.

We use the following mnemonics to identify these workloads: *seq* and *rand* stand for sequential and random; *Xth* specifies that the workload employs X threads; and *Yk* indicates that I/O is performed in Y KiB blocks. For instance, workload *seq-write-16th-4k* employs 16 threads, each writing 4 KiB blocks sequentially. The fio [11] benchmarking tool was used to perform these workloads. Caches were purged in between runs, and the underlying partition was zero-filled before read workloads and trimmed before write workloads.

File system workloads. With the second group of workloads, we evaluated the performance of operating on file systems residing both in the underlying block device and in each of the pass-through devices. Additionally, we evaluated the performance of operating on a FUSE pass-through file system[2] that simply redirects requests to another file system backed by the underlying hardware device, allowing us to measure the overhead imposed by the FUSE framework.

[1] *kbdus* fully validates all interactions with the user-level driver, which cannot directly access any kernel structures or memory from other processes.

[2] We employed the `passthrough_hp.cc` example driver, "intended to be as efficient and correct as possible," from the FUSE user-space library: https://github.com/libfuse/libfuse/blob/fuse-3.9.3/example/passthrough_hp.cc

This is instructive as several storage functionalities such as compression, deduplication, thin provisioning, encryption, erasure coding, and replication may be implemented at both the block layer and the file system layer. In all cases, ext4 [28] was employed as the block device-backed file system due to its widespread use, and its lazy inode table and journal initialization features were disabled to improve result stability.

The file system workloads are divided into three classes: (i) *data-intensive micro workloads* perform I/O operations on one or more files and are otherwise identical to the block device workloads described previously, with each thread operating on a separate file; (ii) *metadata-intensive micro workloads* either create, read, or delete many 4 KiB files, employing 1 or 16 threads, and run for at most 15 minutes; (iii) *macro workloads* emulate a file server with 50 threads, a mail server with 16 threads, and a web server with 100 threads, all running for 60 minutes, and were taken from a previous study of FUSE's performance [36]. All workloads were performed by the Filebench [8, 34] tool. Caches were purged and the ext4 file system recreated in between runs.

Instrumentation and collected metrics. In addition to fio and Filebench, we used iostat [17] to observe CPU utilization and monitor several request processing-related metrics to aid analysis. All tools were configured to report performance and resource utilization every 5 seconds, and we consider only observations taken after the systems had reached steady state by manually discarding those in the warm up period.

Statistical method. We performed a minimum of 5 runs for each combination of underlying storage device, evaluated system, and workload. For each such group of benchmarking runs, the sample mean of each collected metric was calculated (disregarding warm up periods as described above) and Student's t-distribution was used to compute the 95% confidence intervals for the corresponding population means. Values presented later correspond to the aforementioned sample means, and the half widths of the respective confidence intervals are, unless otherwise stated, under 5% of the sample mean for throughput and latency, and under 5 percent points of the sample mean for CPU utilization.

Experimental environment. To evaluate performance under SATA SSDs, we used several identical machines with: one Intel Core i3-4170 CPU, with 4 threads; one 119 GiB, SATA-III, Samsung MZ7LN128 SSD. Performance under NVMe SSDs was evaluated using a machine with: two Intel Xeon Gold 6240 CPUs, each with 36 threads; one 1.46 TiB, NVMe, Dell Express Flash PM1725b SSD. Available RAM was limited to 4 GiB in all machines to accelerate cache warm up.

All systems were identically set up with Ubuntu Server 20.04.1 LTS and Linux kernel 5.8.9, and the following software packages were employed: NBD 3.20 [2], BUSE commit #b4a7f53 [6], nbdcpp commit #908f6b6 [5], nbdkit 1.22.1 [14], tcmu-runner 1.5.2 [16], BDUS 0.1.0, libfuse 3.9.3 [13], fio 3.23 [11], Filebench commit #22620e6 [8], iostat 12.4.0 [17].

6 EVALUATION

We now present the results obtained under both groups of workloads, and then discuss and summarize our findings.

6.1 Block device workloads

The throughput, latency, and CPU utilization achieved by the evaluated systems under block device workloads are presented in Table 1, each column corresponding to a combination of metric and system, and each row to a pairing of storage device and workload. Columns titled "Native" pertain to the setup in which workloads are performed directly on the underlying device. The nbdcpp framework never surpasses BUSE's performance, and as such is omitted. Cells for systems other than Native are colored according to their values (darker is worse), in particular with a white background if they show improvements of over 5% (throughput and latency) or 5 percent points (CPU utilization) over Native.

We begin by noting that the two storage device types give rise to distinct results. This is due to the NVMe device's higher absolute performance, as seen in the throughput and latency results for system Native. Additionally, the overhead on CPU utilization is generally of lower magnitude for experiments conducted on the NVMe device. This is because the system in which that device is installed provides 72 logical cores, while the machines containing SATA devices provide only 4, and CPU utilization is measured from 0% to 100%.

Regarding workloads *seq-read-1th-Xk* on SATA devices, we first observe that BUSE and nbdkit decrease throughput by between 16.7% and 21.4% (latency is affected reciprocally since these workloads are single threaded and closed loop), while TCMU and BDUS incur no perceptible overhead. On NVMe devices, BUSE, nbdkit, and TCMU decrease throughput by between 28.4% and 67.8%, and BDUS by at most 18.9%. Here, the difference in performance between BUSE and the remaining systems is explained by the operating system's *read ahead* optimization, which preemptively reads data that is likely to be requested next by the client. Even though these workloads are single threaded, this results in the concurrent submission of several READ requests, which BUSE cannot leverage due to being unable to process requests in parallel.

Under workloads *seq-read-16th-Xk*, nbdkit and BDUS do not impose any noticeable overhead on both storage device types, and TCMU degrades throughput (latency) only on NVMe by at most 15.5% (18.6%). In contrast, BUSE lowers throughput by up to 44.4% on SATA and 85.9% on NVMe, precisely due to only handling requests sequentially. Note that when compared to Native, BUSE decreases CPU utilization on NVMe due its significant overhead on throughput.

			Throughput (kop/s)					Latency (µs/op)					CPU utilization (%)				
			Native	BUSE	nbdkit	TCMU	BDUS	Native	BUSE	nbdkit	TCMU	BDUS	Native	BUSE	nbdkit	TCMU	BDUS
SATA	seq-read	1th-4k	133.18	-21.4	-16.9	-0.3	-0.5	7.31	+28.0	+20.8	-0.2	+0.3	9	+4	+7	+5	+2
		1th-128k	4.15	-21.0	-16.7	-0.4	-0.2	240.31	+26.7	+20.1	+0.4	+0.2	7	+2	+6	+3	+1
		16th-4k	135.13	-43.8	+0.2	+0.2	+0.2	118.21	+78.2	-0.2	-0.2	-0.2	9	+1	+10	+5	+3
		16th-128k	4.22	-44.4	+0.2	+0.2	+0.2	3790.68	+79.8	-0.2	-0.2	-0.2	6	+2	+10	+5	+4
	rand-read	1th-4k	9.84	-19.1	-30.7	-29.7	-12.6	100.88	+23.6	+43.8	+42.0	+14.3	3	+5	+9	+7	+2
		1th-128k	2.23	-11.7	-19.1	-17.3	-10.4	446.77	+13.4	+23.7	+21.0	+11.6	4	+5	+8	+4	+4
		16th-4k	88.05	-89.2	-28.6	-70.9	-11.2	180.97	+833.0	+38.7	+243.9	+12.3	24	-17	+65	+67	+45
		16th-128k	4.19	-43.3	+0.3	+0.4	+0.2	3814.47	+76.4	-0.3	-0.4	-0.2	7	+2	+11	+7	+4
	seq-write	1th-4k	37.92	-1.3	+0.6	-0.3	0.0	26.19	+1.2	-0.7	+0.3	0.0	1	+14	+5	+2	+2
		1th-128k	1.19	-0.7	+0.6	-0.1	0.0	843.81	+0.6	-0.6	+0.1	-0.1	1	+14	+5	+2	+2
		16th-4k	37.96	-0.7	+0.5	-0.3	+0.1	421.29	+0.7	-0.5	+0.3	-0.1	1	+14	+4	+2	+1
		16th-128k	1.19	-1.0	+0.7	-0.2	0.0	13486.57	+1.0	-0.8	+0.2	0.0	1	+13	+4	+2	+1
	rand-write	1th-4k	38.06	-26.6	+0.3	-2.6	0.0	25.93	+36.3	-1.1	+2.2	-0.2	4	+20	+31	+34	+14
		1th-128k	1.19	-1.1	+0.6	0.0	+0.1	842.74	+1.1	-0.6	0.0	-0.1	1	+13	+4	+2	+2
		16th-4k	38.01	-25.3	+0.7	-1.6	0.0	420.62	+33.8	-0.8	+1.6	0.0	4	+21	+30	+29	+14
		16th-128k	1.19	-0.8	+0.6	-0.2	+0.2	13506.25	+0.8	-0.6	+0.2	-0.2	1	+13	+4	+2	+1
NVMe	seq-read	1th-4k	332.24	-64.8	-28.4	-33.5	-16.4	2.82	+194.0	+43.3	+52.5	+21.6	2	0	+1	0	0
		1th-128k	11.25	-67.8	-30.5	-34.8	-18.9	88.59	+210.7	+44.0	+53.4	+23.4	2	0	+1	0	0
		16th-4k	822.17	-85.9	-2.2	-15.5	+1.6	19.09	+620.3	+2.3	+18.6	-2.0	11	-8	+10	+2	+2
		16th-128k	25.38	-85.7	-2.3	-14.3	+2.4	629.16	+602.6	+2.5	+16.7	-2.5	10	-8	+10	+3	+1
	rand-read	1th-4k	10.49	-24.7	-42.7	-58.0	-18.2	94.06	+33.3	+75.4	+139.8	+22.7	0	0	+1	0	0
		1th-128k	3.32	-33.9	-39.8	-42.4	-21.8	300.19	+51.7	+66.4	+73.7	+28.0	1	0	0	0	0
		16th-4k	146.20	-93.3	-42.1	-38.7	-28.5	107.87	+1403.4	+73.4	+63.9	+40.6	4	-3	+10	+5	+6
		16th-128k	23.80	-84.7	-13.4	-41.6	-2.5	670.74	+556.5	+15.4	+71.3	+2.4	9	-7	+7	+2	+2
	seq-write	1th-4k	407.41	-76.2	-79.6	-13.3	-13.7	2.30	+338.4	+411.0	+15.9	+16.8	3	+1	+1	+2	+2
		1th-128k	13.38	-77.4	-80.7	-3.4	-5.2	74.53	+346.8	+418.2	+3.4	+5.6	3	+1	+1	+2	+3
		16th-4k	402.29	-71.7	-79.3	-28.9	-19.9	39.38	+256.9	+387.8	+42.8	+27.9	7	-3	-4	0	+6
		16th-128k	12.72	-72.9	-80.4	-26.1	-11.3	1256.59	+271.9	+413.0	+37.6	+15.9	7	-3	-4	0	+7
	rand-write	1th-4k	282.71	-87.0	-89.0	-53.8	-38.4	3.25	+735.0	+871.9	+129.1	+67.6	3	-1	+1	+5	+6
		1th-128k	13.36	-72.6	-80.5	-5.0	-8.6	74.53	+269.8	+414.3	+5.1	+9.4	3	+1	+1	+2	+3
		16th-4k	227.56	-83.0	-85.7	-34.2	-34.4	69.84	+504.3	+608.4	+58.6	+55.3	16	-14	-11	-7	-2
		16th-128k	13.04	-72.8	-80.9	-31.6	-24.6	1225.79	+270.6	+427.2	+47.8	+34.8	7	-3	-3	+1	+4

Table 1: Block device workload results. Columns "Native" show absolute values; other columns show the relative difference as a percentage (for throughput and latency) or the absolute difference in percent points (for CPU utilization) against Native.

Under *rand-read-1th-Xk*, BUSE, nbdkit, and TCMU decrease throughput on SATA by up to 30.7%, and BDUS by at most 12.6%. Under *rand-read-16th-Xk*, BUSE degrades throughput (latency) against Native by up to 89.2% (833.0%), as the workloads are multi-threaded. Results for all systems under *rand-read-Xth-Yk* follow a similar trend on NVMe, with the exception of TCMU outperforming nbdkit under *rand-read-16th-4k* (but imposing more overhead than BDUS).

Under write workloads on SATA devices, nbdkit, TCMU, and BDUS have no perceptible impact on throughput and latency, due to asynchronous batching of WRITE requests by the page cache, and BUSE incurs overhead only under *rand-write-Xth-4k*, degrading throughput by up to 26.6% and latency by up to 36.3%. Further, BDUS increases CPU utilization only under *rand-write-Xth-4k* by 14 percent points (pp), while other systems increase it by up to 34 pp.

Write workloads on NVMe exhibited higher variability under all systems (including Native) than the results mentioned so far, with confidence interval half widths being under 10% of the sample mean for throughput and latency. We avoid making fine-grained observations on these results, but nevertheless note that TCMU matches or outperforms BUSE and nbdkit in throughput and latency under all such workloads, and BDUS in turn matches or outperforms TCMU.

6.2 File system workloads

Table 2 presents the throughput, latency, and CPU utilization attained by the systems under file system workloads and follows the same scheme as the previous table, with an additional column for the file system stack employing FUSE.

Data-intensive micro workloads. Generally, results obtained under these workloads on SATA devices are similar to those

			Throughput (kop/s)						Latency (µs/op)						CPU utilization (%)					
			Native	BUSE	nbdkit	TCMU	BDUS	FUSE	Native	BUSE	nbdkit	TCMU	BDUS	FUSE	Native	BUSE	nbdkit	TCMU	BDUS	FUSE
SATA	seq-read	1th-4k	133.01	-20.7	-2.1	0.0	+0.2	+0.1	7.52	+26.0	+2.2	0.0	-0.2	-0.1	14	+1	+15	+5	+1	+10
		1th-128k	4.13	-20.8	0.0	+0.7	+0.7	+1.4	241.87	+26.2	0.0	-0.7	-0.7	-1.4	8	+3	+12	+4	+1	+7
		16th-4k	133.97	-43.6	+0.3	+0.6	+0.5	+0.1	119.30	+77.4	-0.3	-0.6	-0.4	-0.2	12	0	+9	+6	+3	+10
		16th-128k	4.20	-43.6	+0.2	+0.4	+0.3	0.0	3809.50	+77.4	-0.2	-0.4	-0.2	0.0	6	+3	+9	+7	+4	+12
	rand-read	1th-4k	10.34	-18.6	-31.8	-32.2	-12.3	-20.9	96.73	+22.9	+46.6	+47.5	+14.0	+26.4	3	+4	+9	+8	+2	+4
		1th-128k	2.36	-10.6	-19.2	-19.4	-9.8	-19.1	423.39	+11.9	+23.8	+24.1	+10.9	+23.7	3	+5	+8	+5	+4	+5
		16th-4k	89.16	-88.9	-31.4	-72.2	-13.0	-28.6	179.39	+798.4	+45.9	+259.9	+15.0	+40.0	39	-31	+52	+51	+37	+46
		16th-128k	4.40	-43.5	+0.2	+0.2	+0.3	-2.1	3630.68	+76.9	-0.2	-0.2	-0.3	+2.2	7	+2	+11	+7	+4	+12
	seq-write	1th-4k	36.59	+1.5	+1.6	+2.5	+0.9	-0.3	26.85	+0.3	0.0	-1.1	+0.3	-0.1	1	+4	+4	+2	+1	+7
		1th-128k	1.14	+1.3	+1.4	+2.5	+0.9	-0.4	859.00	+0.4	0.0	-1.0	+0.3	-0.1	1	+4	+3	+2	+1	+5
		16th-4k	36.51	+1.4	+1.7	+3.1	+0.7	+0.7	431.78	0.0	-0.4	-1.9	+0.2	-0.4	1	+3	+3	+2	+1	+6
		16th-128k	1.14	+1.5	+1.6	+2.8	+0.9	+0.6	13797.02	+0.1	-0.1	-1.6	+0.2	-0.2	1	+3	+3	+2	+1	+4
	rand-write	1th-4k	37.81	-21.2	+0.4	-6.1	-0.4	-0.3	26.45	+26.9	-0.4	+6.5	+0.4	+0.3	5	+13	+33	+50	+17	+21
		1th-128k	1.17	+0.3	+0.5	+1.5	+0.1	+0.4	850.82	0.0	-0.5	-1.5	0.0	-0.5	1	+4	+4	+2	+1	+4
		16th-4k	37.55	-19.5	+0.4	-4.4	-0.4	+0.5	426.00	+24.2	-0.4	+4.5	+0.4	-0.5	5	+14	+30	+45	+15	+24
		16th-128k	1.17	+0.3	0.0	+2.0	-0.1	+0.7	13659.25	0.0	+0.1	-1.9	+0.2	-0.6	1	+3	+3	+2	+1	+2
	create-1th		22.71	+7.8	+2.6	+1.5	-0.7	-73.2	44.03	-7.2	-2.5	-1.5	+0.7	+273.3	21	+5	+4	+2	+1	+11
	create-16th		24.70	+3.7	+4.8	+3.9	+1.1	-59.2	595.39	-4.9	-2.3	-2.2	-0.4	+164.0	50	-3	-1	-1	-1	+9
	read-1th		16.80	-28.8	-29.6	-31.1	-16.8	-49.9	59.54	+40.4	+42.1	+45.0	+20.1	+99.7	6	+6	+10	+9	+5	+13
	read-16th		44.91	-65.5	-17.6	-30.5	-6.8	-44.6	345.17	+198.1	+17.6	+45.5	+6.3	+84.6	25	-13	+21	+23	+9	+25
	delete-1th		15.78	-10.7	-6.8	-5.0	-2.3	-50.2	63.41	+12.4	+7.5	+5.3	+2.5	+100.6	9	+8	+9	+8	+4	+18
	delete-16th		16.92	-15.1	-3.2	-3.7	-0.9	-48.4	900.88	+20.2	+3.9	+5.2	+1.8	+100.0	21	+5	+9	+7	+4	+15
	file-server		7.71	-42.4	-20.5	-24.0	-23.8	-5.5	5685.73	+96.0	+40.0	+47.4	+46.7	+13.4	6	+1	+5	+3	+2	+19
	mail-server		3.93	+4.9	+0.6	+2.4	+1.8	-6.6	4014.08	-4.7	-0.7	-2.4	-2.0	+7.0	3	+2	+3	+3	+1	+5
	web-server		63.16	-71.5	-6.9	-20.5	-4.1	-45.3	1195.56	+273.8	+0.3	+33.1	+9.8	+95.1	39	-27	+26	+20	+6	+34
NVMe	seq-read	1th-4k	213.51	-44.1	-34.0	-35.9	-18.1	-16.6	4.68	+79.0	+51.8	+56.0	+22.0	+19.9	1	0	+1	0	0	+2
		1th-128k	7.46	-51.9	-36.8	-42.4	-20.5	-20.2	134.15	+107.8	+58.1	+73.6	+25.8	+25.4	1	0	+1	+1	0	+2
		16th-4k	841.69	-86.3	-1.9	-30.9	-3.9	-1.9	19.00	+629.9	+2.0	+45.0	+4.1	+1.9	9	-6	+12	+5	+3	+9
		16th-128k	26.25	-86.4	-2.4	-23.2	-3.9	-4.6	609.08	+632.4	+2.5	+30.2	+4.1	+4.9	6	-4	+12	+6	+3	+10
	rand-read	1th-4k	10.29	-27.4	-41.7	-57.2	-18.2	-57.0	97.16	+37.7	+71.6	+133.8	+22.2	+132.5	0	0	+1	+1	0	0
		1th-128k	3.21	-31.3	-38.1	-41.4	-19.6	-48.0	311.49	+45.5	+61.6	+70.7	+24.4	+92.3	1	0	+1	0	0	0
		16th-4k	143.39	-93.1	-43.7	-38.8	-30.1	-41.3	111.47	+1345.2	+77.7	+63.3	+43.1	+70.8	4	-3	+10	+5	+6	+9
		16th-128k	24.53	-85.2	-13.4	-39.7	-4.6	-17.6	651.83	+575.7	+15.5	+65.9	+4.8	+21.3	6	-4	+9	+3	+3	+7
	seq-write	1th-4k	237.22	-7.9	-8.5	+4.3	+1.3	-64.1	4.23	+8.7	+9.2	-4.4	-1.4	+186.3	3	+1	+2	+1	+1	+1
		1th-128k	11.24	-26.3	-37.8	-13.7	-14.5	-73.7	88.98	+36.1	+61.7	+16.4	+17.3	+287.8	3	+1	+2	+1	+1	+1
		16th-4k	421.46	-32.4	-36.6	+1.4	+0.5	—	37.92	+48.0	+58.5	-1.4	-0.5	—	7	-1	0	+2	+1	—
		16th-128k	13.17	-32.2	-32.8	+1.5	+0.4	—	1213.71	+47.5	+48.7	-1.5	-0.4	—	6	-3	-1	+2	+1	—
	rand-write	1th-4k	239.29	-80.8	-63.8	-23.7	-34.4	-63.5	4.18	+419.2	+177.7	+31.0	+52.5	+179.8	3	-1	+4	+6	+6	+13
		1th-128k	13.22	-21.6	-38.3	-7.4	-15.9	-76.3	75.64	+27.4	+62.7	+8.1	+18.9	+351.2	2	+1	+2	+2	+3	+5
		16th-4k	361.46	-87.3	-75.8	-49.3	-39.2	—	44.25	+684.6	+315.5	+97.7	+64.6	—	8	-6	-1	+2	+4	—
		16th-128k	13.46	-26.4	-18.2	+0.4	+0.1	—	1187.63	+36.4	+22.3	-0.4	-0.1	—	4	-1	+2	+3	+4	—
	create-1th		52.50	-4.0	-4.0	-3.4	-1.6	-90.3	19.05	+4.2	+4.2	+3.5	+1.6	+932.8	2	+1	+1	0	0	0
	create-16th		65.73	-0.4	-2.5	+1.6	-0.6	-90.3	235.63	+0.3	+2.6	-1.4	+0.6	+960.9	18	-1	0	-2	-1	-15
	read-1th		21.73	-47.8	-47.8	-55.7	-25.3	-75.0	46.02	+91.8	+91.9	+125.6	+33.9	+300.7	1	0	+1	0	0	+1
	read-16th		136.41	-85.1	-30.6	-21.9	-17.5	-79.9	79.55	+878.8	+72.1	+45.6	+31.1	+640.2	4	-3	+7	+4	+4	+3
	delete-1th		31.36	-14.5	-25.6	-3.2	-4.0	-84.8	31.90	+16.9	+34.5	+3.3	+4.3	+556.1	1	0	+1	+1	+1	+1
	delete-16th		31.44	-5.9	-26.1	-3.0	-6.6	-80.2	235.72	+43.7	+68.2	+20.2	+13.6	+994.4	4	+1	0	+2	+1	-1
	file-server		65.41	-80.5	-13.8	-32.2	-33.0	-40.7	616.52	+503.9	+10.8	+39.0	+42.1	+74.5	10	-7	+7	+2	0	+17
	mail-server		49.08	-72.6	-41.4	-39.6	-30.3	-57.7	252.08	+301.2	+71.0	+66.6	+43.0	+151.4	5	-2	+3	+2	+2	+1
	web-server		208.66	-92.0	+12.2	-25.9	-17.0	—	231.64	+1961.5	+26.2	+98.4	+42.8	—	26	-23	+1	+1	-1	—

Table 2: File system workload results. Columns "Native" show absolute values; other columns show the relative difference as a percentage (for throughput and latency) or the absolute difference in percent points (for CPU utilization) against Native.

obtained under the analogous block device workloads. FUSE incurs overhead only under *rand-read-1th-Xk* and *rand-read-16th-4k*, lowering throughput by between 19.1% and 28.6% and being outperformed by BDUS.

On NVMe, results for read workloads are also akin to those obtained under the corresponding block device workloads. However, write workloads exhibited less variability than their block device counterparts, with confidence interval half widths being under 5% of the sample mean for throughput and latency. Further, the magnitude of the observed overheads is lower than under the analogous block device workloads, although results follow a similar pattern.

Under workloads *seq-write-16th-Xk* and *rand-write-16th-Xk* on NVMe, FUSE incurred overheads of at least 99.8% on throughput, and we omit such results from the table. Although we did not observe any correctness issues in FUSE's operation, the fact that this only occurred with the dual-socket system hosting the NVMe device leads us to hypothesize that it may be caused by an inadequacy of FUSE's implementation to the NUMA architecture or high number of cores. Under the remaining workloads, BDUS either matches or outperforms FUSE in achieved throughput, latency, and CPU utilization on both SATA and NVMe devices.

Metadata-intensive micro workloads. Regarding workloads *create-Xth*, we first note that the slight improvement in performance exhibited by BUSE on SATA devices appears to be due solely to result variability. Besides this, only FUSE incurs overhead under these workloads, up to 90.3% on throughput and 960.9% on latency on NVMe. Under *read-Xth*, BDUS outperforms all other systems on SATA and NVMe, degrading throughput and latency respectively by at most 16.8% and 20.1% on SATA, and by at most 25.3% and 33.9% on NVMe. Lastly, under *delete-Xth*, nbdkit, TCMU, and BDUS degrade latency on SATA by up to 7.5%. On NVMe, TCMU and BDUS perform the best, increasing latency by at most 20.2%. FUSE is the least performant system on SATA and NVMe, degrading throughput and latency by up to 84.8% and 994.4%, and CPU utilization on SATA by up to 18 pp.

FUSE's significant overhead under metadata-intensive workloads stems from its need to forward many requests relating to file metadata to the user-level driver. Since many of these requests can be served by in-memory operations or amount to manipulating cached data, the remaining systems perform much less user-kernel communication.

Macro workloads. Under *file-server* on SATA devices, FUSE outperforms all other pass-through systems, the former degrading throughput (latency) by 5.5% (13.4%) and the others by at least 20.5% (40.0%). However, FUSE increases CPU utilization by 19 pp, and the remaining systems by at most 5 pp. Further, nbdkit is the best-performing system on NVMe, lowering throughput by 13.8% and increasing latency by 10.8%.

On SATA under *mail-server*, FUSE performs the worst, degrading throughput by 6.6% and latency by 7.0%. On NVMe, all systems impose higher overhead, with BDUS performing the best and degrading those metrics by 30.3% and 43.0%.

Under *web-server* on SATA, BDUS outperforms all other systems, degrading throughput (latency) by 4.1% (9.8%) and increasing CPU utilization by 6 pp. In particular, FUSE imposes overheads of 45.3%, 95.1%, and 34 pp on those metrics. On NVMe, we omit results for FUSE as it lowered throughput by 99.8%, likely due to the aforementioned implementation issues. Here, nbdkit performs the best and appears to batch requests more eagerly than Native, consequently improving throughput by 12.2% but also degrading latency by 26.2%.

Note that on NVMe, nbdkit outperforms TCMU and BDUS under *file-server* and *web-server*, which employ 50 and 100 threads, and the opposite occurs under *mail-server*, which uses 16 threads. This is because nbdkit is configured with 16 threads per each of its 16 connections to the NBD client, for a total of 256 threads, while TCMU and BDUS employ a total of 16 threads. Although this gives nbdkit an unfair advantage, this configuration was used because nbdkit is unable to share worker threads among connections, and employing a single thread per connection would degrade its performance.

6.3 Discussion

We now briefly discuss the main findings of the evaluation. First, the fact that BUSE is only able to process requests sequentially hinders its performance not only under multi-threaded workloads, but also in scenarios where optimizations applied by the operating system introduce parallelism (*e.g.*, read ahead). nbdkit overcomes this restriction but is still shown to significantly degrade performance under many workloads, while noticeably increasing CPU utilization.

A primary factor contributing to the limited performance and higher resource utilization of NBD-based solutions is their reliance on sockets for communication between the kernel client and the user-space server, which necessarily cause extraneous memory copies when transferring requests and replies. Although TCMU sidesteps this limitation and generally outperforms NBD-based solutions under write and metadata-intensive workloads, its overhead is still significant and it imposes similar CPU utilization increments.

Following a clean-slate approach and striving to reduce memory copies and system call invocations, the BDUS framework is able to outperform these systems. Concretely, out of all systems and under all 41 workloads contemplated in the evaluation, BDUS is only noticeably surpassed in throughput or latency by nbdkit under *file-server* and *web-server* on NVMe devices (due to the higher-parallelism configuration for nbdkit that was used in the evaluation), and frequently incurs less CPU utilization than other solutions.

Additionally, BDUS outperforms the FUSE framework under many workloads, particularly metadata-intensive ones. This is because FUSE requires intervention of the user-level driver to satisfy many file metadata requests, which often do not translate into operations on the underlying block device and can thus avoid communication with BDUS' user-level driver [36]. In fact, BDUS never exhibits higher CPU utilization than FUSE, and is surpassed by it in performance only under the *file-server* workload on SATA devices.

For this reason, BDUS should also be considered when developing storage solutions that can be constructed at both the block and file system layers (*e.g.*, compression, deduplication, thin provisioning, encryption, replication [1, 12, 33]). This is in addition to general advantages of development at the block layer such as its simpler interface when compared to that of POSIX file systems, facilitating development, and the wider applicability of solutions exposing that interface.

7 RELATED WORK

Contrarily to BUSE [6], nbdcpp [5], nbdkit [14], and tcmu-runner [16], BDUS does not rely on NBD or Linux's SCSI target and exhibits better performance and lower resource consumption. ABUSE [4] is a bare-bones Linux framework that enables the development of block device drivers in user space, and like BDUS it uses `ioctl()` calls for communication between a kernel module and a user-level driver. However, it only allows drivers to handle READ and WRITE requests, in particular ignoring FLUSH and DISCARD requests indispensable for correct operation with many storage devices. As this precludes the implementation of a correct pass-through driver, and would provide the framework with an unfair performance advantage, it was not included in our evaluation.

Previous works have investigated the development of user-space device drivers in general, without a focus on storage drivers, and evaluations failed to compare proposed solutions with previous ones [3, 21–23, 27, 37]. We did not include any of these systems in our evaluation as [21, 22, 27] are not publicly available, [23, 37] are implemented for non-Unix-based microkernel operating systems, and [3] only allows implementing character device drivers. Conversely, BDUS focuses on providing a user-level development interface for block device drivers, its implementation is open source, and its performance is compared against existing solutions.

Other works provide mechanisms for partially migrating kernel-level device drivers to user space while keeping performance-critical code in the kernel, in an attempt to reap the advantages of user-level drivers while avoiding most performance reductions and porting costs [24, 25, 30]. Since request processing logic is a highly performance-sensitive part of any block device driver, it would be inadequate to implement it in user space using these systems, and thus they were not contemplated in our evaluation. In contrast, BDUS achieves the performance results presented above while transitioning all driver logic to user space.

Like BDUS, platforms that enable the development of file systems in user space have the goal of transitioning part of the storage stack to the user level. The FUSE framework is currently the most prominent example of such a platform, being used both for experimentation and for developing full-fledged production file systems, and has been extensively evaluated [29, 32, 36]. BDUS is complementary to said platforms and features a distinct implementation, as its kernel component interacts with the block layer instead of with the virtual file system layer.

Finally, systems such as SPDK [40] enable direct access from user space to certain types of storage devices, bypassing the operating system's storage stack and thus attaining higher performance at the cost of requiring changes to applications. Orthogonally, SPDK provides facilities to implement custom NBD servers and iSCSI [19] or NVMe-oF [38] targets. These could be used to build user-level block device drivers, but such approaches would rely on storage stacks designed for networked access and thus suffer from the same performance limitations as the solutions evaluated in this paper. BDUS is instead built specifically to enable the development of user-space drivers, achieving higher performance and unlocking further improvements and optimizations.

8 CONCLUSION

We describe and evaluate BUSE [6], nbdcpp [5], nbdkit [14], and tcmu-runner [16], which enable the user-level development of block device drivers by leveraging the NBD protocol [2] and Linux's SCSI target [15], and find that they incur significant overhead on throughput, latency, and CPU utilization under several workloads.

Consequently, we present BDUS, a framework built from the ground up for developing such drivers in user space, and show that its open-source Linux implementation outperforms existing solutions while consuming less CPU resources. By showing that BDUS outperforms FUSE [13] in file system stacks, we also motivate the use of the former for the development of storage functionalities that may be inserted at both the block and file system layers.

BDUS may nevertheless be further optimized. For instance, employing multiple dispatch queues may improve the performance of highly-parallel workloads [18]. Further, io_uring and its polling features may help mitigate context switching delays [7, 39]. Support for splicing [35], which transfers data between file descriptors without copying it to user space, can also be advantageous for drivers that can avoid inspecting the payload of certain requests (*e.g.*, deduplication). We leave such improvements as future development work.

REFERENCES

[1] [n.d.]. Device-mapper Resource Page. Retrieved May 3, 2021 from https://www.sourceware.org/dm

[2] [n.d.]. Network Block Device. Retrieved May 3, 2021 from https://nbd.sourceforge.io

[3] 2003. FUSD - a Linux Framework for User-Space Devices. Retrieved May 3, 2021 from http://www.circlemud.org/jelson/software/fusd

[4] 2015. naota/abuse-kmod: ABUSE: user space block device driver. Retrieved May 3, 2021 from https://github.com/naota/abuse-kmod

[5] 2017. dsroche/nbdcpp: Network Block Device drivers in userspace C++. Retrieved May 3, 2021 from https://github.com/dsroche/nbdcpp

[6] 2018. acozzette/BUSE: A block device in user space for Linux. Retrieved May 3, 2021 from https://github.com/acozzette/BUSE

[7] 2019. Efficient IO with io_uring. Retrieved May 3, 2021 from https://kernel.dk/io_uring.pdf

[8] 2020. filebench/filebench: File system and storage benchmark that uses a custom language to generate a large variety of workloads. Retrieved May 3, 2021 from https://github.com/filebench/filebench

[9] 2020. TCM Userspace Design. Retrieved May 3, 2021 from https://www.kernel.org/doc/html/latest/target/tcmu-design.html

[10] 2020. The Userspace I/O HOWTO. Retrieved May 3, 2021 from https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html

[11] 2021. axboe/fio: Flexible I/O Tester. Retrieved May 3, 2021 from https://github.com/axboe/fio

[12] 2021. DRBD. Retrieved May 3, 2021 from https://www.linbit.com/drbd

[13] 2021. libfuse/libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. Retrieved May 3, 2021 from https://github.com/libfuse/libfuse

[14] 2021. libguestfs/nbdkit: NBD server toolkit with stable ABI and permissive license. Retrieved May 3, 2021 from https://github.com/libguestfs/nbdkit

[15] 2021. Linux SCSI Target. Retrieved May 3, 2021 from http://linux-iscsi.org

[16] 2021. open-iscsi/tcmu-runner: A daemon that handles the userspace side of the LIO TCM-User backstore. Retrieved May 3, 2021 from https://github.com/open-iscsi/tcmu-runner

[17] 2021. SYSSTAT. Retrieved May 3, 2021 from http://sebastien.godard.pagesperso-orange.fr

[18] 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*. https://doi.org/10.1145/2485732.2485740

[19] M. Chadalapaka, J. Satran, K. Meth, and D. Black. 2014. *Internet Small Computer System Interface (iSCSI) Protocol (Consolidated)*. RFC 7143. Retrieved May 3, 2021 from https://tools.ietf.org/html/rfc7143

[20] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. *ACM SIGPLAN Notices* 51, 6 (June 2016), 431–447. https://doi.org/10.1145/2980983.2908101

[21] Peter Chubb. 2004. Get More Device Drivers out of the Kernel!. In *Proceedings of the Linux Symposium*, Vol. 1. 149–161.

[22] Peter Chubb. 2004. Linux Kernel Infrastructure for User-Level Device Drivers. In *Proceedings of the Linux.Conf.Au*.

[23] Kevin Elphinstone and Stefan Götz. 2004. Initial Evaluation of a User-Level Device Driver Framework. In *Proceedings of the 9th Asia-Pacific Conference on Advances in Computer Systems Architecture*. 256–269. https://doi.org/10.1007/978-3-540-30102-8_21

[24] Vinod Ganapathy, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. 2007. Microdrivers: A New Architecture for Device Drivers. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. Article 15, 6 pages.

[25] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. 2008. The Design and Implementation of Microdrivers. *ACM SIGOPS Operating Systems Review* 42, 2 (March 2008), 168–178. https://doi.org/10.1145/1353535.1346303

[26] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proceedings of the 21st USENIX Security Symposium*. 459–474.

[27] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yue-Ting Shen, Kevin Elphinstone, and Gernot Heiser. 2005. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology* 20, 5 (Sept. 2005), 654–664. https://doi.org/10.1007/s11390-005-0654-4

[28] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, Vol. 2. 21–33.

[29] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and Extension of User Space File Systems. In *Proceedings of the 25th ACM Symposium on Applied Computing*. 206–213. https://doi.org/10.1145/1774088.1774130

[30] Matthew J Renzelmann and Michael M Swift. 2009. Decaf: Moving Device Drivers to a Modern Language. In *Proceedings of the USENIX Annual Technical Conference*.

[31] Michael F. Spear, Tom Roeder, Orion Hodson, Galen C. Hunt, and Steven Levi. 2006. Solving the Starting Problem: Device Drivers As Self-describing Artifacts. *ACM SIGOPS Operating Systems Review* 40, 4 (April 2006), 45–57. https://doi.org/10.1145/1218063.1217941

[32] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. 2015. Terra Incognita: On the Practicality of User-Space File Systems. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems*.

[33] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. 2014. Dmdedup: Device Mapper Target for Data Deduplication. In *Proceedings of the 2014 Ottawa Linux Symposium*.

[34] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *;login: The USENIX Magazine* 41, 1 (March 2016), 6–12.

[35] Linus Torvalds. 2006. Re: Linux 2.6.17-rc2. Mailing list. Retrieved May 3, 2021 from https://lkml.org/lkml/2006/4/19/237

[36] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. 2019. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Transactions on Storage* 15, 2, Article 15 (May 2019), 49 pages. https://doi.org/10.1145/3310148

[37] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. 2008. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*. 241–254.

[38] NVM Express Workgroup. 2019. *NVM Express™ over Fabrics Revision 1.1*. Specification. Retrieved May 3, 2021 from https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf

[39] Jisoo Yang, Dave B Minturn, and Frank Hady. 2012. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*.

[40] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science*. 154–161. https://doi.org/10.1109/CloudCom.2017.14