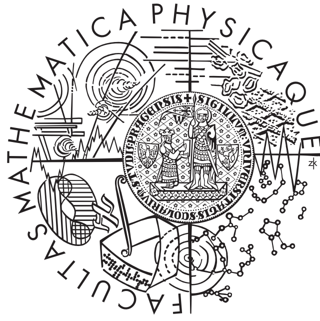

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Vojtěch Aschenbrenner

BUSE: Block Device in Userspace

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Prof. Peter Desnoyers, Ph.D.
Study programme: Computer Science
Study branch: Software Systems

Prague 2021

Declaration

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, July 22, 2021

Vojtěch Aschenbrenner

Acknowledgement

I would like to thank my supervisor, professor Peter Desnoyers from Northeastern University for introducing me to computer storage research, his genuine interest in the topic and guidance during the thesis creation process.

I am also grateful to professor Petr Tůma from Charles University for the role of a local thesis consultant and for making the cooperation with professor Peter Desnoyers formally possible. His different perspective on the topic was also very helpful.

Annotation

Title	BUSE: Block Device in Userspace
Author	Vojtěch Aschenbrenner v@asch.cz
Department	Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University, Prague, Czech Republic
Supervisor	Prof. Peter Desnoyers, Ph.D. p.desnoyers@northeastern.edu Northeastern University, Boston, MA, USA
Consultant	Prof. Ing. Petr Tůma, Dr. petr.tuma@d3s.mff.cuni.cz Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University, Prague, Czech Republic

Abstract

Implementation of block device drivers in userspace of modern general-purpose operating systems, although possible, is fairly uncommon, poorly supported and usually achieves only low performance. Being able to implement high-performance drivers in userspace with ease would allow for faster iterations in storage research and would make it possible to design block devices which operate in radically different ways.

In this thesis, we present Block Device in Userspace (BUSE), a Linux kernel module and communication protocol which makes it easy to develop userspace block-device drivers. Compared to the existing approaches, BUSE can scale on modern multicore architectures and provides at least 7x higher throughput with significantly simpler setup. Furthermore, the kernel module communicates with the userspace driver through shared memory, eliminating an extraneous memory copy. BUSE also solves the write-after-write and read-after-write consistency issues which stem from the use of multiple hardware queues in the Linux storage stack, allowing the implementation to focus on the domain of the problem.

As a proof-of-concept, we implemented Block Device in S3 (BS3), a

userspace block device implementation backed by Amazon S3 (or any other S3-compatible storage) on top of BUSE. BS3 can be used as a generic disk providing a throughput of more than 10 GB/s, making it faster than the fastest possible locally-attached PCIe 3.0 4x NVMe SSDs. It is up to 130x faster than CloudBD, a commercial product advertising high performance. S3's durability of 99.999999999 % (eleven nines) translates into the durability of BS3 block devices. Furthermore, BS3 is prefix-consistent under all failure conditions, preserving file system crash consistency guarantees.

Keywords

Block, Device, BUSE, BS3, S3, Linux, Kernel, Storage, Userspace, Driver

Contents

1 Introduction 4

1.1 Contribution 5

1.1.1 Block Device in Userspace (BUSE) 5

1.1.2 Block Device in S3 (BS3) 5

2 BUSE | Background Knowledge 7

2.1 Block Devices 7

2.1.1 Ordering and Durability 7

2.1.2 Other Operations 8

2.2 Page Cache 8

2.3 Multi-Queue Block I/O Queueing Mechanism 9

2.3.1 History and Motivation 9

2.3.2 New Solution: blk-mq 11

3 BUSE | Problem Statement & Related Work 12

3.1 Advantages of BUSE Drivers 12

3.2 Related Work 14

3.2.1 Network Block Device 14

3.2.2 Target Core Module in Userspace 15

3.2.3 Storage Performance Development Kit 15

3.2.4 Filesystem in Userspace 15

3.2.5 BDUS 16

4 BUSE | Architecture & Implementation 17

4.1 Read Path 17

4.2 Write Path 19

4.2.1 Flush Handling 20

4.3 Hazards 21

4.3.1 Read-after-Write Hazard 21

4.3.2 Write-after-Write Hazard 21

4.4 Device Management and Configuration 22

4.4.1 Declaration 22

4.4.2 Configuration 23

4.4.3 Power On 25

4.4.4 Userspace Disconnect / Crash 25

4.4.5 Power Off / Termination 25

4.4.6 Destruction 25
4.5 Usage Example 26
4.5.1 Read Path 26
4.5.2 Write Path 27
4.6 BUSE Userspace Daemon Library 27
4.6.1 Go Interface 28
4.6.2 Options 28
4.6.3 Usage 29

5 BUSE | Evaluation **30**
5.1 Performance Evaluation in Cloud 30
5.1.1 Benefits 30
5.1.2 Drawbacks Mitigation 31
5.2 Experimental Setup 31
5.2.1 Software Configuration 31
5.2.2 Hardware Configuration 31
5.2.3 Null User-space Implementation 32
5.3 Microbenchmarks 34
5.3.1 I/O Parameter Sensitivity 34
5.3.2 Scaling with number of vCPUs 35

6 BS3 | Background Knowledge **44**
6.1 Object Storage 44
6.2 Log-structured Storage 44

7 BS3 | Problem Statement & Related Work **46**
7.1 Network Attached Disk 46
7.2 Related Work 47
7.2.1 Proprietary 47
7.2.2 Academia 48

8 BS3 | Architecture & Implementation **50**
8.1 Overview 50
8.1.1 Write Path 50
8.1.2 Read Path and LBA Mapping 51
8.2 Implementation Details 51
8.2.1 Write Path 51
8.2.2 Map Implementation 53
8.2.3 Object Operations Implementation 55
8.2.4 Garbage Collection 55
8.2.5 Map Recovery 56
8.2.6 Prefix Consistency 57
8.3 Null Mode 57

9 BS3 | Evaluation **58**
9.1 Experimental Setup 58
9.2 Microbenchmarks 58

9.2.1 I/O Parameters Sensitivity 58
9.2.2 Scaling with number of I/O Jobs 62
9.3 Macrobenchmarks 65
9.3.1 Fileserver 65
9.3.2 Varmail 65
9.3.3 OLTP 66

10 Conclusion **70**
10.1 Future Work 71

Bibliography **73**

A Thesis Meta-repository **79**

B BS3: Block Device in S3 **80**
B.1 Requirements 80
B.2 Installation 80
B.3 Usage 80
B.4 Configuration File 80

C BUSE: Block Device in Userspace **84**
C.1 Requirements 84
C.2 Installation 84

D Thesis Experiments **85**
D.1 Repository Structure 85
D.2 Requirements 85
D.3 FIO Experiments Usage 85

E Thesis Text **87**
E.1 Requirements 87
E.2 Usage 87

Chapter 1

Introduction

The ultimate storage system is always accessible, highly reliable, has unlimited performance, unlimited capacity, is easy to maintain and is compatible with all existing software. These goals are clearly out of reach of any locally-attached storage, as local resources are always finite.

Remote storage however has the potential to achieve on most of these goals due to the disaggregation of the client, the communication fabric and the physical storage backend. The storage backend can be designed to scale in size and performance, be extremely reliable and provide an API which clearly separates the client from the actual storage technology.

Recently, object stores are becoming widespread and are relied upon heavily in the cloud environment. They have very appealing properties, making them great backends for remote storage and retrieval of data. Most cloud providers in the market today provide an object store which is virtually unlimited in capacity, has practically unlimited performance only limited by the transport fabric, and often unbeatable durability. For example Amazon S3 has a stated durability of 99.999999999 % (eleven nines) and allows clients to upload and download arbitrary unstructured data in the form of objects. The API of S3—and for that matter, any object store—is incompatible with the traditional Unix block device APIs, requiring applications to be custom-tailored to be able to use them.

Missing pieces. There are two pieces missing which inhibit large-scale deployment of block devices using object stores as their backends:

1. Storage drivers with reasonable performance are implemented in the kernel. This increases development complexity due to the unavailability of userspace libraries which wrap the necessary APIs and other constraints of the kernel environment. Furthermore, maintenance of kernel drivers which are not part of the mainline release is a daunting task due to the ever-changing storage stack APIs. This effectively prevents the development of novel and/or experimental block devices.

2. The interface of object storage is vastly different from the interface of the regular block storage. Therefore, a "translation layer" is needed for translating I/O requests of one type into I/O requests of the other type. Such translation layer should provide sufficient performance to be a viable replacement of a traditional block device.

1.1 Contribution

This thesis aims to provide both missing pieces. In the first part of the thesis, we research how to implement an efficient block device in userspace (BUSE) and in the second part, we use BUSE to create a high-performance block device backed by Amazon S3 (BS3).

1.1.1 Block Device in Userspace (BUSE)

In the first part of the thesis, we analyze existing approaches to the development of userspace drivers. We study how they work within the Linux kernel storage stack. We especially take interest in their performance characteristics. The benchmarks show that all the solutions are only able to utilize a single hardware queue, which extremely limits their performance on modern multicore machines. We then design, implement and benchmark a novel solution which has following key attributes:

- Per-core hardware queues making the solution scalable with number of cores. We demonstrate write throughput of up to 115 GB/s and read throughput up to 72 GB/s on a machine with 48 vCPUs. Being able to provide extremely high throughput means that for realistic workloads with much lower demands, the CPU utilization is very low. For that reason, per-core scalability is fundamental for modern many-core low-power architectures.
- Batching of I/O write requests, improving single queue performance.
- Optional weak flush semantics, greatly lowering latency when durability semantics of flushes is not needed.
- In-kernel conflict resolution, making the userspace implementation much simpler.
- Performance up to 8x better than existing solutions.

1.1.2 Block Device in S3 (BS3)

The second part of the thesis is dedicated to BS3, a virtual disk for general data storage using Amazon S3 as a backend. In this part, we analyze related work from both industry and academia. Based on the analysis, we design, implement and evaluate BS3. The final solution has following key properties:

- No extra memory copies leading to very high performance.

- Scalability with number of cores.
- Throughput of more than 10 GB/s. This typically means that the network connection is the bottleneck, even with a dedicated 100 Gb/s link. This throughput is 2.5x higher than maximum throughput for PCIe 3.0 4x NVMe SSDs, which stands at 4 GB/s.
- Highly durable due to Amazon S3 being used.
- Performance up to 130x higher than a similar commercial solution (CloudBD).
- Consistent under all crash conditions. The log-structured style of writes combined with the atomicity of Amazon S3 object upload means that the device is in a consistent state under all crash scenarios, preserving file system crash consistency guarantees.

Chapter 2

BUSE | Background Knowledge

In this chapter, we introduce basic concepts of the Linux storage stack.

2.1 Block Devices

Unix-derived operating systems present devices designated to store data (HDDs, SDDs, USB drives, network-attached storage. . .) in the form of so-called block devices.

A *block device* is an abstraction allowing applications to work with fundamentally different devices in a unified way, usually through the block special file in `/dev` directory (Rubini and Corbet [1]). The well-known Linux syscalls `read(2)` and `write(2)` are used to read data from and store data to the block device respectively. The granularity of these operations is one block.

A *block* is a contiguous fixed-size region of data. The size of blocks, or *block size*, is advertised by the block device, and was historically synonymous with 512 B. Lately, 4kB-sized blocks are becoming the norm. The only requirement enforced by the Linux block layer is that the block size be a multiple of 512, and more exotic block sizes are possible, although seldom used in practice.

An *extent* is a contiguous sequence of blocks.

2.1.1 Ordering and Durability

Block devices provide fairly weak durability and ordering semantics. Still, providing these guarantees without sacrificing performance is a non-trivial task, as we shall see in later chapters.

Except in cases stated in the rest of this section, operations on a block device are generally unordered, i.e. their effects may be observed in any order.

Write ordering. Two writes are ordered if and only if one is completed before the other one is issued.

Flush as a barrier. Writes completed before a flush command happen before all writes issued after the flush command. There is no ordering within either group other than the one specified in previous paragraph.

A flush request is generated with the `sync(2)`, `fsync(2)` and `fdatasync(2)` syscalls.

Flush as a durability command. Block devices usually contain volatile write caches to speed up I/O operations, and they acknowledge write requests as soon as data reach that volatile cache. A flush command is used to make sure that all preceding write requests are stored persistently.

Flushes were introduced in 2010 to replace earlier block barriers as announced in (LWN [2]).

2.1.2 Other Operations

Block device drivers may elect to support other, more specialized operations, of which the following are of importance to us:

- A "discard" operation marks blocks on the device as currently unused. This has significant performance implications for devices with large erase blocks, such as flash-based devices and shingled magnetic recording (SMR) drives.
- A "write zeroes" request is useful for filling large extents of blocks with zeroes efficiently. Note that this does not mean that the device must actually be filled with zeroes; a sufficiently smart implementation may simply record that certain segments of the device should return all zeroes when read (unless overwritten in the meantime).

2.2 Page Cache

An important component of the block layer in Linux is the *page cache*. The page cache serves as a layer which caches data.

All read requests are first checked against the page cache and in the case of a cache hit, the (physical) block device need not be contacted at all. For high-latency devices such as spindle drives, the page cache significantly lowers the perceived latency of the device.

In the case of writes, the page cache serves as a write-back cache, delaying the actual write transaction to a more suitable time, e.g. when more data is accumulated. That in turns makes it possible to join adjacent writes, which often results in less overhead overall. For example, it may be possible to minimize head movement or achieve better compression.

The page cache occupies unused pages of main memory, and so is typically volatile. Under power failure, all writes residing only in the page cache will

be lost. As with the volatile device caches, a `sync(2)` or `fsync(2)` call makes sure that write requests are sent to the block device and persisted.

The page cache has one more important feature: when enabled, the block layer may perform a *read ahead* of data into the cache, lowering the latency of consecutive sequential reads. The triggering of this logic is subject to various heuristics. Read-ahead is especially useful for applications which often issue many small consecutive reads.

Interestingly, in our experience, for high performance block devices such as fast NVMe drives or a BUSE block device with fast userspace driver, page cache introduces a significant performance bottleneck. In these cases, the page cache can be bypassed by specifying the `O_DIRECT` flag when opening the block special file. Please refer to `open(2)`.

2.3 Multi-Queue Block I/O Queueing Mechanism

This section describes Linux Multi-Queue Block I/O Queueing Mechanism known as *blk-mq* (Linux, LWN, Bjørling et al. [3, 4, 5]). It is a Linux Block Layer framework introduced in Kernel 3.13 and completed in Kernel 3.16 (August 2014) aiming at multi-core scalability and capable of over 15 million IOPS with high-performance flash devices on multi-socket machines. BUSE is implemented as a blk-mq driver.

2.3.1 History and Motivation

Historically, the Linux kernel block layer provided two ways how a block driver could be attached to it.

One of them was so called "request-based" interface (*blk-sq* in this text). This mode of operation provides a single request queue for the entire system, where I/O requests are produced by the applications on one end and consumed and handled by the driver on another end. Requests in the queue can be reordered, merged or limited by bandwidth of fairness policies. Design of the Single Queue framework can be seen on Figure 2.1.

The rapid adoption of flash-based storage technologies capable of hundreds of thousand of I/O operations per second (IOPS) in recent decade has rendered the original single-queue model obsolete. The request queue in Figure 2.1 is protected with a single lock; with a thousand-fold increase in IOPS, the lock became a source of significant contention, making it impossible to leverage the performance of modern hardware.

The other way of writing a block device driver was to completely skip the single queue and hook in the bio layer like e.g. MD RAID does. This option is highly impractical as the device driver has to reimplement all the functionality provided by blk-sq.

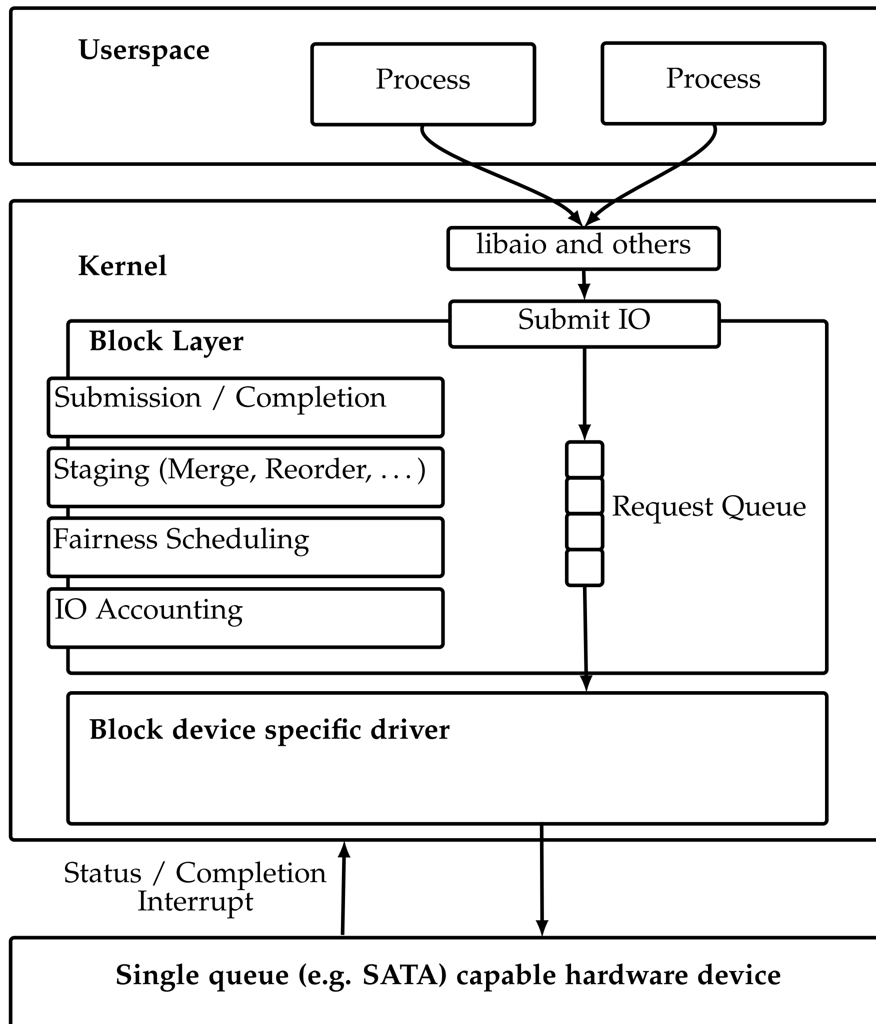

Figure 2.1: Single queue Linux block layer design. Figure is based on (Bjørling et al. [5]).

2.3.2 New Solution: blk-mq

Figure 2.2 shows the design of blk-mq. The main architectural change is clearly visible: the single queue is replaced by two separate sets of queues.

There are *software queues* to which each CPU core submits I/O requests, much like with blk-sq. Since there is one software queue per CPU core, lock contention is greatly reduced. These queues support many of the operations originally provided by blk-sq, such as I/O scheduling and fairness policies.

The other queues are *hardware dispatch queues*. The number of hardware dispatch queues depends on the device and the driver; there can be as little as a single hardware dispatch queue and as many as there are CPU cores in the system. These queues are used to keep track of I/O requests dispatched to the device until their completion.

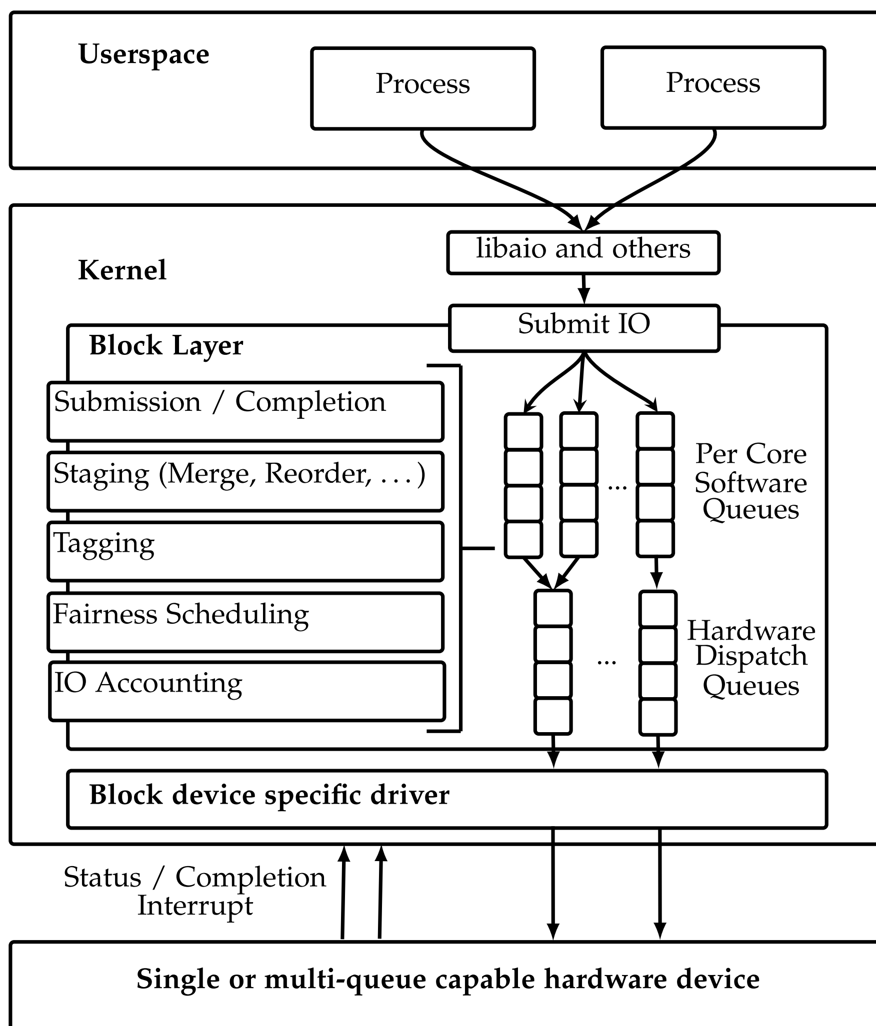

Figure 2.2: Multi-queue Linux block layer design. Figure is based on (Bjørling et al. [5]).

Chapter 3

BUSE | Problem Statement & Related Work

Traditionally, block device drivers are implemented in the Linux kernel, using the API of the block layer directly. This approach allows for highly efficient drivers to be written, as no extraneous memory copies or context switches are required. However, this performance comes at the expense of development complexity due to the restriction of the kernel environment.

Under certain conditions, it may be useful to implement the driver in userspace rather than in the kernel. BUSE aims to provide a way for userspace programs to act as de-facto *userspace block device drivers*. It achieves that by exposing a minimal interface from the kernel to userspace which makes it possible to handle requests sent to the hardware queues of blk-mq in an efficient manner.

3.1 Advantages of BUSE Drivers

In this section, we outline some of the perceived advantages of (BUSE-based) userspace drivers, hinting upon possible use-cases.

Block device drivers prototyping. The complexity of kernel development can be a limiting factor for research in the area of block devices, where the ability to test new ideas quickly and with relative ease is of high value. Prototyping can be faster and easier in userspace since higher-level programming languages cay be used (Python, Go...). Existing libraries can be leveraged to create non-traditional block devices, such as block devices backed by 3rd-party network services etc. Existing infrastructure for testing and debugging of software can be used without modification.

Easier development. As we will see later in Section 4.3, the highly asynchronous environment in which block drivers operate invites many hazards.

Some of these hazards are related to the implementation of the driver itself;

e.g. it is necessary to protect any shared data structures from race conditions with proper synchronization. That is always the case, irrespective of the operation of the block device itself—be it a physical block device, or a "virtual" network attached storage. This point is not about these issues.

Other hazards (which are of interest here) stem from the design of the block layer and the durability and ordering guarantees discussed in Section 2.1.1: certain operations must be carried out in particular order. How much this is a driver's concern depends on the block device in question. Hardware devices often address these issues with dedicated circuitry, while "virtual" block devices have to emulate that circuitry in software.

We believe it is better to provide these ordering guarantees *to* the block devices, rather than expecting compliance *from* the block devices. Operation ordering can be addressed by the common kernel part of the driver, the userspace component can be blissfully ignorant of any ordering requirements whatsoever.

This separation of concerns leads to much simpler userspace drivers which only focus on implementing their core logic.

Single common codebase. Having a standard way of exposing block devices to userspace also means that shared userspace libraries may be created which further simplify the task of writing block drivers. One such library is provided with BUSE to simplify the development of Go drivers specifically.

Common storage stack functionality. Nowadays, block device drivers can already be implemented completely in userspace (with no kernel counterpart) with e.g. SPDK. However, a feature-complete implementation of a block device driver based on SPDK would have to reimplement lots of functionality which is already provided by the kernel:

- Block layer software queue services: scheduling, I/O accounting, plugging etc.
- Page cache services: prefetching, caching.
- Stackability: a regular block device can be part of the stack together with dm-crypt, md-raid, LVM etc.

All of these functions are however available to the BUSE driver in the kernel.

Performance. Until now, driver authors had to choose between providing a kernel driver which had the potential to be efficient and performant, or a userspace driver. Depending on the technology used, one could either sacrifice performance (NBD, TCMU) or the common functionality of the storage stack and a unified block device interface (SPDK).

BUSE demonstrates that it is possible to retain most of the advantages of a purely in-kernel driver while also providing the convenience of userspace software development and retaining the well-known Unix interface.

Fault-tolerance. Development and testing of highly experimental code in the kernel is challenging as we experienced firsthand during the development of the BUSE kernel driver. Simply put, almost any bug in kernel code leads to either a kernel panic or irrecoverable corruption of the running system, requiring a reboot. This makes debugging expensive in terms of development time.

With userspace drivers, this is much less of a problem: if the userspace component crashes, it may simply reconnect to the kernel driver and continue processing requests. Granted, instability of the userspace driver is not something we're aiming for in any case, but being able to recover from it gracefully is very convenient.

3.2 Related Work

This chapter briefly describes the currently available solutions and their drawbacks.

3.2.1 Network Block Device

Network Block Device (NBD) is originally a network protocol which made it possible to use a remote machine's block device as if it was attached to the local system (NBD [6]). It is a fairly old protocol, developed around 1997 (Linux Journal [7]) and like BUSE, comprised of 2 parts:

Client. The client component of NBD is part of the Linux kernel and configures a local block device exposed as `/dev/nbdX`. Requests submitted to this device are sent through a socket to the server side implemented in userspace. The client can be configured with a userspace utility called `nbd-client`.

Server. Server implements userspace handlers for requests sent by the client. NBD can use Unix-domain sockets instead of network sockets to eliminate the overhead of connection management. Furthermore, a multi-connect connection can be used to increase performance due to introduced parallelism in the server part. High-performance server implementations with plugin support exist, such as (nbdkit [8]).

Limitations. The most limiting factors of NBD are 1) the fact that communication via sockets implies memory copy and 2) that the driver uses just one hardware dispatch queue. The choice to support a single hardware queue is enforced by the semantics of the protocol and cannot be easily fixed without changing the protocol in the first place.

NBD forms the foundation of a commercial product called CloudBD (CloudBD [9]). Both NBD by itself and CloudBD will be evaluated in Chapter 5 (for comparison with BUSE) and Chapter 9 (for comparison with BS3).

3.2.2 Target Core Module in Userspace

Target Core Module (TCM) (Datera [10]) is Linux kernel SCSI (Seagate [11]) target. SCSI target is an endpoint which handles SCSI commands, generated by the SCSI initiator. SCSI target can be viewed as analogous to NBD server and SCSI initiator to NBD client from the previous section. TCM, which is also called Linux IO (LIO), is again made up of two components.

Backstore. Backstore is actually a driver for TCM, providing concrete implementation of the SCSI target. E.g., it can provide already existing block device on the system as a target, or use a file in the local filesystem to serve as a backstore.

The important backstore for us is TCM in Userspace (TCMU) (Kernel [12]), which forwards SCSI commands to a userspace process. The communication between the kernel component and the userspace component is done via UIO (Kernel, LWN [13, 14]) and frameworks like tcmu-runner (Open-iSCSI [15]) and go-tcmu (CoreOS [16]) make it easy to implement the userspace logic.

Fabric. Fabric exposes backstores to initiators. Various protocols are available, most prominently iSCSI (Chadalapaka et al. [17]) and Fibre Channel (Weber et al. [18]). For us, *loopback* is the most relevant fabric, as it exposes the backstore as a block device in `/dev`.

3.2.3 Storage Performance Development Kit

Storage Performance Development Kit (SPDK) (Yang et al. [19]) allows accessing chosen storage devices directly from userspace, eliminating system calls and associated context switches, leading to higher performance. However, applications cannot use the traditional Unix file API and have to be modified to support the custom SPDK interface. This also means that kernel-operated file systems cannot be used. Furthermore, since the operating system block layer is not used, all functionality provided by the block layer has to be reimplemented and devices cannot be used as a part of the stackable storage layer, as mentioned before.

3.2.4 Filesystem in Userspace

Filesystem in Userspace (FUSE) (libfuse, Szeredi, Vangoor, Tarasov, and Zadok [20, 21, 22]) is the main inspiration for BUSE. FUSE allows implementation of a file system in userspace without writing any kernel code just by using *libfuse* and implementing appropriate callbacks. FUSE is successfully used for research purposes (Tarasov et al. [23]) as well as for production file systems (Boyer, Broomfield, and Perrotti, NTFS-3G, sshfs [24, 25, 26]) where performance is not critical.

3.2.5 BDUS

Developed independently, BDUS (Faria et al. [27]) was published only days before this thesis was submitted. The motivation is to provide a framework for creating a block device in userspace. Like BUSE, BDUS is implemented as *blk-mq* driver, however with a single hardware queue only—completely missing the apparent trend.

Unlike BUSE, BDUS uses a control file to communicate with the userspace driver and relies on IOCTLs to send commands. Like BUSE, it attempts to minimize the number of memory copies by copying payloads directly to/from the user buffer.

Chapter 4

BUSE | Architecture & Implementation

Figure 4.1 shows the design of the BUSE driver. The entry point is an I/O request coming from a hardware dispatch queue which is handled by BUSE write queue or read queue, depending on the I/O request type. When the I/O request is added to the appropriate queue, the userspace daemon reads the request, performs necessary operations to handle it and acknowledges it back to the in-kernel driver.

Communication between userspace and kernel is achieved with two character devices for every single BUSE queue, with separate read and write character device per queue. Reading from the character device reads the next I/O request and writing to the character device acknowledges an I/O request. For the I/O requests' data, a memory buffer is shared between kernel and userspace. Sharing is achieved when the userspace memory-maps the character device file into its memory.

In the following sections we describe request handling in finer detail for the read path and the write path separately. Furthermore, we mention some implementation details and describe the communication protocol. Finally, the BUSE device lifecycle, management and configuration are explained.

4.1 Read Path

Read queues handle all read requests; there are as many read queues as there are hardware queues. There is no resource sharing between the queues on the hot path and so no locking is usually necessary. All of these design choices follow from our previous research of blk-mq and the problems it solves.

When a read I/O request is received, following actions take place:

1. BUSE driver in the kernel reserves space in the shared memory for the data that will eventually be read by the userspace daemon. A bitmap is used to track the allocated space.

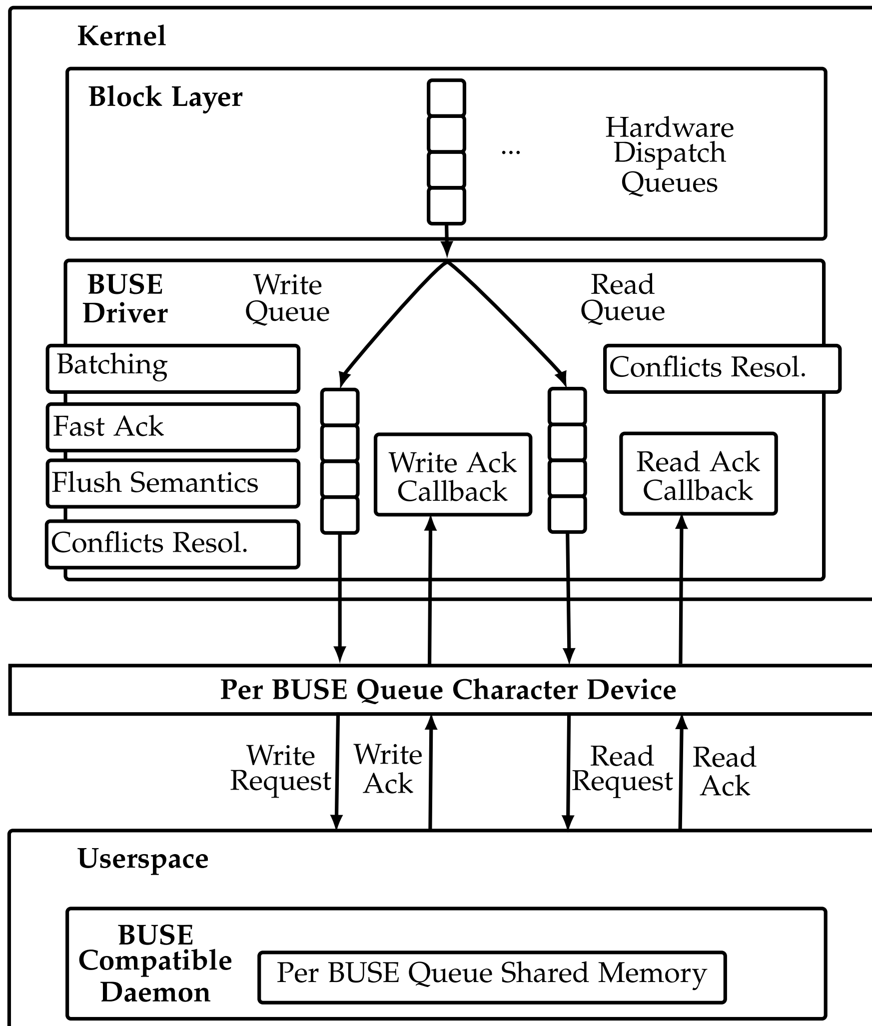

Figure 4.1: BUSE design.

2. A read chunk is created. Read chunk describes request details such as sector from which data should be read, the number of sectors (blocks) to read and an offset to the shared memory where the data are expected to be found once processing is finished. The precise description of wire format may be found in Table 4.1.
3. The chunk is added to the queue of chunks yet to be received by userspace.
4. The userspace daemon obtains the read chunk via the character device, carries out a specific read operation and writes the read data back into the appropriate segment of the shared memory buffer as requested by kernel. Note that usually, the userspace driver will use the shared memory buffer directly. E.g., when reading from a network, the driver will receive data directly into the shared memory buffer where the kernel driver expects them, avoiding expensive memory copies.
5. Userspace acknowledges the request by writing the shared memory offset (which is unique to the current request at this time) to the character device. Acknowledgements of I/O requests are allowed to be done out-of-order, which allows asynchronous processing and avoids head-of-line blocking.
6. The kernel driver copies data from the shared memory to the buffer provided by *blk-mq* and marks the I/O complete. It also frees the space in the shared memory buffer for another request.

Table 4.1: Read chunk description. All values are interpreted as little endian unsigned integers.

Byte Position	Description
0–7	First sector to be read.
8–15	Number of sectors to be read.
16–23	Offset to read queue shared memory.

4.2 Write Path

Write queues are slightly more complicated than the read queues, as they leverage write semantics described in Section 2.1.1 to maximize write performance. The following steps are performed when a write request is received:

1. BUSE driver in the kernel creates a write chunk. A write chunk is merely a batch of write requests and is the smallest unit transfered to userspace. A write chunk contains both the data (the write payloads) as well as the metadata (information about sectors being written). The shared memory of the write queue is the memory backing all the write chunks. Wire formats of the write chunk and the individual write requests are described in Table 4.2 and Table 4.3 respectively. Please note that this step can be

skipped if the current write chunk has capacity to hold the incoming write request.

2. The driver appends the write request to the currently open write chunk and immediately completes the I/O request with respect to blk-mq. (I.e., from the perspective of the storage stack, the request is complete.) Should the total size of all writes exceed the target write chunk size, a new chunk is opened and the old one is closed.
3. When the write chunk is closed, its metadata are added to the list of chunks yet to be received by userspace. The write chunk metadata contain the number of writes batched in the chunk and an offset to the shared memory where the chunk is located.
4. Userspace reads the write chunk metadata, extracts all write requests from the chunk and performs appropriate write operations.
5. Userspace acknowledges the write chunk to the kernel by writing the shared memory offset (which is unique to the current request at this time) to the character device.
6. The kernel driver recycles the write chunk. The write chunk is now empty and can be used to hold another batch of writes. The driver does not perform any action in the blk-mq direction, as the write requests were marked complete in step 2 above.

Table 4.2: Write chunk description. All values are interpreted as little endian unsigned integers.

Byte Position	Description
0–7	Offset to write queue shared memory.
8–15	Number of write records in the write chunk.

Table 4.3: Write record description. All values are interpreted as little endian unsigned integers.

Byte Position	Description
0–7	First sector to be written.
8–15	Number of sectors to be written.
16–23	Sequential number of the write I/O request.
24–31	Reserved for future use.

4.2.1 Flush Handling

Since there is no guaranteed ordering on the dispatch queues and a writing application can migrate between different CPU cores, it is necessary to broadcast flush I/O request to all write queues. Every single write queue responds

to the flush request in the following way:

1. Closes the current write chunk (unless empty) and puts it to the list of chunks yet to be received by userspace.
2. Creates a special empty *flush chunk* which is added to the list right after the currently closed chunk. To distinguish flush chunks from the ordinary write chunks, the flush chunks have a shared memory offset greater than 2^{32}, a value which is not used for any valid shared memory offset.

The original flush request is marked complete as soon as all broadcast requests are acknowledged by the userspace driver. The handling of flushes in userspace is specific to the userspace driver to the extent allowed by the flush semantics, just as the handling of reads and writes is.

Weak Flush. This way of handling flushes preserves the barrier semantics, but leaves out the durability semantics. The userspace driver can immediately acknowledge the flush request without waiting for preceding write requests to become persistent. This can significantly improve performance in situations where durability is not needed or is achieved through other means.

4.3 Hazards

If implemented exactly as described above, both the read queues and the write queues would suffer from several problems. Write batching in write queues introduces the possibility of a *read-after-write hazard* while the per CPU queues introduce a *write-after-write hazard*. Both situations are however resolved correctly by the kernel driver and described in following sections.

4.3.1 Read-after-Write Hazard

If an application performs a "write block b", the following "read block b" operation can return old data. The reason is that the userspace is not aware of the "write block b" request due to the batching done by the kernel driver.

The kernel driver solves this issue by scanning all the write chunks not yet received by userspace (in all the write queues) for conflicting writes, each time a read request is received. If a conflicting write is found, the read is postponed until the conflicting write is acknowledged by the userspace. Once the write is received by userspace, returning the correct value becomes the userspace driver's responsibility.

4.3.2 Write-after-Write Hazard

BUSE enforces ordering of writes to the same sector by adding a sequence number to each write request. Having an atomic integer for every sector however would consume too much memory, hence we define configurable colli-

sion areas with shared atomic counters. The sequence number of each write request is part of the request sent to userspace, the userspace is then required to ignore late writes.

Write ordering is essential to correctness, since an application can issue two ordered writes (by two subsequent `write(2)` calls) into two different hardware dispatch queues (when rescheduled to different core).

4.4 Device Management and Configuration

BUSE relies on configfs for its configuration and management during runtime. Configfs is a virtual file system used for creating, managing and deleting kernel objects from userspace with a simple file-oriented API. It is very similar to sysfs, however sysfs operates on objects created by kernelspace.

The lifecycle of a BUSE block device is shown in Figure 4.2 and the individual states are described in the rest of this section.

Lifecycle management is an important part of the design. As mentioned in Section 3.1 before, it is important for the userspace driver to be able to reconnect to the kernel driver should it crash. This feature was especially useful during development of the BS3 driver.

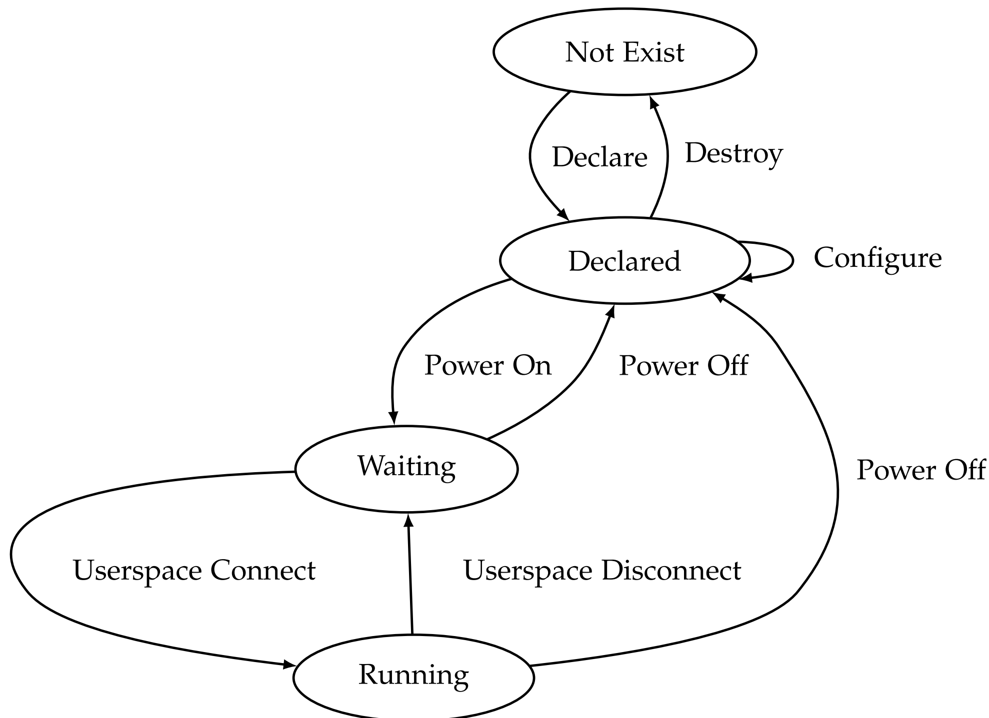

Figure 4.2: BUSE device life-cycle.

4.4.1 Declaration

When the BUSE kernel driver is loaded, no device will appear, as it first has to be declared via configfs. This has several advantages. For example, it is

not necessary to know the number of devices or their configuration options beforehand, and the number of devices and their options can be dynamically changed as needed.

A new block device is declared by creating a new directory in `/sys/kernel/config/buse`. The convention here is to name the directory with a number which will become the suffix of the newly created block device name. For example, creating the directory `/sys/kernel/config/buse/0` will declare a new block device `buse0`. The declaration of a BUSE block device is depicted in Listing 4.1.

Listing 4.1: BUSE Declaration.

```
$ mkdir /sys/kernel/config/buse/0
```

The device is now declared, but that still does not mean it is available as a block device (and indeed it will be missing from e.g. `lsblk(8)` output). The device still needs to be configured and powered on to appear in the system.

4.4.2 Configuration

All configurable options can be displayed by simply listing the directory content of the particular BUSE device as can be seen in Listing 4.2 for device `buse0`.

Listing 4.2: BUSE Configuration Options.

```
$ ls /sys/kernel/config/buse/0
blocksize
collision_area_size
hw_queues
no_scheduler
power
queue_depth
read_shm_size
size
write_chunk_size
write_shm_size
```

To get the current configuration value and to set a new value, `read(2)` and `write(2)` are used on the file corresponding to the option, respectively. Listing 4.3 shows how to find out what the configured block device size is, change it to 32 GB and read the new value.

When the user tries to configure an option with an invalid value, an error is returned and nothing is changed. Also, when the user tries to change any configuration option for device in the Waiting or Running states, an error is returned, as the device must be in the Declared state to be configurable.

Listing 4.3: BUSE Size Configuration.

```
# Read current device size (1GB)
$ cat /sys/kernel/config/buse/0/size
1073741824
```

```
# Set size to 32GB
$ echo $((32*1024*1024*1024)) > /sys/kernel/config/buse/0/size

# Read current device size (32GB)
$ cat /sys/kernel/config/buse/0/size
34359738368
```

We now briefly describe all available configuration parameters:

- **blocksize:** The minimum block size (in bytes) the block device can handle. Only 512 and 4096 are currently accepted (this is not a hard requirement and may be changed in the future, the restriction was imposed to minimize the testing surface).
- **hw_queues:** Number of hardware queues provided by the block device. This will be usually set to the total number of threads on the machine. However, for optimum performance, this should be fine-tuned, as it depends on the use case and the implementation of the userspace driver itself.
- **power:** Option for powering the device on and off (`1` = on, `0` = off).
- **queue_depth:** Depth of each hardware queue specified as the number of outstanding requests.
- **size:** Size (in bytes) of the block device. Note that if the userspace driver and file system support it, block devices can change its size while in use.
- **write_chunk_size:** Size (in bytes) of the chunk where write requests and corresponding data are batched. Chunk is the smallest unit of data sent to userspace (with the exception of flushes).
- **write_shm_size:** Size (in bytes) of the shared memory between the kernel and userspace of each write queue. Data of write chunks are stored in the shared memory, hence size of the shared memory limits the number of write requests pending in the userspace.
- **read_shm_size:** Size (in bytes) of the shared memory between the kernel and userspace of each read queue. For every read request, an appropriately-sized memory slot is reserved in the shared memory of the read queue, and its offset is sent to the userspace along with the read request.
- **collision_area_size:** Size (in bytes) of the area sharing the same write sequential counter. This is mainly for performance tuning.
- **no_scheduler:** If set to `1`, the I/O request should go directly to hardware dispatch queues, bypassing the staging area of software queues. This means no merging, reordering etc. of the I/O requests will be done.

4.4.3 Power On

BUSE devices can be powered on only when in the Declared state. By writing `1` to the `power` file, the device changes its mode to Waiting. In this state, the device creates all the necessary character devices (one pair per hardware queue) and waits until all character devices are bound by the userspace daemon. In other words, BUSE waits until the daemon calls `open(2)` on each of the character devices.

Only then BUSE registers the block device with blk-mq and a new `/dev/buseX` (in our example, `/dev/buse0`) device appears and can be used as regular block device with all I/O requests being served from userspace. Ultimately, this is the Running state.

4.4.4 Userspace Disconnect / Crash

The userspace daemon can crash or disconnect from BUSE for various reasons instead of shutting down gracefully. In that case, BUSE will notice a `release` on the character device and will immediately start to refuse all subsequent I/O requests. The block device will fallback to the Waiting state.

When the userspace daemon is connected again, all pending I/Os are redelivered to the userspace daemon (at-least-once delivery semantics) and the block device continues to process I/O requests, switching back to the Running state.

4.4.5 Power Off / Termination

Graceful shutdown is achieved with a simple handshake between the userspace daemon and the kernel module.

The userspace is the initiator of the shutdown. It writes a `0` into the `power` file, kicking off the shutdown sequence. No more I/O requests are accepted by the block device and termination chunks are sent to each queue. When the userspace daemon receives a termination chunk from a read/write queue, it can be certain that no more requests will be sent to it. Once the termination chunk is received for the last queue, the userspace driver may exit, leaving the device in the Declared state.

4.4.6 Destruction

To completely destroy the device, it has to be in the Declared state. `rmdir(2)` can then be called to finalize removal as shown in Listing 4.4.

Listing 4.4: BUSE Device Destruction.

```
# Make sure device 0 exists.
$ ls /sys/kernel/config/buse/
0

# Check its power state (powered off).
```

```
$ cat /sys/kernel/config/buse/0/power
0

# Destroy the device.
$ rmdir /sys/kernel/config/buse/0

# Make sure device 0 does not exist.
$ ls /sys/kernel/config/buse/ # no output
```

4.5 Usage Example

In this section, we briefly demonstrate how to use the BUSE API and how to write a simple userspace daemon which merely acknowledges I/O requests. The example however is otherwise complete, and the only missing part is the actual device-dependent read/write/flush logic.

The example is split into two parts, the read path and the write path.

4.5.1 Read Path

Listing 4.5 shows the implementation of a userspace daemon which handles read requests sent to the read queue number `{MINOR}` of the block device `/dev/buse{MAJOR}`. The program performs the following high-level steps:

1. Opens the read queue character device.
2. Memory-maps (`mmap`) the character device to get access to the shared memory area.
3. Calls `read` on the character device to receive a read request.
4. Parses the read request and performs the device-specific read operation.
5. Writes the result of the read operation to the correct shared memory location.
6. Acknowledges the read request by writing the shared memory offset to the character device.

Listing 4.5: BUSE Read Path API Usage in Python.

```
with open(f'/dev/buse{MAJOR}-r{MINOR}', 'r+b', buffering=0) as f:
  mm = mmap.mmap(f.fileno(), BUF_SIZE)
  while True:
    request = f.read(24)
    sector, len, offset = request[:8], request[8:16], request[16:24]
    offset_int = int.from_bytes(offset, byteorder='little')
    sector_int = int.from_bytes(sector, byteorder='little')
    len_int = int.from_bytes(len, byteorder='little')
    # Device-specific read operation
    data = os.pread(STORE, len_int * 512, sector_int * 512)
    mm[offset_int:offset_int + len_int * 512] = data
    f.write(offset)
  mm.close()
```

4.5.2 Write Path

Listing 4.6 shows the implementation of a userspace daemon which handles write (and flush) requests sent to the write queue number `{MINOR}` of the block device `/dev/buse{MAJOR}`. The program performs the following high-level steps:

1. Opens the write queue character device.
2. Memory-maps (`mmap`) the character device to get access to the shared memory area.
3. Calls `read` on character device to receive a write chunk.
4. Parses the write chunk (offset in shared memory and number of write requests).
5. For every write request in the write chunk, it parses the request and performs the device-specific write/flush operation.
6. Acknowledges the write chunk by writing the shared memory offset to the character device.

Listing 4.6: BUSE Write Path API Usage in Python.

```
with open(f'/dev/buse{MAJOR}-w{MINOR}', 'r+b', buffering=0) as f:
  mm = mmap.mmap(f.fileno(), BUF_SIZE)
  while True:
    request = f.read(16)
    offset, writes = request[:8], request[8:16]
    offset_int = int.from_bytes(offset, byteorder='little')
    writes_int = int.from_bytes(writes, byteorder='little')

    tmp = offset_int
    frontier = tmp + METADATA_SIZE
    for i in range(writes_int):
      write_io = mm[tmp:tmp+32]
      tmp = tmp + 32

      sector = int.from_bytes(write_io[:8], byteorder='little')
      len = int.from_bytes(write_io[8:16], byteorder='little')
      seq = int.from_bytes(write_io[16:24], byteorder='little')
      rfu = int.from_bytes(write_io[24:32], byteorder='little')

      # Device-specific write/flush operation

      os.pwrite(STORE, mm[frontier:frontier+len*512], sector*512)
      frontier += len*512
    f.write(offset)
  mm.close()
```

4.6 BUSE Userspace Daemon Library

To further simplify the development of userspace drivers, we provide a `buse` library implemented in Go which wraps the setup of the device and its graceful shutdown. We expect more (specialized) libraries to appear in the future.

4.6.1 Go Interface

To use the library, a receiver implementing the `buse.BuseReadWriter` interface has to be provided. The interface is shown in Listing 4.7.

Listing 4.7: BUSE Library Interface

```
// BuseReadWriter is an interface describing the basic
// operations needed to read from and write to a BUSE block
// device.
type BuseReadWriter interface {
    // BuseRead should read the extent starting at the
    // given sector with the given length. The read data
    // should be written to the provided sice. The chunk
    // is guaranteed to have sufficient capacity to hold
    // the data.
    //
    // This method is called by the BUSE library in
    // response to a read request received from the
    // kernel driver.
    BuseRead(sector, length int64, chunk []byte) error

    // BuseWrite should handle all writes stored in the
    // given chunk. The first argument holds the number
    // of writes in the chunk.
    //
    // This method is called by the BUSE library in
    // response to a write or flush request received
    // from the kernel driver.
    BuseWrite(writes int64, chunk []byte) error

    // BusePreRun is called immediately before the
    // device is started.
    BusePreRun()

    // BusePostRemove is called after the device is
    // removed.
    BusePostRemove()
}
```

4.6.2 Options

The library provides a structure for convenient setting of configuration options. It can be seen in Listing 4.8, with most of the options being direct counterparts of the configfs options. However, there are some new options which we will briefly describe:

Listing 4.8: BUSE library options.

```
type Options struct {
    Durable        bool
    WriteChunkSize int64
    BlockSize      int64
    Threads        int
    Major          int64
```

```
    WriteShmSize   int64
    ReadShmSize    int64
    Size           int64
    CollisionArea  int64
    QueueDepth     int64
    Scheduler      bool
}
```

- **Durable:** If it is set to `false` then a weak flush mode is enabled.
- **Threads:** Number of threads (more precisely goroutines) to use. This implies the number of hardware queues.
- **Major:** The numeric suffix for the created block device. I.e. if Major = 7 then device `/dev/buse7` is created with configfs directory in `/sys/config/kernel/buse/7`.

4.6.3 Usage

Once the required interface is implemented (as `buseReadWriter` in this example), the following library functions can be called.

buse.New(buseReadWriter, buse.Options{})

This function is a constructor for a BUSE device, which calls the provided callbacks when I/O requests are received. After this function call, the device is in Waiting state.

buse.Run()

Transition into the Running state is done with this call. It starts the requested number of goroutines to serve the appropriate BUSE queues. This function calls the `BusePreRun` callback.

buse.StopDevice()

This function just writes a `0` into the `power` file, which initiates the shutdown process and sending of termination chunks. The library correctly handles the termination chunks and successfully returns from the buse.Run function. This function is usually called in a signal handler as a response to SIGINT or SIGTERM, etc.

buse.RemoveDevice()

This function just calls `rmdir(2)` on the appropriate directory to destroy the device and calls the `BusePostRemove` callback.

Chapter 5

BUSE | Evaluation

We evaluate BUSE performance on multi-core machines and compare it with NBD and TCMU. The Flexible I/O tester (FIO) (Axboe [28]) microbenchmark was used to benchmark the throughput of a "null backend" of each storage technology in turn. The *null backend* is a minimal implementation of a userspace driver (irrespective of technology) which merely acknowledges all I/O requests.

5.1 Performance Evaluation in Cloud

All experiments were run in the Amazon Web Services (AWS) Cloud. Unless otherwise stated, presented data estimator is always mean with 95 % confidence intervals computed using the bootstrap method (Dragicevic, Efron and Tibshirani, Efron [29, 30, 31]).

Recently, cloud environment is becoming quite popular for performance evaluation (Bulej et al. [32]). We briefly discuss the main advantages and potential problems of benchmarking in the cloud in the rest of this section.

5.1.1 Benefits

Unbounded hardware resources for experiments parallelization. Resource requirements are not constant during the development process and the peak is during the performance evaluation phase, where large amounts of measurements need to be taken. Unbounded hardware resources provide the ability to massively parallelize experiments, allowing us to run more experiments and obtain more robust data.

Third party reproducibility. Using a public cloud service makes all results instantly reproducible with the same software and hardware setup.

Access to a variety of hardware configurations, including the most recent. Performing experiments on a variety of hardware with the access to the most

recent ones makes the experiments more relevant.

5.1.2 Drawbacks Mitigation

The potential downside to the measurement in cloud is lack of target hardware control, hypervisor overhead and noise generated by other tenants sharing the same host (Iosup, Yigitbasi, and Epema, Laaber, Scheuner, and Leitner, Leitner and Cito [33, 34, 35]). For storage benchmarks, these drawbacks have minimal impact, especially in the following environment:

Hypervisor with little to no overhead. Amazon Elastic Cloud (EC2) (Amazon [36]) hypervisor, Nitro (Amazon [37]), has virtualization overhead of less than 1 % (Gregg [38]). It is negligible for storage experiments.

Reserved resources. We use only instances with reserved vCPUs, i.e. there is no virtual machine over-provisioning.

Randomization. We randomize experiment start time, i.e. we perform every benchmark iteration at different time of the day and different day of the week.

Benchmark isolation. New machine instance is spawned for every benchmark executed. It prevents shared resource pollution and minimizes the provider's performance limiting/boosting features. Since instances of our choice are billed per second, it does not have negative impact on the cost of the experiments.

5.2 Experimental Setup

5.2.1 Software Configuration

We used Ubuntu 21.04 with kernel version 5.11.0-22-generic, Go version 1.16.2 and FIO 3.25 in all experiments.

5.2.2 Hardware Configuration

General-purpose Elastic Compute Cloud (EC2) instances (the m5 family) were chosen for experiments as they represent the common data center/on-premise server hardware.

Later in chapter Chapter 9, we will be benchmarking BS3. As BS3 is network-attached storage, instances with the highest available network throughput will be needed. At the same time, we would like to test both BUSE and BS3 on the same hardware. Therefore we chose the m5zn EC2 instance family.

All instance types in the m5zn family are described in Table 5.1. The interesting hardware details of the instance family are:

- 2nd Generation Intel Xeon Scalable Processors — Cascade Lake.
- Model Name: Intel(R) Xeon(R) Platinum 8252C CPU @ 3.80 GHz
- Base CPU frequency 3.8 GHz.
- All-core turbo CPU frequency up to 4.5 GHz.
- Up to 100 Gbps of network bandwidth on the largest instance size.
- 12 cores (24 vCPUs — threads) per one socket, 2 sockets in total.
- CPU architecture captured by `hwloc` (Broquedis et al. [39]) can be seen on Figure 5.1.

Table 5.1: EC2 m5zn family instance types. All types are without dedicated NVMe SSD and only EBS volumes are offered for storage. Instances with network bandwidth labelled "Up to X" implement limiting/-boosting features. m5zn.12xlarge is the largest instance spanning the whole underlying physical node.

Instance Type	vCPUs (Threads)	Memory (GiB)	NVMe SSD (GiB)	Network Bandwidth (Gb/s)	EBS Bandwidth (Mb/s)
m5zn.large	2	8	None	Up to 25	Up to 3,170
m5zn.xlarge	4	16	None	Up to 25	Up to 3,170
m5zn.2xlarge	8	32	None	Up to 25	3,170
m5zn.3xlarge	12	48	None	Up to 25	4,750
m5zn.6xlarge	24	96	None	50	9,500
m5zn.12xlarge	48	192	None	100	19,000

5.2.3 Null User-space Implementation

For each technology of interest, we implemented a "null driver" which merely acknowledges I/O requests sent by the kernel. I.e., the written data are not written anywhere and all read requests are acknowledged immediately without writing data into the read buffer. The semantics of the read path is that of `/dev/zero` and the semantics of the write path is that of `/dev/null`.

This will allows us to benchmark the maximum throughput each technology has to offer. For each technology, this establishes the theoretical upper bound; we cannot possibly hope for higher throughput or lower latency with any practical implementation of a block device.

Capacity of all block devices was set to 1 TB and block size of the device to 4 kB.

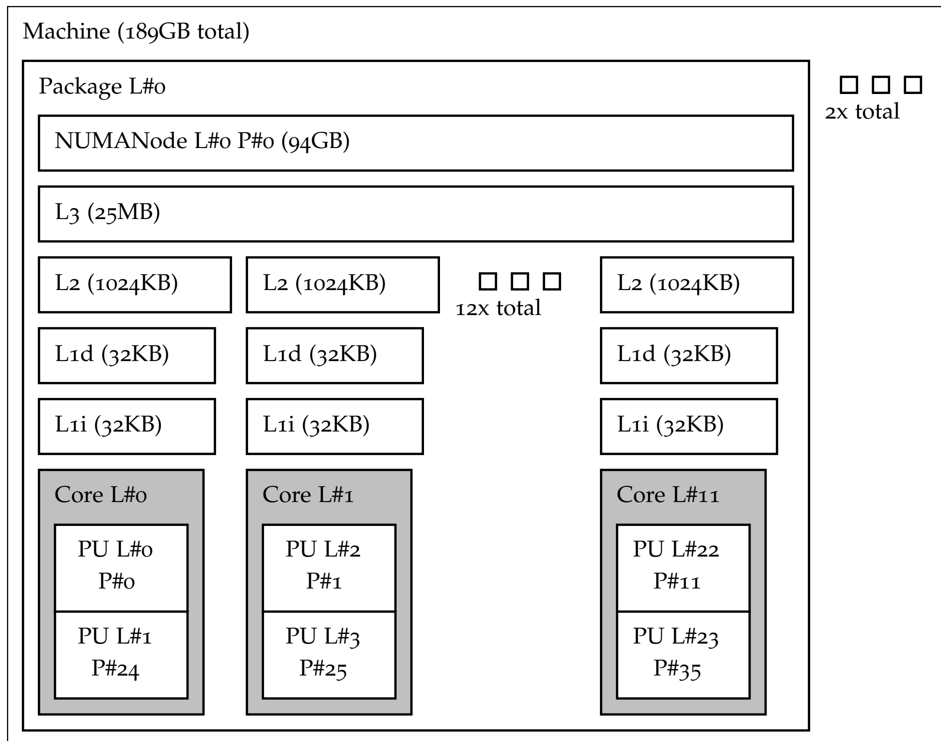

Figure 5.1: CPU architecture of Amazon EC2 m5zn instance host. It has 2 NUMA nodes, each with 25 MB L3 cache and 12 physical cores with hyper-threading. Every core has 32 kB L1 instruction cache, 32 kB L1 data cache and 1 MB L2 cache.

bs3-null We implemented a null driver for BUSE using the `buse` Go library which will also be used later to implement BS3. The configuration of the BUSE module was the following:

- Write Shared Memory: 512 MB
- Write Chunk Size: 32 MB
- Write Durable: True
- Read Shared Memory: 512 MB

nbd-go-null We implemented a null Go plugin for the high-performance NBD daemon nbdkit version `1.24.1-2ubuntu1` and used the Linux kernel NBD client configured with nbd-client version `1:3.21-1build2`. NBD specific settings were as follows:

- Multi-connection: Allowed
- Connection Type: Unix Socket
- Number of Connections: Total Number of vCPUs

tcmu-go-null go-tcmu version `20180105` was used to implement a null userspace driver for TCMU.

5.3 Microbenchmarks

We ran FIO sequential write and sequential read benchmark with following common parameters:

- `ioengine=io_uring` to use io_uring to submit I/O requests.
- `end_fsync=1` to issue a sync when the write stage completes.
- `direct=1` to use the block device directly, i.e. skip the page cache.
- `runtime=60` to run for 60 seconds to obtain representative results.

5.3.1 I/O Parameter Sensitivity

This section describes how read and write throughput of competing technologies depend on the I/O parameters of the synthetic benchmark.

Because it is unfeasible to present results for all instance types in the m5zn family with all the combinations of I/O parameters in a comprehensible form, we selected an "average" machine from the m5zn family, the m5zn.3xlarge, and described the results we obtained for this instance type. This machine utilizes a common 6-core CPU with hyper-threading. It provides 12 vCPUs and 48 GiB of memory. We verified that using this instance type does not introduce any bias and leads to a fair comparison.

Figure 5.2 and Figure 5.3 show how read and write throughput, respectively, depend on the number of jobs, the I/O depth and the block size. Every row in each figure represents a different I/O depth and every column a different number of I/O jobs. (Please note that the I/O depth parameter specifies the I/O depth of each job, not a total across the jobs.)

Scaling with block size. All storage solutions perform better with larger block size until around 256 kB. The scalability with block size is expected since the limiting factor is the number of IOPS.

Throughput and working set size. The highest throughput will be reported for runs where the total data size of outstanding writes is near the total capacity of L3 caches. When the size is lower, then the cache is never fully utilized and when it is larger, cache misses occur for outstanding writes, leading to lower performance.

If we take the figure for writes, we can for example see the local maximum when running 6 jobs with 8 outstanding writes each and a block size of 256 kB. This means, that $6 \cdot 8 \cdot 256\,\text{kB} = 12\,\text{MB}$ is the maximum total size of outstanding writes. Since the m5zn.3xlarge machine has 12 vCPUs (out of the 24 vCPUs total per socket), we should get an L3 cache figure which is half the total L3 cache of the entire machine. L3 cache for one socket is 25 MB, therefore 12.5 MB for the m5zn.3xlarge instance. This corresponds with the empirical

observation that the highest possible throughput for this instance is achieved when there is ~12 MB of writes in flight.

Scaling with number of jobs. BUSE is the only solution which significantly scales when running more than one I/O job. This is because NBD and TCMU always use only one hardware queue of the Linux storage stack layer. BUSE is able to use all of the available queues.

BUSE has 2x – 6x higher throughput for writes and 2x – 3x for reads with 12 vCPUs. In vast majority of cases, BUSE performs better than both NBD and TCMU. For experiments with multiple jobs and block size higher than 16 kB, BUSE is up to 6x faster for writes than others with peak throughput of approximately 40 GB/s with 12 jobs and I/O depth of 1 per job. For reads, it is up to 3x faster than others with peak throughput of approximately 32 GB/s.

5.3.2 Scaling with number of vCPUs

To evaluate how throughput of competing technologies depends on the number of vCPUs, we chose the parameters as follows:

- I/O depth of 4.
- Block size of 64 kB.
- Number of I/O jobs equal to the number of vCPUs.

As can be seen in Section 5.3.1, no storage technology should benefit from this choice.

The results can be seen in Figures 5.4 and 5.5 for writes and reads respectively. Exact values can futher be seen in Tables 5.2 and 5.3. Note that the largest single-socket instance has 24 vCPUs. Instance with 48 vCPUs is a NUMA machine spanning 2 sockets.

BUSE is the only significantly scaling technology. For the write path, only BUSE scales. With 2 vCPUs, BUSE has approximately 2x higher throughput than others. With 24 vCPUs, BUSE has throughput 5x higher than NBD and 8x higher than TCMU. Finally with 48 vCPUs, it has 8x and 31x higher throughput. Maximum write bandwidth is 114 GB/s for BUSE, 14 GB/s for NBD and 7 GB/s for TCMU.

Similarly for the read path. Only BUSE scales significantly and has always higher throughput than others. With 24 vCPUs BUSE has 3x higher throughput than NBD and 6x higher than TCMU. With 48 vCPUs, BUSE has 6x higher throughput than NBD and 17x higher than TCMU. Maximum read bandwidth is 72 GB/s for BUSE, 12 GB/s for NBD and 6 GB/s for TCMU.

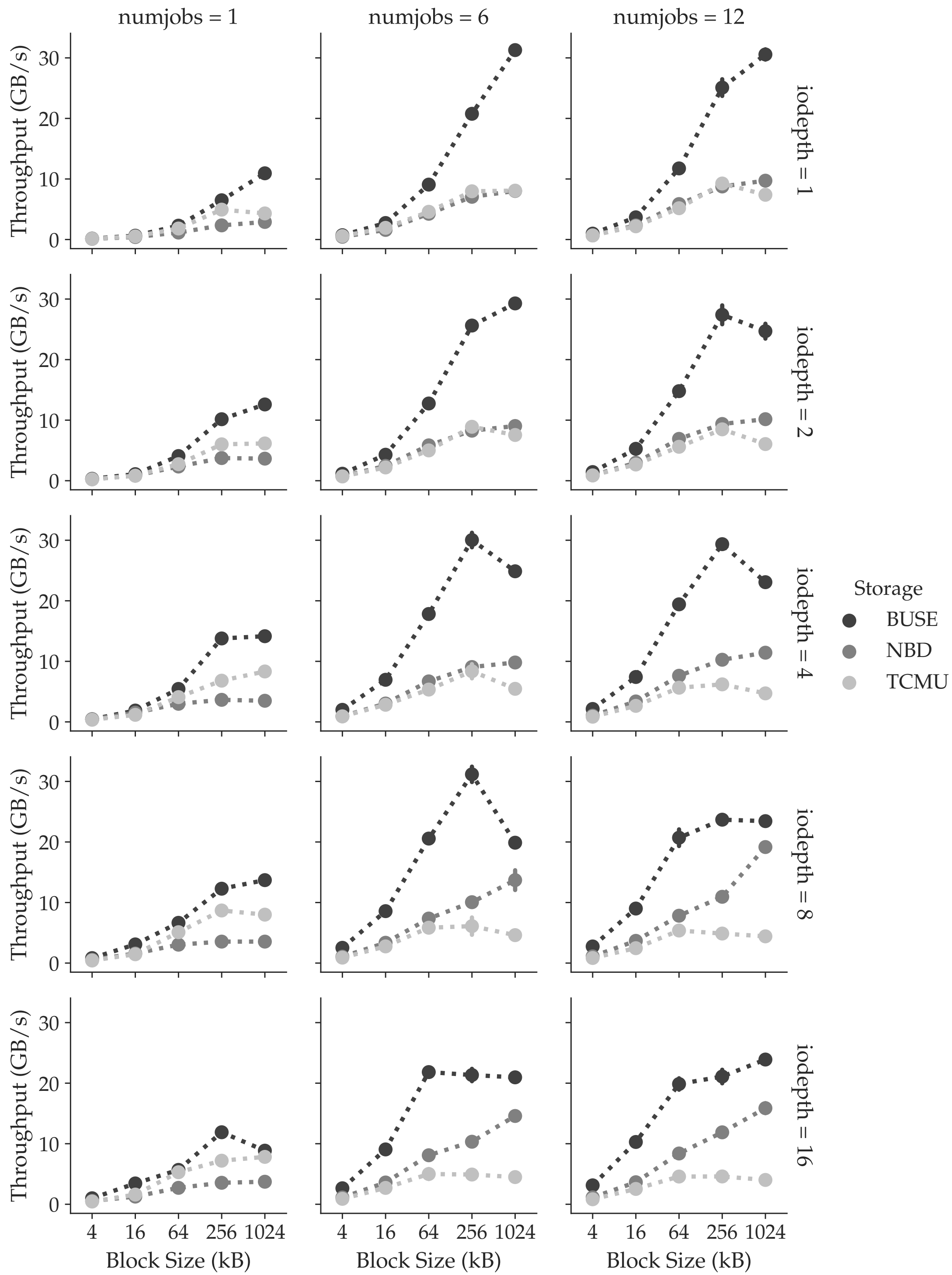

Figure 5.2: Read throughput dependency on block size with different I/O parameters on m5zn.3xlarge instance (12 vCPUs). Single-threaded results fall below 12 GB/s for all solutions. BUSE scales significantly with the number of I/O jobs, peaking around 32 GB/s. BUSE is approximately 2x – 3x times faster than others.

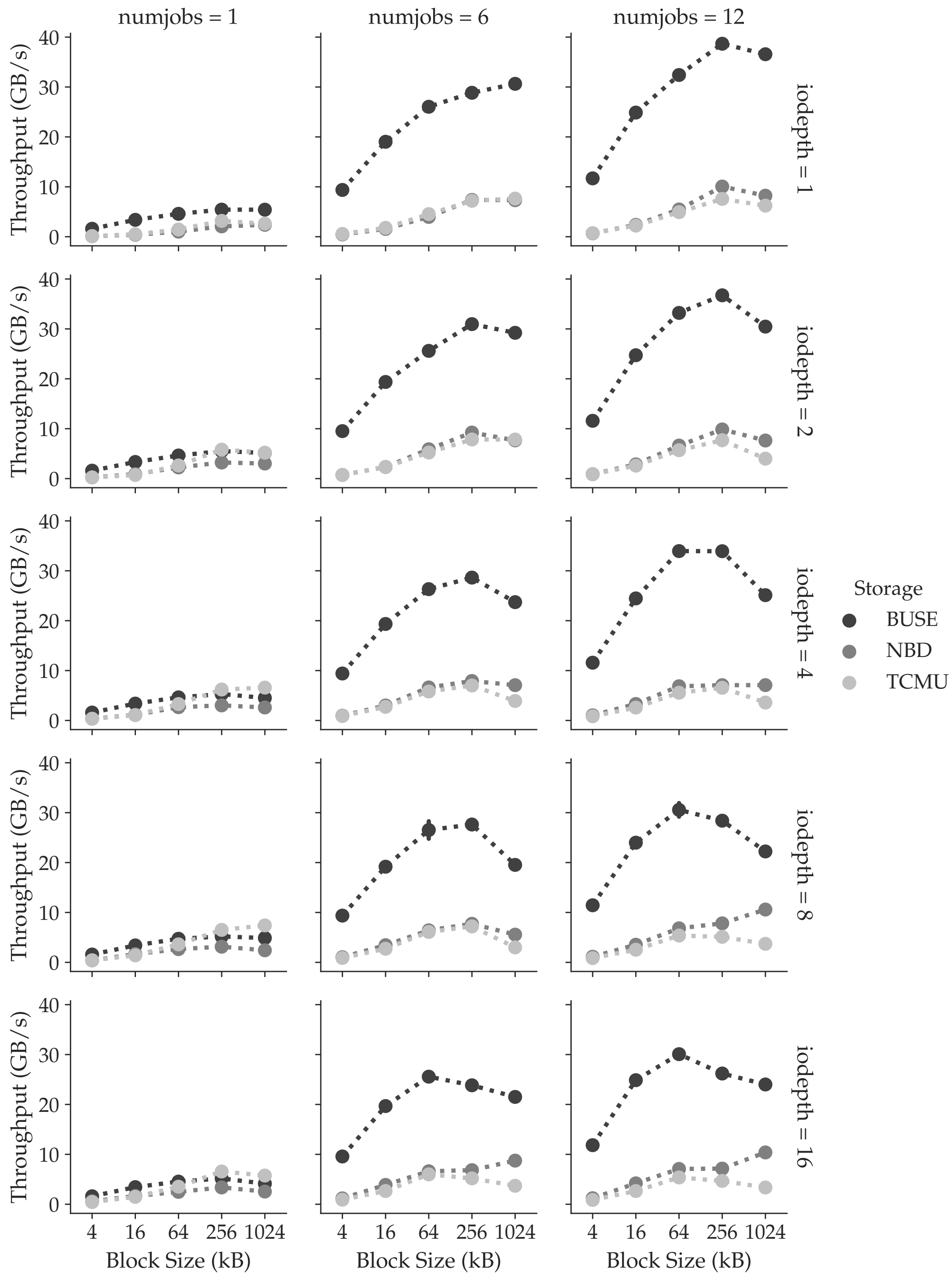

Figure 5.3: Write throughput dependency on block size with different I/O parameters on m5zn.3xlarge instance (12 vCPUs). Single-threaded results fall below 10 GB/s for all solutions. Only BUSE scales with the number of I/O jobs, peaking around 38 GB/s. BUSE is approximately 2x – 6x times faster than others.

BUSE is NUMA-aware on Write Path. BUSE is NUMA-aware on the write path because the shared memory buffers are allocated in a NUMA-aware way (from the local NUMA node).

BUSE is NUMA-aware on Read Path. Read path is fundamentally different from the write path. On the read path, we allocate a space in the shared memory and pass it to the userspace, which fills it with the requested data (or does nothing in case of null backend). The kernel part is then notified and a kernel thread in the context of the userspace process copies data from shared memory to the I/O command's buffer.

The issue is that the userspace thread performing the copy can be scheduled to a different NUMA node than memory was allocated from. To make BUSE scale on NUMA, we started one userspace daemon per NUMA node and pinned every daemon to the set of cores corresponding to the appropriate NUMA node.

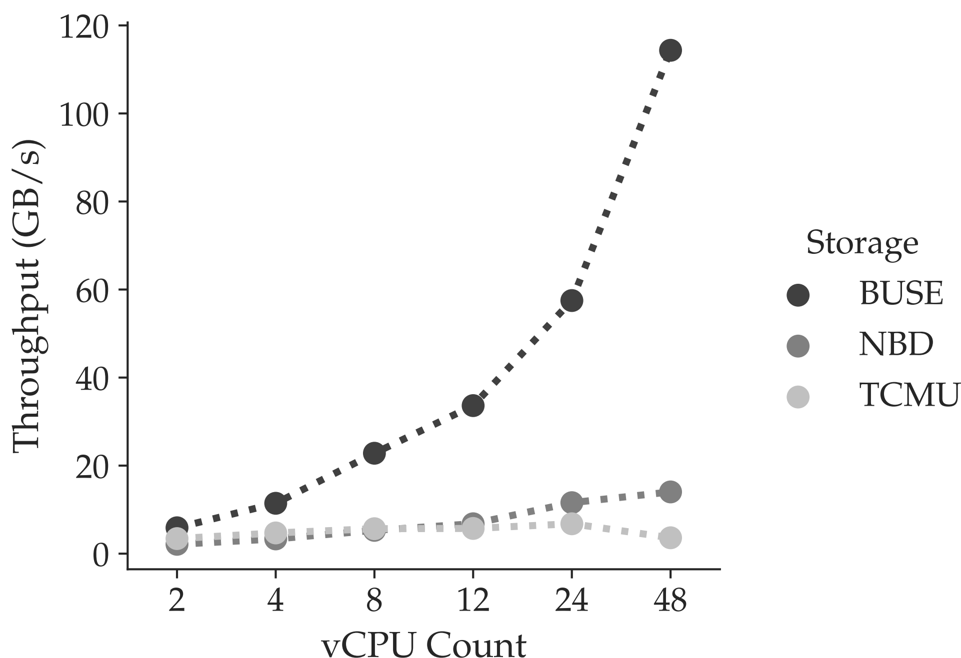

Figure 5.4: Write throughput dependency on vCPU count when I/O depth is 4 and block size 64 kB. The number of I/O jobs is equal to the number of vCPUs. BUSE has the highest throughput regardless of vCPU count. Furthermore, it is the only technology which scales with the number of vCPUs. BUSE is approximately 2x – 8x faster than others. Note the NUMA scaling (48 vCPUs are spread across 2 sockets).

Table 5.2: Write throughput for different instance types (scaling with number of vCPUs). See corresponding Figure 5.4 for detailed description.

Instance Type	vCPUs (Threads)	BUSE (GB/s)	NBD (GB/s)	TCMU (GB/s)
m5zn.large	2	5.88 (-0.09, +0.09)	2.12 (-0.02, +0.02)	3.42 (-0.05, +0.06)
m5zn.xlarge	4	11.44 (-0.06, +0.05)	3.35 (-0.07, +0.04)	4.71 (-0.09, +0.09)
m5zn.2xlarge	8	22.81 (-0.89, +0.58)	5.29 (-0.16, +0.18)	5.67 (-0.06, +0.06)
m5zn.3xlarge	12	33.63 (-0.82, +0.53)	6.79 (-0.42, +0.23)	5.73 (-0.11, +0.12)
m5zn.6xlarge	24	57.50 (-0.08, +0.08)	11.58 (-0.17, +0.19)	6.79 (-0.05, +0.07)
m5zn.12xlarge	48	114.35 (-0.71, +0.68)	14.02 (-0.28, +0.27)	3.61 (-0.10, +0.09)

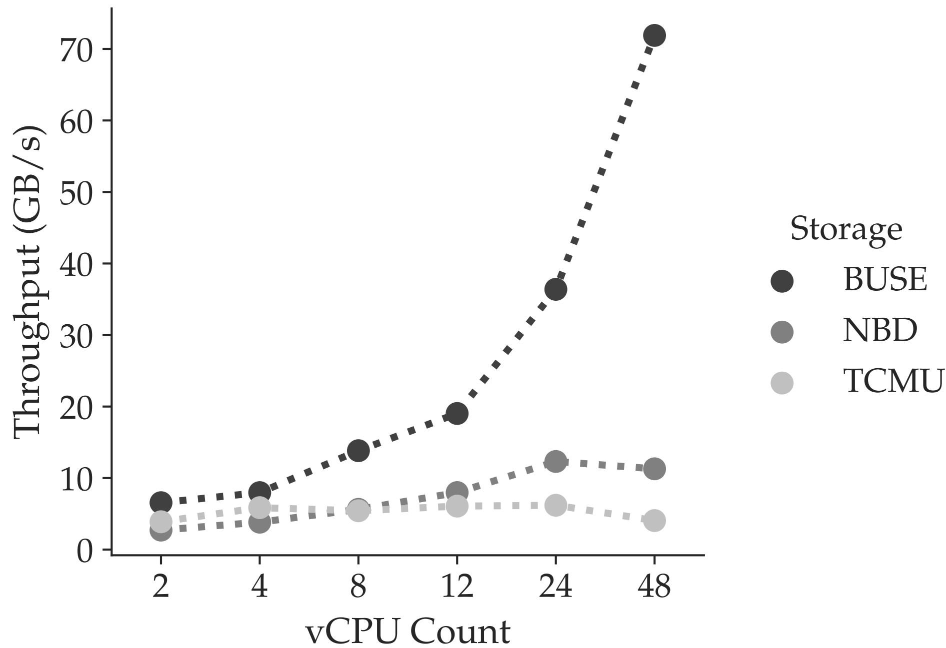

Figure 5.5: Read throughput dependency on vCPU count when I/O depth is 4 and block size 64 kB. The number of I/O jobs is equal to the number of vCPUs. BUSE has the highest throughput regardless of vCPU count. Furthermore, it is the only technology which scales with number of vCPUs. BUSE is approximately 2x – 17x faster than others.

Table 5.3: Read throughput for different instance types (scaling with number of vCPUs). See corresponding Figure 5.5 for detailed description.

Instance Type	vCPUs (Threads)	BUSE (GB/s)	NBD (GB/s)	TCMU (GB/s)
m5zn.large	2	6.56 (-0.05, +0.06)	2.72 (-0.04, +0.05)	3.89 (-0.15, +0.12)
m5zn.xlarge	4	7.96 (-0.12, +0.14)	3.83 (-0.05, +0.05)	5.86 (-0.17, +0.20)
m5zn.2xlarge	8	13.83 (-0.27, +0.21)	5.63 (-0.24, +0.22)	5.42 (-0.09, +0.10)
m5zn.3xlarge	12	19.03 (-0.42, +0.41)	7.97 (-0.28, +0.44)	6.08 (-0.40, +0.40)
m5zn.6xlarge	24	36.39 (-0.29, +0.19)	12.32 (-0.16, +0.17)	6.19 (-0.05, +0.05)
m5zn.12xlarge	48	71.89 (-0.34, +0.29)	11.29 (-0.56, +0.42)	4.05 (-0.17, +0.17)

Additional Benchmark: BDUS

As already mentioned in Section 3.2.5, we noticed the publication of BDUS (Faria et al. [27]) only a few days before the thesis deadline. Despite that, we were able to perform a limited amount of measurements and compare it to all storage technologies mentioned before. We implemented yet another "null" implementation for BDUS, just as we did with all the other technologies. Figures 5.6 and 5.7 show bar charts for read and write throughput respectively with the corresponding values presented in Tables 5.4 and 5.5. We present results for the following cases:

- 48 vCPUs is a 2-socket NUMA machine.
- 24 vCPUs is a single socket machine with the maximum amount of vCPUs in a single socket.
- 12 and 8 vCPUs additionally, to observe the trend with lower amount of vCPUs. Lower vCPU counts are omitted for brevity, since BDUS has approximately the same throughput for all cases, and the throughput of other technologies with low vCPU count is presented in the previous section.

BDUS throughput is comparable to that of TCMU and NBD. This result is unsurprising for the following reasons:

- Only a single hardware dispatch queue is used, and so performance does not scale with the number of CPUs.
- No batching is used. This means high overhead for communication with userspace per every I/O request.
- `copy_to_user` is used to copy payloads of I/O requests. In our experience, this is detrimental to performance due to additional security checks performed by the function.

Since its throughput is similar to TCMU, BUSE evaluation remains valid for BDUS as well.

Interestingly, for high vCPU count (12, 24) BDUS performs 2x worse than NBD, which also uses only a single hardware dispatch queue.

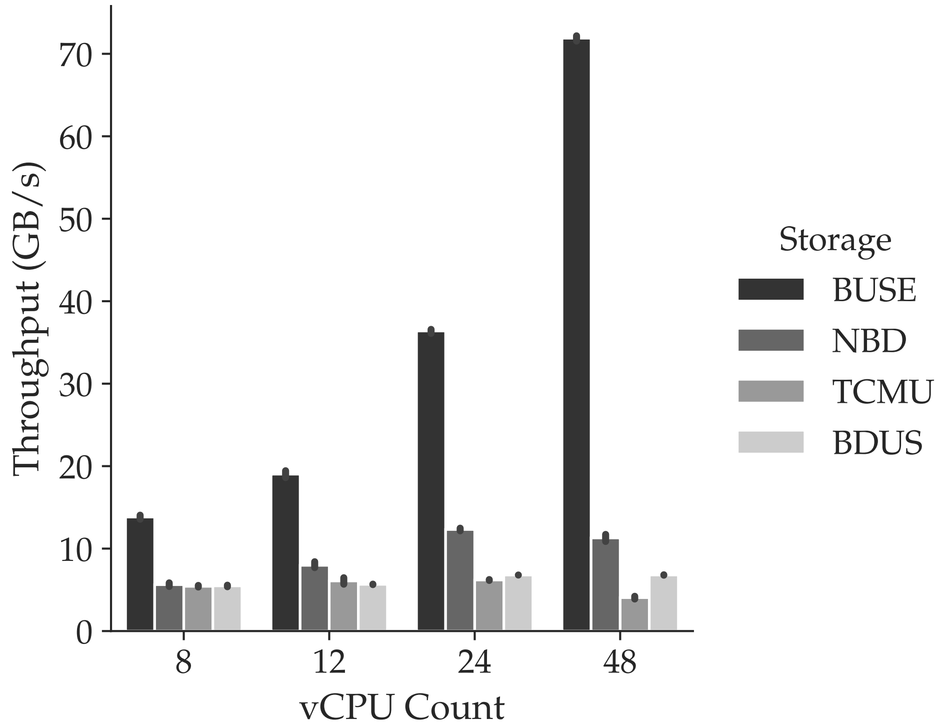

Figure 5.6: Read throughput for different number of vCPUs and different storage technologies. Parameters are the same as in previous experiment: I/O depth is 4, block size is 64 kB and number of I/O jobs is equal to vCPU count. BUSE dominates all technologies and has up to 19x higher throughput than BDUS. BDUS does not perform better than NBD or TCMU and can be up to 2x worse than NBD in multi-threaded workloads.

Table 5.4: Read throughput for different number of vCPUs (scaling with number of vCPUs). See corresponding Figure 5.6 for detailed explanation.

vCPUs (Threads)	BUSE (GB/s)	NBD (GB/s)	TCMU (GB/s)	BDUS (GB/s)
8	13.83 (-0.27, +0.23)	5.63 (-0.22, +0.22)	5.42 (-0.09, +0.11)	5.48 (-0.11, +0.08)
12	19.03 (-0.42, +0.41)	7.97 (-0.28, +0.44)	6.08 (-0.40, +0.39)	5.67 (-0.03, +0.03)
24	36.39 (-0.29, +0.19)	12.32 (-0.16, +0.16)	6.19 (-0.05, +0.05)	6.80 (-0.00, +0.00)
48	71.89 (-0.35, +0.29)	11.29 (-0.43, +0.43)	4.05 (-0.17, +0.17)	6.80 (-0.03, +0.03)

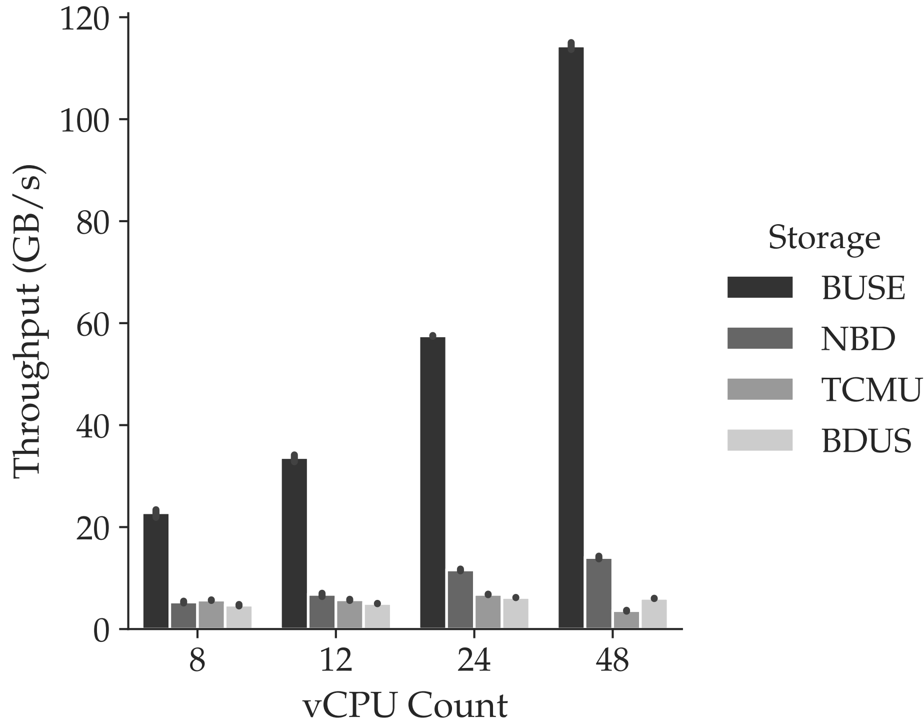

Figure 5.7: Write throughput for different number of vCPUs and different storage technologies. Parameters are the same as in previous experiment: I/O depth is 4, block size is 64 kB and number of I/O jobs is equal to vCPU count. BUSE dominates all technologies and has up to 19x higher throughput than BDUS. BDUS does not perform better than NBD or TCMU and can be up to 2x worse than NBD in multi-threaded workloads.

Table 5.5: Write throughput for different number of vCPUs (scaling with number of vCPUs). See corresponding Figure 5.7 for detailed explanation.

vCPUs (Threads)	BUSE (GB/s)	NBD (GB/s)	TCMU (GB/s)	BDUS (GB/s)
8	22.81 (-0.89, +0.60)	5.29 (-0.18, +0.18)	5.67 (-0.06, +0.06)	4.68 (-0.15, +0.15)
12	33.63 (-0.82, +0.52)	6.79 (-0.41, +0.24)	5.73 (-0.12, +0.12)	5.01 (-0.03, +0.03)
24	57.50 (-0.08, +0.08)	11.58 (-0.17, +0.19)	6.79 (-0.05, +0.07)	6.19 (-0.02, +0.02)
48	114.35 (-0.68, +0.68)	14.02 (-0.26, +0.28)	3.61 (-0.10, +0.08)	6.01 (-0.03, +0.03)

Chapter 6

BS3 | Background Knowledge

6.1 Object Storage

Object storage is a storage architecture which stores data as variable-sized *objects*, as opposed to block storage which stores data in fix-sized *blocks*. Objects do not have any common definition, but usually are identified by a unique identifier called the *key* and store arbitrary unstructured data.

An API (often an HTTP API) provides means of uploading objects to and downloading objects from the storage. Often it's impossible to modify parts of objects and a new object with the same name must be uploaded instead, or it incurs a performance penalty.

Nowadays, vast majority of storage is actually object storage (Wikipedia [40]). Over 70 % of top 100 supercomputers (including Fugaku, Titan and Sequoia) use Lustre (Braam [41]), a distributed object filesystem. Furthermore, object storage is the de-facto standard storage of social networks data, e.g. Facebook is using their Haystack object store (Facebook [42]) to store photos.

Some of the well-known object storage technologies are Amazon S3 (Amazon [43]), DigitalOcean Spaces (DigitalOcean [44]), OpenStack Swift (OpenStack [45]), Microsoft Azure Blob Storage (Microsoft [46]) and Google Cloud Storage (Google [47]).

Probably the most notable characteristics of publicly available object storages are their scalability and reliability. For example Amazon S3 has a practically unlimited capacity and throughput and a stated reliability of 99.999999999 % (eleven nines).

6.2 Log-structured Storage

During writes, instead of modifying the previously written data, we can instead append a "write record" describing the original intent to write the data

in the first place. This approach is called *journalling* or *log-structured writes*, as all writes are "logged" into a "journal" before they are actually carried out.

For file systems, this is what makes crash recovery possible. File systems organized in this manner are then called *journalling file systems*. For example, the well-known file systems Ext3/4 and XFS are all journalling file systems.

In the general sense, a "log-structured" technology will tend to avoid rewriting data. In this context, it is often called "append-only": data is replaced by writing new data into the log rather then replacing old data.

Chapter 7

BS3 | Problem Statement & Related Work

7.1 Network Attached Disk

A *network attached disk* is a block device disaggregated from the system which uses it. Disaggregation makes it possible to achieve the following properties which are hard or even impossible to achieve with local storage:

- Unlimited scalability.
- Unlimited capacity.
- Extreme durability.
- High performance.
- Accessible from multiple places.

The usability of the disk is determined by the local interface, i.e. how the device is presented to the local system. Ideally the disk is accessible as a block device, just like any other locally attached storage. This makes it possible to use the disk with other block layer technologies such as dm-crypt, lvm, md-raid etc. Furthermore, any existing application can use the disk without modification. The standard Unix block device interface was described in Section 2.1.

Recently, object storages have gained significant popularity, and most cloud providers provide an object storage of some kind. The object storage often complements a block storage offering, usually at a significant discount and with favorable parameters of the service. This business model is possible due to the scale in which object storages are typically deployed.

Ideally we would be able to combine the interface and semantics of a locally-attached block device with the many advantages of object storage. This chapter explores that idea.

7.2 Related Work

7.2.1 Proprietary

Large Providers' Block Storage

Most cloud service providers (more specifically virtual machines providers) offer some proprietary virtual block device service. Little is usually known about how these services work.

Amazon Elastic Block Store (EBS). Amazon provides EBS (Amazon [48]), virtual block storage remotely attached to a centralized storage. It heavily relies on offloading of operations for encryption, rate limiting etc. to a dedicated Nitro card (Amazon [37]). EBS provides snapshots and is billed per IOPS and capacity.

Google Persistent Disk. Google's alternative to EBS is Google Persistent Disk (Google [49]). It is a direct competitor with virtually the same features and billing scheme.

Microsoft Azure Disk Storage. Microsoft is another large player in the cloud market, providing a virtual block device which can be connected to Azure virtual machines (Microsoft [50]).

Third-party Block Storage

Plenty of third-party solutions exist, trying to offer better performance, price, scalability or durability than the native products of cloud providers. Unfortunately, all of the solutions are closed-source.

However, by studying one of these commercial products, CloudBD (CloudBD [51]), we were able to understand how it works. Our findings are presented below.

CloudBD. CloudBD is a direct competitor to BS3. It uses an S3-compatible object storage service as a backend to provide network attached storage. It seems that CloudBD relies on NBD, which is corroborated by the fact that with every CloudBD drive a `/dev/nbdX` special block file is created and an application called `cbdkit` is running. We believe that `cbdkit` is most probably forked `nbdkit`, the high-performance NBD server described in Section 3.2.1.

From the conducted experiments and from official CloudBD documentation (CloudBD [51]), we deduced that some kind of direct mapping of parts of the logical block space onto S3 objects is probably used. This is in contrast to BS3 which uses objects to implement log-structured storage.

Log-structured writes. Other proprietary solutions exist which use log-structured writes to attain higher performance and better consistency properties. Unfortunately, it is impossible to provide similar information as in the case of CloudBD since they do not provide any trial version which could be explored in a controlled local environment. These systems are:

- Microsoft Avere (Microsoft [52]).
- ObjectiveFS (ObjectiveFS [53]).
- Panzura CloudFS (Panzura [54]).

7.2.2 Academia

Blizzard

Blizzard (Mickens et al. [55]) is built on Flat Datacenter Storage (Nightingale et al. [56]). The technology is quite dated, but in its day achieved respectable performance by striping writes over a set of spindle drives available over a dedicated network. Blizzard provides crash consistency and introduces non-durable flush semantics, which were a direct inspiration for BUSE in weak flush mode (described in Section 4.2.1).

Petal

Petal (Lee and Thekkath [57]) is an example of a scalable network virtual disk from the 1990s. Petal's network was composed of several servers and every server had a couple hard drives. This network provided distributed and highly available containers which were mountable on the client side. Petal striped writes onto multiple drives which promoted horizontal scalability.

Salus

Salus (Wang et al. [58]) is a virtual block device implemented on top of HDFS/HBase (Apache HBase [59]). It uses batching to improve write performance and performs periodic checkpointing. Ordered-commit semantics is used during normal operation and prefix semantics is used for crash recovery (Mickens et al. [55]).

Ursa

Ursa (Li et al. [60]) is similar to Salus, however it uses a hybrid setup of fast low-latency SSDs and large capacity high-latency HDDs. Instead of using SSDs for caching—as is often the case in SSD/HDD deployments—they are used to store primary replicas of data while backup replicas are stored on HDDs. Ursa uses NBD.

BlueSky

BlueSky (Vrable, Savage, and Voelker [61]) provides an NFS/CIFS interface to cloud storage such as Amazon S3 and Microsoft Azure. It leverages the reliability of hosted cloud storage. However performance aspects are not addressed sufficiently due to large WAN latencies. A caching solution was proposed which would reside on the local side but was subject of future research.

Ceph Rados Block Device (RBD)

Ceph (Weil et al., Weil et al. [62, 63]) RBD is a popular open-source solution how to create a scalable and reliable object storage not only on commodity hardware. RBD creates a block device backed by an image in the object storage. It has support in Linux mainline kernel as well as in userspace via *librbd*. However, its performance as a block device is mediocre, since it translates every block device write into a network operation performing write on the backend. This introduces extreme latencies and performance-wise is much worse than the other presented solutions.

Chapter 8

BS3 | Architecture & Implementation

BS3 is a network-attached block device using the S3 protocol to communicate with any S3-compatible object storage. It handles traditional requests like read, write and flush and translates them to S3 operations. It is implemented in Go and uses the `buse` library presented in Section 4.6.

Although it was primarily designed for high throughput and to be able to saturate up to 100 Gbps networks in data centers, it can be also used as a network block device in a midscale or personal setup.

When used with Amazon S3, it can provide a virtually unlimited block device with unbeatable durability of 99.999999999 % (eleven nines). BS3 installed on an Amazon EC2 instance can be in some cases a faster and cheaper alternative to Amazon EBS or even a local NVMe drive. In combination with on-premise S3-compatible storage like MinIO, it can be a high-performance alternative to remotely attached block devices or a single-client network file system.

8.1 Overview

Every write request has to be mapped to an S3-compatible operation. A naive implementation would map each Logical Block Address (LBA, a block number) to an object in S3. Although that would work, it would lead to extremely poor performance due to high latency of S3 operations. Furthermore, this approach would have consistency issues, making it hard to recover from failures.

8.1.1 Write Path

Instead of the naive direct mapping, we opted to use a log-structured approach typically used in journalling file systems and modern SSDs and SMR drives. Every write request simply appends a write record into the log. Chunks of the log are numbered sequentially and uploaded as S3 objects.

The batching of writes done by BUSE which was explained in Section 4.2 caters specifically to this use-case. Writes are already batched into chunks and stored in the shared memory, ready to be uploaded as objects without modification. In many ways, batching on the write path is the only viable approach, and we believe that other BUSE-based drivers other than BS3 will benefit from it.

8.1.2 Read Path and LBA Mapping

Our scheme needs to somehow store the information about the mapping of LBAs to ranges in S3 objects since it cannot be computed statically. This *map* is consulted for every read to find out the objects (and ranges within the objects) which need to be downloaded and is updated with every write. The objects are then downloaded in parallel to satisfy read requests.

The map can be any dictionary-like data structure. The space/time tradeoff of the data structure is of utmost importance, as high-performance data structures will tend to consume more memory, while compact data structures will often provide somewhat lower performance. The choice of data structure will typically depend on use case and system parameters.

8.2 Implementation Details

BS3 is implemented as Go program where asynchronous operations are implemented with goroutines and channels. With these two language primitives, the massive parallelism exposed in I/O-intense application was easy to manage and, most importantly, allowed us to implement a correct solution in a reasonable amount of code. Figure 8.1 shows the architecture of BS3, which is further described in following sections.

8.2.1 Write Path

When a write request is read from the character device, corresponding write chunk with batched writes is read from shared memory. BS3 assigns a sequential number to the chunk and uses it as an object key. In this moment the chunk (object) is marked as ready to be sent to S3. As soon as the object is correctly uploaded to S3 storage, the map is updated and the whole write chunk is acknowledged back to BUSE in-kernel driver. From now on, all succeeding reads are able to read the newly written data.

S3 Object Structure

The S3 object, in fact a BUSE write chunk with a sequential number, is comprised of two parts.

The first part, *header*, contains metadata about the writes the object contains. The second *data* part contains the actual data that were written to the block device.

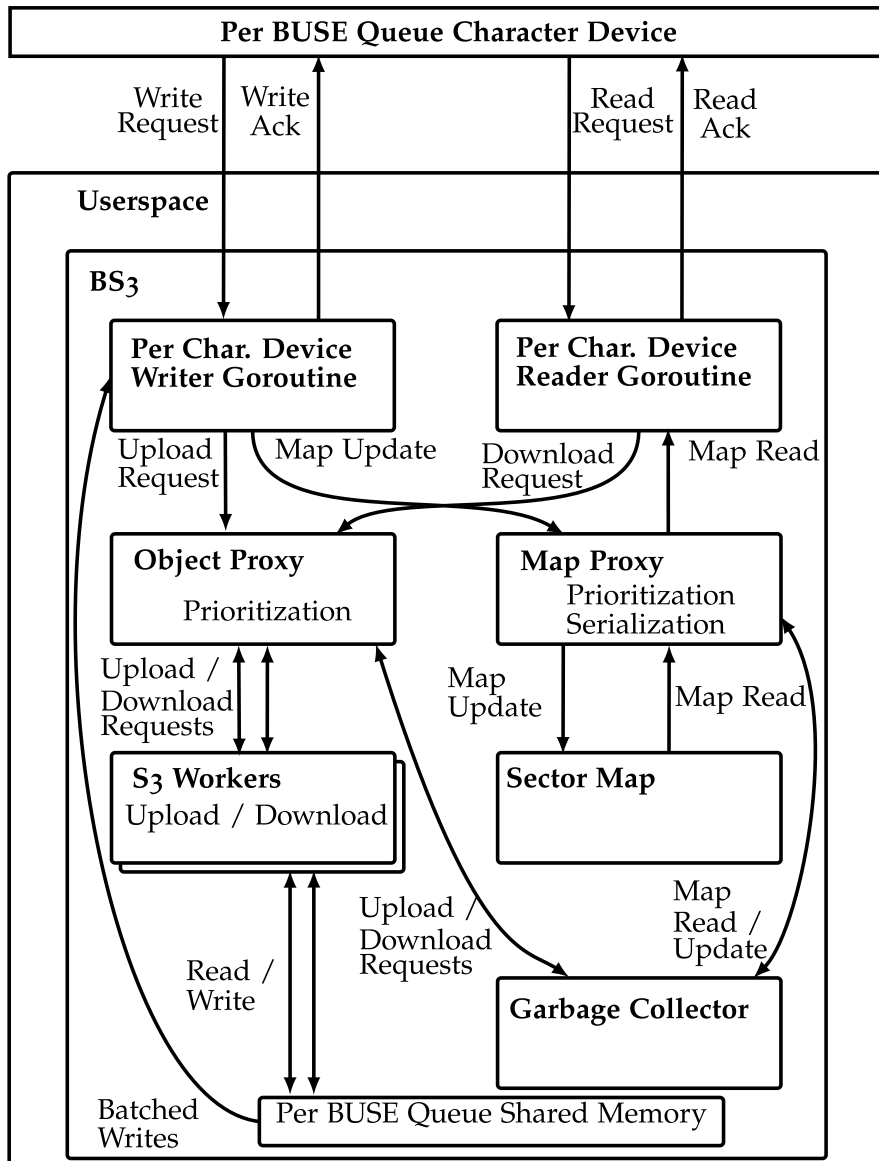

Figure 8.1: BS3 design.

The example of header with 3 writes is shown in Table 8.2 and corresponding data part in Table 8.3. Table 8.1 shows the structure of a write record in the BUSE write chunk header, which is equivalent to the description of a write chunk from Section 8.1.1.

The split between a header part and a data part is done statically by BUSE and depends on the sector size and maximum configured size of the write chunk. In the example below the split happens at the 1 kB mark.

Table 8.1: Structure of a write record in the object header.

Byte Position	Short Name	Description
0–7	Sector	First sector to be written.
8–15	Length	Number of sectors to be written.
16–23	Seq	Sequential number of the write I/O request.
24–31	RFU	Reserved for future use.

Table 8.2: Example of S3 Object Header for 3 writes. First write is to sector 1, has length 5 and sequential number 10. One write record has size of 32 B.

Byte position	Sector	Length	Seq	RFU
0–31	1	5	10	-
32–63	2	8	9	-
64–95	3	7	20	-

Table 8.3: Example of S3 Object Data part for writes from Table 8.2 with header-data boundary at 1 kB. Sector size is 4 kB.

Byte position	Description
1 k–20 k	Write 1 Data
21 k–52 k	Write 2 Data
53 k–80 k	Write 3 Data

8.2.2 Map Implementation

BS3 does not depend on one particular map implementation. As noted above, different data structures may play the role of a map, and the choice depends on use-case. Instead, BS3 uses a *map proxy* which hides concrete map implementation behind unified API.

Furthermore, the proxy performs serialization of the map requests, since it is

expected that multiple threads will modify the map simultaneously. It can also prioritize requests, for example to defer updates from non-critical processes such as background garbage collection in favor of regular read and write requests.

Map proxy is internally implemented as a goroutine and all operations translate into communication through Go channels. This solution was selected instead of traditional locking for its higher cache efficiency: the map is always scanned by the same thread, not by multiple distinct threads after obtaining the lock.

Sector Map

Currently the only implementation of the map is a high-performance *sector map*. It is a simple open-addressed array, where LBA serves as the index. It contains write metadata mentioned in Table 8.1 (Sector, Length, Seq and RFU).

Since we often need to scan the array sequentially and since sequential scan of an array is an extremely efficient operation, this map type is highly efficient. The disadvantage of the simple approach is the space complexity, where for 1 TB volume the map consumes $1\,\text{TB}/4\,\text{kB} \cdot 32\,\text{B} = 8\,\text{GB}$ of memory, assuming a 32 B write record. However, 32 B for a write record is an upper bound, a more realistic write record size is 12 B as described in Table 8.4. Then, a 1 TB block device would consume $1\,\text{TB}/4\,\text{kB} \cdot 12 = 3\,\text{GB}$ of memory for the map, which was deemed acceptable.

Table 8.4: Proposed write record for more space efficient solution. All computations are done with a sector size of 4 kB.

Byte Position	Name	Implication
0–4	Sector	Max device size of 16 TB.
5–6	Length	Max extent size of 256 MB.
7–11	Seq	2ˆ40 writes per collision area without remount.

Update Operation. Every new write request updates the map simply by iterating over all the written sectors and replacing values with the new write record data. During this operations the write sequence number is checked to fence against obsolete writes, resolving write-after-write conflicts efficiently. (The write-after-write problem was described in Section 4.3.2.)

Lookup Operation. A single read request will typically need to download multiple ranges of multiple S3 objects. The lookup operation iterates over all sectors requested in a read request and composes a list of ranges that need to be fetched from S3.

Suggested alternatives

In our future research, we would like to focus on exploring page-table-like data structures to strike a better trade-off between performance and memory requirements. Even though the sector map fares quite well, it becomes impractical for enormous block devices which are only ever partially written.

8.2.3 Object Operations Implementation

Similarly to the sector map implementation, an *object proxy* is used to hide concrete implementation of the object operations and performs prioritization on incoming requests. With this architecture, the backend can be easily swapped out for a completely different one as long as it provides means to upload, download and delete objects.

AWS Go SDK. In the current implementation, BS3 uses the S3 library from AWS Go SDK, making it possible to use HTTP2, encryption and advanced S3 features like multi-part uploads with no additional code.

Object key transformation. Amazon S3 limits the number of operations performed on objects with a common key prefix. To overcome this rate limiting feature we perform a very simple transformation on the object key before the object is uploaded. We split the key into two parts and use the high part as a key and the low part as a prefix. This operation is easily invertible to obtain the original object number from a key and a prefix.

8.2.4 Garbage Collection

Because of log-structures writes, data in S3 are never overwritten. New writes eventually make old writes obsolete, however objects containing obsolete writes are still present in S3. Objects can have only a few obsolete writes mixed with still relevant data, or they can be made up entirely of obsolete writes. Objects with only obsolete writes are called *dead objects*.

Dead Object Garbage Collection

Dead objects can be deleted from S3 in a process called *dead-object garbage collection*. Since dead objects don't contain any live data at all, they can be deleted and all data currently written to the block device will still be accessible.

The implementation detail related to consistency is that if we actually delete the object, we cannot distinguish between a garbage-collected and never uploaded object—all we see is a gap in the object sequence. To solve this issue, instead of deleting the object, we upload an empty object with the same key. This minor detail will be explained more in Section 8.2.5 and Section 8.2.6.

Garbage collection of dead objects is extremely efficient as we keep track of object utilization online. It is performed in user-configured intervals and has

no performance impact, since all S3 and map operations are performed with a lower priority than regular block-device I/O. We can prioritize regular traffic as S3 is practically infinite and we cannot run out of space.

Threshold Garbage Collection

Non-dead objects have a certain *utilisation*, i.e. the ratio between the size of non-obsolete writes in the object and the total size of the object. The utilisation of an object is 0 if and only if the object is dead, and in the interval $(0, 1]$ otherwise.

The user can configure an object utilisation threshold and trigger the *threshold garbage collection* by sending `SIGUSR1` to the BS3 daemon. Then, all objects with utilisation ratio lower than selected threshold will be garbage collected: all non-dead writes from all objects with ratio below the threshold will be extracted into new objects and reuploaded. Once new objects are uploaded and the map is updated, the original objects become dead objects and are garbage-collected in the next cycle of dead garbage collection.

Defragmentation. The threshold garbage collection inherently performs defragmentation, since the extracted writes are selected in LBA order. This means that objects uploaded during threshold garbage collection contain data in LBA order. Utilisation threshold of 1 effectively triggers defragmentation of the entire block device.

8.2.5 Map Recovery

To attach an already created existing device, the map has to be reconstructed to the state when the block device was last disconnected.

Objects Roll-forward. All the write records describing every write request are stored in object headers. To recover the map state, it is sufficient to sequentially read headers of all the objects in the order of growing object ID and replay write requests against the map.

Once all objects are processed the map is up-to-date again. The object roll-forward is stopped as soon as we hit a gap in the sequence of object IDs. It means that some object was not uploaded successfully and the writes it was supposed to contain would be missing, compromising consistency. When this happens, all objects after the first gap are deleted since they cannot contribute to the prefix-consistent state.

Map serialization. The actual map state can also be serialized and uploaded as a separate object. This is beneficial for block devices backed by a huge amount of objects, where roll-forward map recovery would take a long time. Our implementation serializes the map during block device shutdown and uploads it as an S3 object with a special key. When the block device is about

to be started, BS3 first checks whether a serialized map exists and potentially downloads the object and recovers the map to that state.

When the block device is disconnected non-gracefully, or when upload fails, the serialized map is not uploaded. In that case the map from last graceful shutdown is used and combined with roll-forward from objects that are newer than the serialized map.

8.2.6 Prefix Consistency

BS3 is a prefix-consistent block device. It means that it always recovers into a consistent state which is a prefix of the actual state. (Chidambaram et al. [64])

This property is made possible by the fact that every object's key is a sequence number and all writes from object with lower number were written before writes from objects with higher sequential number. During the roll-forward recovery, objects are read in the order of growing sequence number (key). Any gaps in the sequence indicate missing objects, and objects following the first gap cannot contribute to the prefix-consistent state.

8.3 Null Mode

BS3 can be started in a *null mode*. Then it behaves like a null device for BUSE. It immediately acknowledges all read and write requests without any map or object operations. This mode is useful for performing experiments with BUSE and can be enabled in the configuration file with the `null` option or with the environment variable `BS3_NULL=true`.

Chapter 9

BS3 | Evaluation

We evaluate BS3 performance on multi-core EC2 instances running in AWS and compare it with CloudBD. Both technologies use Amazon S3 as a backend. We again used FIO as a microbenchmark for the raw block device and also Filebench (Tarasov, Zadok, and Shepler [65]) as a macrobenchmark of the block device with a file system stacked on top of it.

Details about performance evaluation in the cloud were presented in Section 5.1.

9.1 Experimental Setup

The experimental setup is very similar to the BUSE experimental setup from Section 5.2. The parameters of BS3 were the same as in the BUSE experiments and CloudBD was set to the high performance mode, i.e. with highest memory usage and read-ahead disabled. As a backend, Amazon S3 within the same region was used.

9.2 Microbenchmarks

For microbenchmarking, FIO was used almost in the same way as it was for the BUSE evaluation (Section 5.3). We added random reads and random writes benchmarks, because the block devices are not "null" block devices anymore and different I/O patterns matter.

9.2.1 I/O Parameters Sensitivity

m5zn.12xlarge was used in BS3 and CloudBD experiments as opposed to m5zn.3xlarge which was used for the BUSE experiments. The reason was limited network bandwidth at 25 Gb/s, which was extremely low for BS3. m5zn.12xlarge has network bandwidth limited at 100 Gb/s, has 48 vCPUs and 192 GiB of memory. We kept the number of I/O jobs the same as for

the smaller instance type to keep the BUSE results comparable. Scalability with number of I/O jobs is evaluated separately in Section 9.2.2.

Observations — Write Path

Figure 9.1 shows write throughput dependency on the block size with different I/O parameters. Every row represents a different I/O depth and every column a different number of I/O jobs. Random and sequential writes are shown to study differences between BS3 and CloudBD.

BS3 performs the same for sequential and random I/O pattern. CloudBD does not. BS3 is log-structured block device and it is expected that random and sequential writes will perform similarly. On the other hand, CloudBD uses direct mapping of LBA ranges to S3 objects and due to write amplification, the throughput of random writes is significantly lower than the throughput of sequential writes.

BS3 is block size oblivious. CloudBD is not. BS3 performs the same for any block size. This is caused by batching writes into larger chunks in the kernel. CloudBD does not batch writes in the kernel, hence the overhead of kernel-userspace transition is higher for CloudBD. CloudBD also needs to copy data one more time than BS3 as it relies in NBD.

BS3 scales with number of I/O jobs. CloudBD does not. BS3 is able to use all queues provided by the Linux kernel storage stack, therefore it scales well with number of I/O jobs. CloudBD uses only a single queue.

BS3 is (several) order(s) of magnitude faster for (random) writes with higher parallelism. BS3 has 12x and multiple thousands time higher throughput for sequential and random writes respectively with 12 I/O jobs and block sizes less than 1 MB.

Observations — Read Path

Figure 9.2 is the analogous figure for reads. Only random read throughput is shown since the sequential read throughput is the same due to disabled read-ahead capability of CloudBD.

High latency with low parallelism degrades throughput. The latency on the read path is significantly larger than the latency on the write path. It is an issue for both BS3 and CloudBD. However, BS3 is able to leverage high parallelism (large I/O depth and/or number of I/O jobs) and generate throughput around 6 GB/s with 1 MB blocks, I/O depth of 32 and 12 I/O jobs. CloudBD is limited by using only a single hardware queue and seems to be capped at around 2 GB/s.

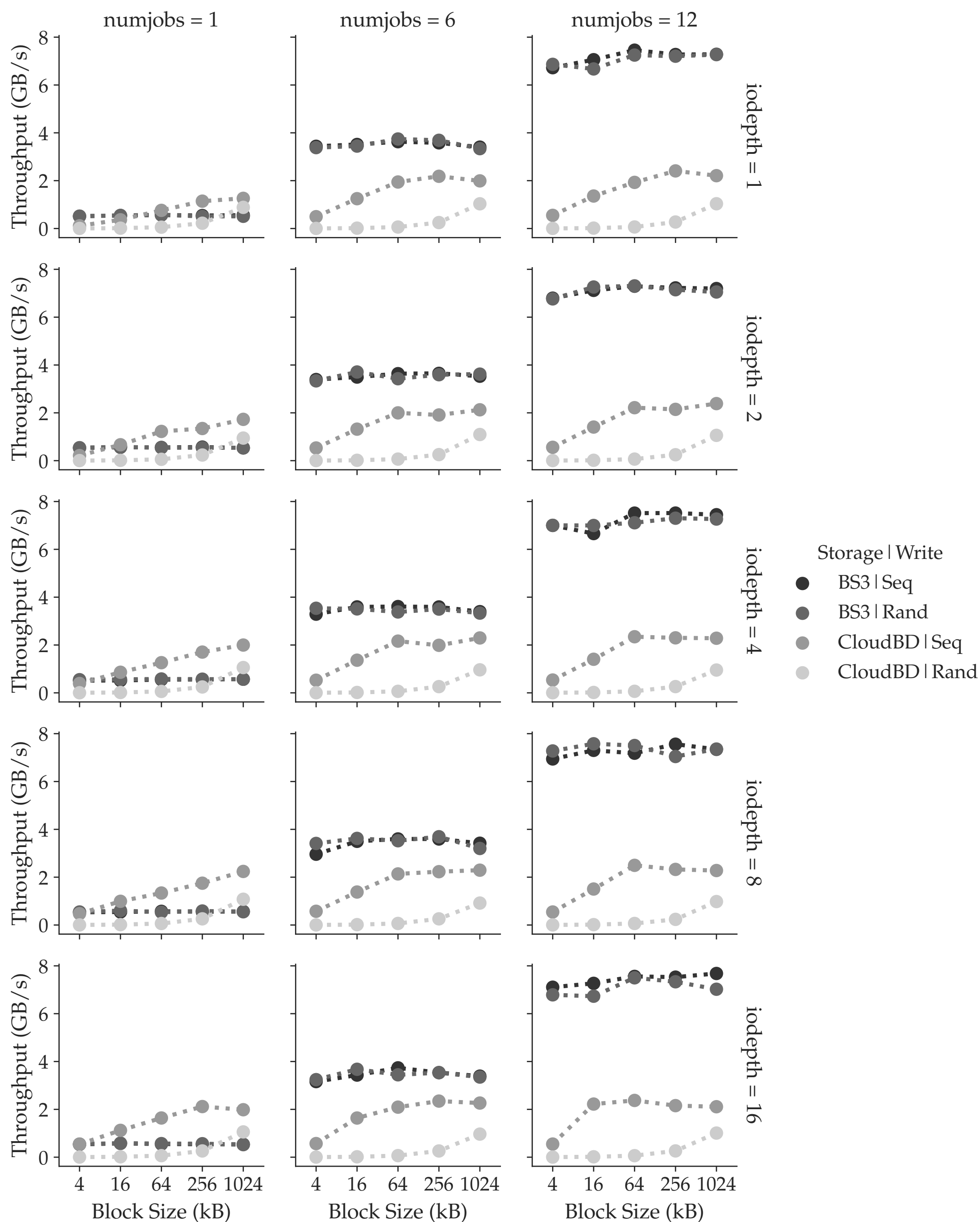

Figure 9.1: Write throughput dependency on block size with different I/O parameters on an m5zn.12xlarge instance (48 vCPUs). Smaller instances had too strict network bandwidth limits. CloudBD does not scale with number of I/O jobs but scales with block size. It is also negatively affected by the random I/O pattern. BS3 scales well with number of I/O jobs and its performance does not depend on I/O pattern nor block size. It is up to 8x faster than CloudBD for 4 kB writes and up to 3x faster for 1 MB writes. The lead of BS3 is even more apparent for higher number of I/O jobs.

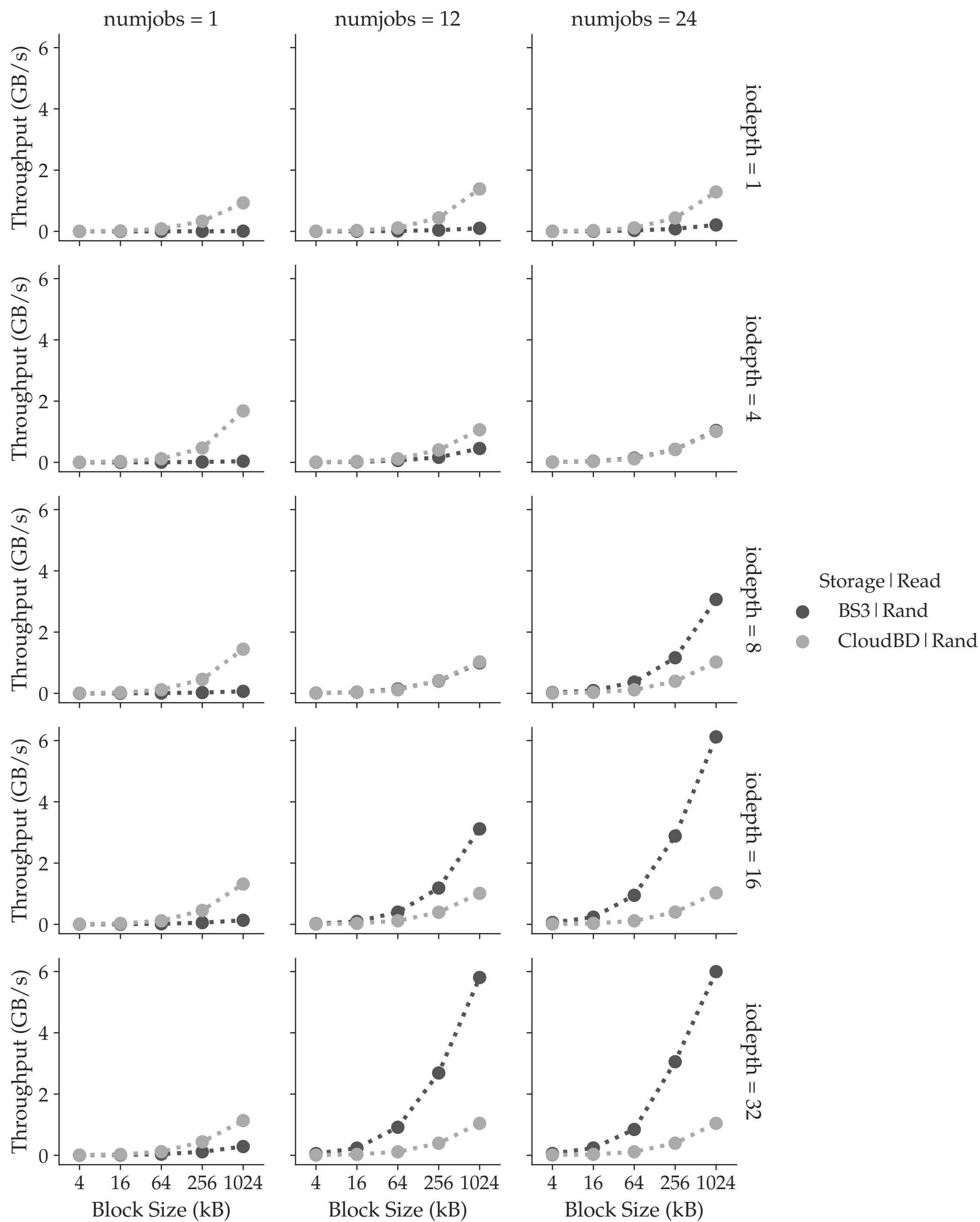

Figure 9.2: Read throughput dependency on block size with different I/O parameters on an m5zn.12xlarge instance (48 vCPUs). Smaller instances had too strict network bandwidth limits. BS3 benefits from high parallelism and for 12 I/O jobs and I/O depth 32, it reaches 6 GB/s of throughput for 1 MB block size. BS3 has 6x higher throughput than CloudBD in this case. With smaller block sizes and low parallelism, the network latency dominates and BS3 is on par with CloudBD. CloudBD scales better with block size when parallelism is very low, where it is up to 1x – 2x faster for large block sizes (256 kB and 1 MB).

9.2.2 Scaling with number of I/O Jobs

Figures 9.3 and 9.4 show how BS3 and CloudBD scales with number of I/O jobs for reads and writes respectively. Tables 9.1 and 9.2 show numeric values for corresponding figures. For writes, the I/O parameters are the same as for the BUSE scaling experiment, i.e. block size 64 kB and I/O depth 4. For reads, the block size is raised to 256 kB and I/O depth to 16, because with lower values both technologies are limited by high latency on the read path. The read variant shows only random reads because sequential reads were very similar.

BS3 scales with number of I/O jobs. CloudBD does not. For both reads and writes, BS3 scales with the number of I/O jobs, however CloudBD has slightly more optimized read path, which results in better throughput in the synchronous case (no parallelism) and the case with limited parallelism. Write throughput of BS3 is 5x and 130x higher for sequential and random I/O pattern respectively when 24 I/O jobs are used. BS3 has 6x higher read throughput for workload with 24 I/O jobs.

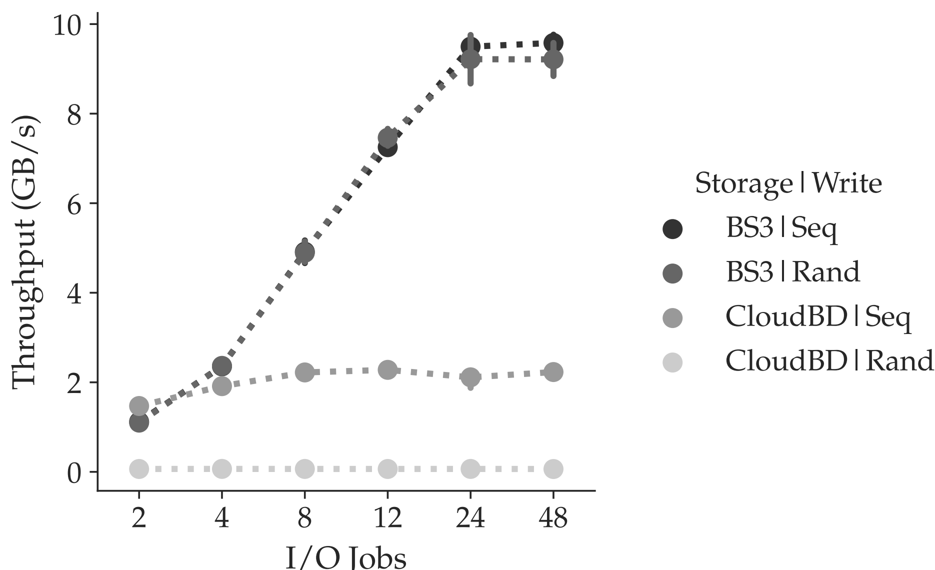

Figure 9.3: Write throughput dependency on number of I/O jobs for block size 64 kB and an I/O depth of 4 on m5zn.12xlarge instance (48 vCPUs). CloudBD is negatively affected by the random I/O pattern and does not scale with the number of I/O jobs. BS3 performs the same regardless the I/O pattern and scales until ~10 GB/s where BS3 is 5x and 130x faster for sequential and random write respectively.

Table 9.1: Write throughput for different number of I/O jobs, block size of 64 kB and an I/O depth of 4 on an m5zn.12xlarge instance. See corresponding Figure 9.3 for detailed description.

I/O Jobs	BS3 | Seq (GB/s)	BS3 | Rand (GB/s)	CloudBD | Seq (GB/s)	CloudBD | Rand (GB/s)
2	1.12 (-0.07, +0.07)	1.10 (-0.04, +0.04)	1.47 (-0.05, +0.05)	0.06 (-0.00, +0.00)
4	2.36 (-0.15, +0.11)	2.37 (-0.10, +0.09)	1.91 (-0.05, +0.06)	0.07 (-0.00, +0.00)
8	4.92 (-0.25, +0.25)	4.89 (-0.21, +0.25)	2.22 (-0.07, +0.06)	0.07 (-0.00, +0.00)
12	7.25 (-0.13, +0.13)	7.46 (-0.18, +0.20)	2.28 (-0.03, +0.03)	0.06 (-0.00, +0.00)
24	9.50 (-0.17, +0.17)	9.22 (-0.54, +0.54)	2.11 (-0.23, +0.15)	0.07 (-0.00, +0.00)
48	9.58 (-0.16, +0.18)	9.21 (-0.37, +0.37)	2.23 (-0.05, +0.06)	0.06 (-0.00, +0.00)

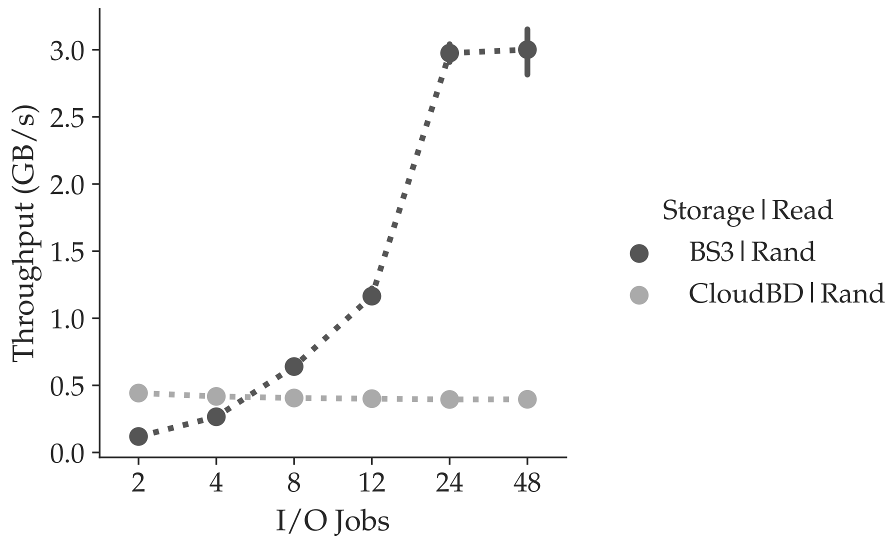

Figure 9.4: Read throughput dependency on number of I/O jobs for block size 256 kB and I/O depth 16 on an m5zn.12xlarge instance (48 vCPUs). CloudBD does not scale with the number of I/O jobs, however it performs slightly better for number of I/O jobs lower than 4. BS3 scales with the number of I/O jobs and with 24 I/O jobs has read throughput of more than 3 GB/s (6x times higher than CloudBD). Read performance of both solutions is highly affected by the read path latency.

Table 9.2: Read throughput for different number of I/O jobs, block size of 256 kB and an I/O depth of 16 on an m5zn.12xlarge instance. See corresponding Figure 9.4 for detailed description.

I/O Jobs	BS3 | Rand (GB/s)	CloudBD | Rand (GB/s)
2	0.12 (-0.00, +0.00)	0.44 (-0.00, +0.00)
4	0.27 (-0.01, +0.01)	0.42 (-0.01, +0.01)
8	0.64 (-0.02, +0.03)	0.41 (-0.01, +0.01)
12	1.16 (-0.05, +0.05)	0.40 (-0.01, +0.01)
24	2.98 (-0.08, +0.08)	0.39 (-0.00, +0.00)
48	3.00 (-0.19, +0.15)	0.40 (-0.00, +0.00)

9.3 Macrobenchmarks

We evaluated performance in real-world workloads with Filebench, widely used file system-level benchmark in the storage literature, and deployed the ext4 filesystem with default settings. In the following sections, we describe the selected Filebench workloads and present respective results in the form of multiple bar plots. Note that we needed to adjust workloads parameters because default values were designed for obsolete systems. Also keep in mind that the presented throughput is an application-level throughput reported by Filebench.

9.3.1 Fileserver

The *Fileserver* workload simulates a network fileserver by issuing operations like create, delete, append, stat, open, close, read and write. It does not sync data during the workload hence it has very few barriers and large amount of outstanding writes. This high parallelism is beneficial for BS3 as can be seen on Figure 9.5 and in Table 9.3, where BS3 has 4x higher (8x in weak flush mode) throughput than CloudBD.

Adjusted parameters

- `$nfiles=960000`
- `$nthreads=96`
- `$iosize=10m`

9.3.2 Varmail

The *Varmail* workload simulates a mail server storing e-mails in separate files, e.g. like Postfix (Venema [66]) does by default. It generates an access pattern very similar to Filebench, however with smaller files and, most importantly, introduces a sync operation after every fileappend operation. *sync* operations introduce ordering and reduce parallelism on the write path, making this workload bounded by the latency of the sync operation. Figure 9.6 and Table 9.4 show similar performance of BS3 and CloudBD caused by the aforementioned latency. However, if BS3 in weak flush mode is used, the latency on the write path is eliminated and the performance is 30x higher than both CloudBD and BS3 in strict flush mode.

Adjusted parameters

- `$nfiles=100000`
- `$meandirwidth=100000000`
- `$filesize=cvar(type=cvar-gamma,parameters=mean:32768;gamma:1.5)`

- `$nthreads=256`
- `$iosize=2m`
- `$meanappendsize=32k`

9.3.3 OLTP

OLTP is very similar workload to Varmail. It also presents sync-heavy workload, however with higher number of smaller writes between sync operations. Similarly to Varmail, Figure 9.7 and Table 9.5 show similar performance for CloudBD and BS3 in strict flush mode and 9x higher throughput if BS3 in weak flush mode is used.

Adjusted parameters

- `$nfiles=500`
- `$nshadows=500`
- `$ndbwriters=30`
- `$filesize=100m`
- `$logfilesize=100m`

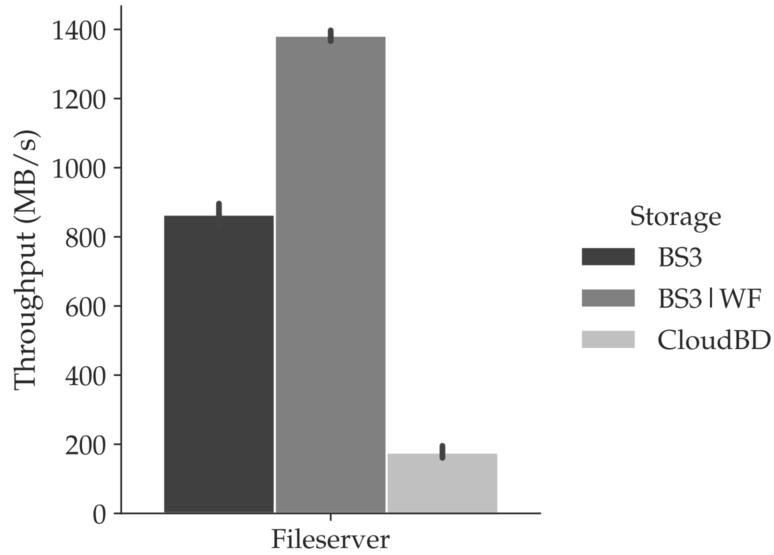

Figure 9.5: Throughput of Fileserver workload for BS3, BS3 in weak flush mode and CloudBD. BS3 benefits from high parallelism in the workload and has 5x and 8x higher throughput than CloudBD.

Table 9.3: Fileserver workload throughput for BS3, BS3 in weak flush mode and CloudBD. See corresponding Figure 9.5 for detailed description.

Storage	Throughput (MB/s)	Conf. Interval (MB/s)
BS3	864.20	(-35.37, +32.20)
BS3 | WF	1381.86	(-15.79, +15.49)
CloudBD	175.91	(-15.33, +19.73)

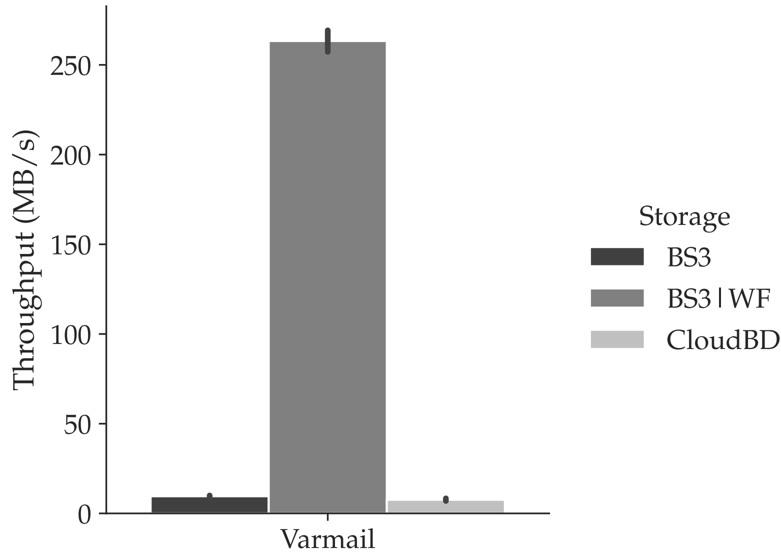

Figure 9.6: Throughput of Varmail workload for BS3, BS3 in weak flush mode and CloudBD. Large number of sync operations degrades throughput for BS3 in strict flush mode and CloudBD because they significantly increase I/O command latency. If BS3 in weak flush mode is used, the I/O latency stays low and throughput is at least 30x higher.

Table 9.4: Varmail workload throughput for BS3, BS3 in weak flush mode and CloudBD. See corresponding Figure 9.6 for detailed description.

Storage	Throughput (MB/s)	Conf. Interval (MB/s)
BS3	9.57	(-0.33, +0.40)
BS3 | WF	263.31	(-6.06, +5.94)
CloudBD	7.61	(-0.67, +0.77)

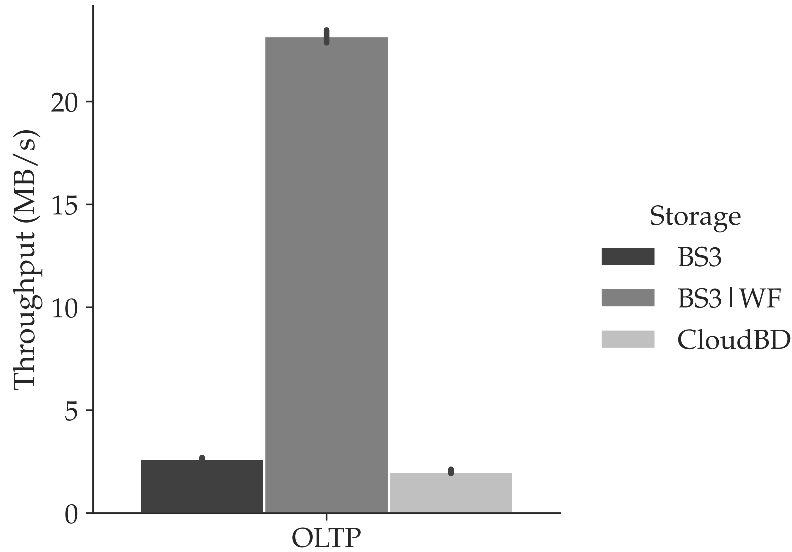

Figure 9.7: Throughput of OLTP workload for BS3, BS3 in weak flush mode and CloudBD. Large number of sync operations degrades throughput for BS3 in strict flush mode and CloudBD because they significantly increase I/O commands latency. If BS3 in weak flush mode is used, the I/O latency stays low and throughput is 9x higher.

Table 9.5: OLTP workload throughput for BS3, BS3 in weak flush mode and CloudBD. See corresponding Figure 9.7 for detailed description.

Storage	Throughput (MB/s)	Conf. Interval (MB/s)
BS3	2.63	(-0.06, +0.07)
BS3 | WF	23.17	(-0.31, +0.30)
CloudBD	2.01	(-0.09, +0.11)

Chapter 10

Conclusion

Nowadays, object storage systems of large cloud providers offer performance, reliability and scalability guarantees rarely provided by other storage technologies. In this thesis, we leverage the superior attributes of object storage systems and implement a block device on top of them with superior parameters. However, the efficient implementation of a block device backed by Amazon S3 first required the invention of a solution for efficient implementation of block device drivers in userspace.

Therefore, in the first part of the thesis, we presented BUSE, a mechanism for creating a block device in userspace. Compared to other solutions, BUSE has following advantages:

- Per-core scaling. Experiments show ~120 GB/s write throughput and ~72 GB/s read throughput for 48 vCPUs. This result is at least 7x better than any other existing technologies on the same hardware. Being able to provide extremely high throughput means that, for realistic workloads, the CPU utilization will stay very low. For that reason, per-core scalability is fundamental for modern many-core low-power architectures.
- Batching of writes, improving single queue performance.
- Introduces weak flush mode, in which the durable semantics of the flush command is omitted. This significantly lowers latency.
- Solves consistency issues, such as write-after-write and read-after-write, which stem from the use of multiple hardware queues in the Linux kernel storage stack. This allows the userspace driver to focus on its core business.
- Provides a simple interface to create a block device without extraneous memory copies.

The second part of the thesis focused on BS3. BS3 is a high-performance block device which uses an S3-compatible object storage as a backend. It provides unmatched performance and following features:

- The "no extraneous memory copies" design of BUSE directly translates into the performance of BS3.
- Efficient implementation with various performance-related design decisions, e.g. prioritization of extent map requests and S3 object operations.
- Scalability with number of cores. During the evaluation, BS3 was able to generate 10+ GB/s throughput. With these data flows, the network connectivity will be the bottleneck in most cases, since we are almost saturating a 100 Gbit/s network.
- Write throughput higher than the most recent PCIe 3.0 4x NVMe SSDs, which have a theoretical limit of 4 GB/s.
- Durability of 99.999999999 % (eleven nines) due to the use of Amazon S3. This makes BS3 ideal for high-performance backup solutions.
- The write throughput is up to 130x higher than CloudBD, a competing commercial solution.
- The block device is consistent under all circumstances due to the combination of log-structured writes and the atomicity of Amazon S3.

These qualities make BS3 a very promising candidate for:

- A cheaper yet powerful alternative to existing block storage solution of cloud providers, such as Amazon's EBS.
- On-premise block-device connected to scalable object storage architecture such as MinIO.
- Block-device for disk-less nodes in a private cloud.
- High-performance backup solutions.

10.1 Future Work

With BUSE, we can implement very unconventional block devices with superior performance and reliability characteristics, as exemplified by BS3. We would like to publish our work on a storage-related conference and initiate the process of inclusion of BUSE into the Linux mainline kernel.

Many improvements of BS3 can be implemented:

- **Snapshots:** Log-structured writes make them relatively easy and straightforward to provide. The technical challenge will be to adjust the garbage collection logic to work with object reference counters.
- **Clones:** One block device could provide a base image for several other block devices, saving the space and improving performance. Technical challenges are the same as in the case of snapshots.

- **Local NVMe-based caching:** As we saw in the evaluation, read performance is relatively low compared to write performance. This is due to inherent high latency of reads over a network. With smart local caching, this latency could be minimized, significantly improving throughput of real-world workloads.

Bibliography

[1] Alessandro Rubini and Jonathan Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001 (cited on page 7).

[2] LWN. *The end of block barriers*. URL: https://lwn.net/Articles/400541/ (visited on 07/22/2021) (cited on page 8).

[3] Linux. *Multi-Queue Block IO Queueing Mechanism (blk-mq)*. URL: https://www.kernel.org/doc/html/v5.10/block/blk-mq.html (visited on 06/17/2021) (cited on page 9).

[4] LWN. *The multiqueue block layer*. URL: https://lwn.net/Articles/552904 (visited on 06/17/2021) (cited on page 9).

[5] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. "Linux block IO: introducing multi-queue SSD access on multi-core systems." In: *Proceedings of the 6th international systems and storage conference*. 2013, pages 1–10 (cited on pages 9–11).

[6] NBD. *Network Block Device*. 2021. URL: https://nbd.sourceforge.io (visited on 04/18/2021) (cited on page 14).

[7] Linux Journal. *The Network Block Device*. 2021. URL: https://www.linuxjournal.com/article/3778 (visited on 04/18/2021) (cited on page 14).

[8] nbdkit. *NBD server with stable plugin ABI and permissive license*. 2021. URL: https://gitlab.com/nbdkit/nbdkit (visited on 04/18/2021) (cited on page 14).

[9] CloudBD. *CloudBD*. URL: https://www.cloudbd.io (visited on 04/17/2021) (cited on page 14).

[10] Datera. *Linux-IO*. URL: http://linux-iscsi.org (visited on 04/18/2021) (cited on page 15).

[11] Seagate. *SCSI Commands Reference Manual*. Version 100293068, Rev. J. Oct. 2016. URL: https://www.seagate.com/files/staticfiles/support/docs/manual/Interface%5C%20manuals/100293068j.pdf (cited on page 15).

[12] Linux Kernel. *TCMU Design*. URL: https://www.kernel.org/doc/Documentation/target/tcmu-design.txt (visited on 04/18/2021) (cited on page 15).

[13] Linux Kernel. *The Userspace I/O HOWTO*. URL: `https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html` (visited on 04/18/2021) (cited on page 15).

[14] LWN. *UIO: user-space drivers*. URL: `https://lwn.net/Articles/232575` (visited on 04/18/2021) (cited on page 15).

[15] Open-iSCSI. *tcmu-runner*. 2021. URL: `https://github.com/open-iscsi/tcmu-runner` (visited on 04/18/2021) (cited on page 15).

[16] CoreOS. *go-tcmu*. 2021. URL: `https://github.com/coreos/go-tcmu` (visited on 04/18/2021) (cited on page 15).

[17] M. Chadalapaka, J. Satran, K. Meth, and D. Black. *Internet Small Computer System Interface (iSCSI) Protocol*. RFC 7143. RFC Editor, Apr. 2014. URL: `https://www.rfc-editor.org/rfc/rfc7143.txt` (cited on page 15).

[18] R. Weber, M. Rajagopal, F. Travostino, M. O'Donnell, C. Monia, and M. Merhar. *Fibre Channel (FC) Frame Encapsulation*. RFC 3643. RFC Editor, Dec. 2003. URL: `https://www.rfc-editor.org/rfc/rfc3643.txt` (cited on page 15).

[19] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. "Spdk: A development kit to build high performance storage applications." In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2017, pages 154–161 (cited on page 15).

[20] libfuse. *The reference implementation of the Linux FUSE*. URL: `https://github.com/libfuse/libfuse` (visited on 04/18/2021) (cited on page 15).

[21] Miklos Szeredi. *FUSE: Filesystem in userspace*. 2010 (cited on page 15).

[22] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To {FUSE} or not to {FUSE}: Performance of user-space file systems." In: *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*. 2017, pages 59–72 (cited on page 15).

[23] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. "Terra incognita: On the practicality of user-space file systems." In: *7th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. 2015 (cited on page 15).

[24] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. *Glusterfs one storage server to rule them all*. Technical report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2012 (cited on page 15).

[25] NTFS-3G. *NTFS-3G Safe Read/Write NTFS Driver*. URL: `https://sf.net/projects/ntfs-3g` (visited on 04/18/2021) (cited on page 15).

[26] sshfs. *A network filesystem client to connect to SSH servers*. URL: https://github.com/libfuse/sshfs (visited on 04/18/2021) (cited on page 15).

[27] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. "BDUS: implementing block devices in user space." In: *Proceedings of the 14th ACM International Conference on Systems and Storage*. 2021, pages 1–11 (cited on pages 16, 41).

[28] Jens Axboe. *Flexible I/O Tester*. URL: https://github.com/axboe/fio (visited on 03/07/2021) (cited on page 30).

[29] Pierre Dragicevic. "Fair statistical communication in HCI." In: *Modern statistical methods for HCI*. Springer, 2016, pages 291–330 (cited on page 30).

[30] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994 (cited on page 30).

[31] Bradley Efron. "Second thoughts on the bootstrap." In: *Statistical science* 18.2 (2003), pages 135–140 (cited on page 30).

[32] Lubomír Bulej, Vojtěch Horký, Petr Tuma, François Farquet, and Aleksandar Prokopec. "Duet benchmarking: improving measurement accuracy in the cloud." In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020, pages 100–107 (cited on page 30).

[33] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. "On the performance variability of production cloud services." In: *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2011, pages 104–113 (cited on page 31).

[34] Christoph Laaber, Joel Scheuner, and Philipp Leitner. "Software microbenchmarking in the cloud. How bad is it really?" In: *Empirical Software Engineering* 24.4 (2019), pages 2469–2508 (cited on page 31).

[35] Philipp Leitner and Jürgen Cito. "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds." In: *ACM Transactions on Internet Technology (TOIT)* 16.3 (2016), pages 1–23 (cited on page 31).

[36] Amazon. *Amazon EC2*. URL: https://aws.amazon.com/ec2 (visited on 06/17/2021) (cited on page 31).

[37] Amazon. *Amazon Nitro*. URL: https://aws.amazon.com/ec2/nitro (visited on 06/17/2021) (cited on pages 31, 47).

[38] Brendan Gregg. *AWS EC2 Virtualization 2017: Introducing Nitro*. URL: http://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html (visited on 06/17/2021) (cited on page 31).

[39] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. "hwloc: A generic framework for managing hardware affinities in HPC applications." In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE. 2010, pages 180–186 (cited on page 32).

[40] Wikipedia. *Wikipedia: Object Storage*. URL: https://en.wikipedia.org/wiki/Object_storage#Market_adoption (visited on 04/17/2021) (cited on page 44).

[41] Peter Braam. "The Lustre storage architecture." In: *arXiv preprint arXiv:1903.01955* (2019) (cited on page 44).

[42] Facebook. *Needle in a haystack: efficient storage of billions of photos*. URL: https://engineering.fb.com/2009/04/30/core-data/needle-in-a-haystack-efficient-storage-of-billions-of-photos (visited on 04/17/2021) (cited on page 44).

[43] Amazon. *Amazon Simple Storage Service (S3)*. URL: https://aws.amazon.com/s3 (visited on 04/18/2021) (cited on page 44).

[44] DigitalOcean. *DigitalOcean: Spaces Object Storage*. URL: https://www.digitalocean.com/products/spaces (visited on 04/18/2021) (cited on page 44).

[45] OpenStack. *OpenStack Object Storage ("Swift")*. URL: https://wiki.openstack.org/wiki/Swift (visited on 04/18/2021) (cited on page 44).

[46] Microsoft. *Microsoft Azure Blob Storage*. URL: https://azure.microsoft.com/en-us/services/storage/blobs (visited on 04/18/2021) (cited on page 44).

[47] Google. *Google Cloud Storage*. URL: https://cloud.google.com/storage (visited on 04/18/2021) (cited on page 44).

[48] Amazon. *Amazon Elastic Block Store (EBS)*. URL: https://aws.amazon.com/ebs (visited on 04/18/2021) (cited on page 47).

[49] Google. *Google Persistent Disk: durable block storage*. URL: https://cloud.google.com/persistent-disk (visited on 04/18/2021) (cited on page 47).

[50] Microsoft. *Microsoft Azure Disk Storage*. URL: https://azure.microsoft.com/en-us/services/storage/disks (visited on 04/18/2021) (cited on page 47).

[51] CloudBD. *CloudBD Documentation*. URL: https://www.cloudbd.io/docs (visited on 04/17/2021) (cited on page 47).

[52] Microsoft. *Microsoft Avere Systems*. URL: https://www.microsoft.com/en-us/avere (visited on 04/18/2021) (cited on page 48).

[53] ObjectiveFS. *ObjectiveFS: Scalable High Performance File Storage*. URL: https://objectivefs.com (visited on 04/18/2021) (cited on page 48).

[54] Panzura. *CloudFS - the Global Cloud File System Replacing Legacy Storage*. URL: https://panzura.com/products (visited on 04/18/2021) (cited on page 48).

[55] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. "Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications." In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pages 257–273 (cited on page 48).

[56] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. "Flat datacenter storage." In: *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, pages 1–15 (cited on page 48).

[57] Edward K Lee and Chandramohan A Thekkath. "Petal: Distributed virtual disks." In: *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. 1996, pages 84–92 (cited on page 48).

[58] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. "Robustness in the Salus scalable block store." In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pages 357–370 (cited on page 48).

[59] Apache HBase. *Apache HBase*. URL: https://hbase.apache.org (visited on 04/17/2021) (cited on page 48).

[60] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. "Ursa: Hybrid block storage for cloud-scale virtual disks." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pages 1–17 (cited on page 48).

[61] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. "BlueSky: a cloud-backed file system for the enterprise." In: *Proceedings of the 10th USENIX conference on File and Storage Technologies*. 2012, pages 19–19 (cited on page 49).

[62] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. "Ceph: A scalable, high-performance distributed file system." In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pages 307–320 (cited on page 49).

[63] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. "CRUSH: Controlled, scalable, decentralized placement of replicated data." In: *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE. 2006, pages 31–31 (cited on page 49).

[64] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Optimistic crash consistency." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pages 228–243 (cited on page 57).

[65] Vasily Tarasov, Erez Zadok, and Spencer Shepler. *Filebench: A flexible framework for file system benchmarking*. URL: https://github.com/filebench/filebench (visited on 03/07/2021) (cited on page 58).

[66] Wietse Venema. *Postfix*. URL: http://www.postfix.org/ (visited on 06/17/2021) (cited on page 65).

Appendix A

Thesis Meta-repository

Repository	Description
bs3	BS3: Block Device in S3
buse	BUSE: Block Device in Userspace
experiments	Thesis Experiments
text	Thesis Text

Appendix B

BS3: Block Device in S3

B.1 Requirements

- BUSE
- GNU Make
- Go 1.16 or newer

B.2 Installation

```
make
sudo make install
```

B.3 Usage

```
# Edit /etc/bs3/config.toml first

systemctl start bs3
systemctl status bs3
systemctl stop bs3
```

B.4 Configuration File

```
# Specify the major of the corresponding buse device you
# want to configure and connect to. E.g. 0 if you want to
# work with /dev/buse0.
major = 0

# Number of user-space daemon threads which is also a
# maximal number of queues storage stack uses. This is
# limited to the number of CPUs. I.e. minimal value is 1 and
# maximal is number of CPUs. Optimally it should be set to
# the number of CPUs. 0 means optimal value.
threads = 0
```

```
# Size of the created block device in GB.
size = 8 #GB

# Block size of created device. 512 or 4096. It is forbidden
# to change block_size on the existent block device. In B.
block_size = 4096 #B

# Whether IOs should be scheduled by linux kernel stack.
scheduler = false

# IO queue depth for created block device.
queue_depth = 256

# Use null backend, i.e. just immediately acknowledge reads
# and writes and drop them. Useful for testing raw BUSE
# performance. Otherwise useless because all data are lost.
null = false

# Enable web-based go pprof profiler for performance
# profiling.
profiler = false

# Profiler port.
profiler_port = 6060

# Configuration related to AWS S3
[s3]
# AWS Access Key
access_key = "Server-Access-Key"

# AWS Secret Key
secret_key = "Server-Secret-Key"

# Bucket where to store objects.
bucket = "bs3"

# <protocol>://<ip>:<port> of the S3 backend. AWS S3
# endpoint is used when empty string.
remote = "http://192.168.122.1:9000"

# Region to use.
region = "us-east-1"

# Max number of threads to spawn for uploads and downloads.
uploaders = 384
downloaders = 384

# Configuration specific to write path.
[write]
# Semantics of the flush request. True means durable device,
# i.e. flush request gets acknowledge when data are
# persisted on the backend. False means eventually durable,
# i.e. flush request just a barrier.
durable = false
```

```
# Size of the shared memory between kernel and user space
# for data being written. The size is per one thread. In MB.
shared_buffer_size = 32 #MB

# Size of the chunk. Chunk is the smallest piece which can
# be sent to the user space and where all writes are stored.
# In MB.
chunk_size = 4 #MB

# The whole address space is divided into collision domains.
# Every collision domain has its own counter for writes'
# sequential numbers. This is useful when we don't want to
# have one shared counter for writes. Instead we split it
# into parts and save the cache coherency protocol traffic.
# In MB.
collision_chunk_size = 1 #MB

# Configuration specific to read path.
[read]

# Size of the shared memory between kernel and user space
# for data being read. The size is per one thread. In MB.
shared_buffer_size = 32 #MB

# Garbage Collection related configuration
[gc]
# Step when scanning the extent map. In blocks.
step = 1024

# Threshold for live data in the object. Objects under this
# threshold are garbage collected by the "threshold GC"
# which is trigerred by SIGUSR1. This type of GC is heavy on
# resources and should be planned by the timer for not
# intense times.
live_data = 0.3

# Timeout to wait before any of requests from GC thread will
# be served by the extent map and object manager. In ms.
idle_timeout = 200

# How many seconds to wait before next periodic GC round.
# This is related to "dead GC" cleaning just dead objects.
# It very light on resources and does not contend for the
# extent map like the "threshold GC".
wait = 600

# Configuration specific to the logger.
[log]
# Minimal level of logged messages. Following levels are
# provided: panic 5, fatal 4, error 3, warn 2, info 1, debug
# 0, trace -1
level = -1

# Pretty print means nicer log output for human but much
```

```
# slower than non-pretty json output.
pretty = true
```

Appendix C

BUSE: Block Device in Userspace

C.1 Requirements

- GNU Make
- Linux Kernel 5.11 or newer
- Linux Kernel Headers

C.2 Installation

```
cd kernel
make
sudo make install
sudo modprobe buse
```

Appendix D

Thesis Experiments

In the spirit of reproducible research this repository contains source files for experiments rerun.

D.1 Repository Structure

Directory	Description
cpuinfo	Target machines' `/proc/cpuinfo`.
data	Experiments results and generated `csv` files.
extra	`tcmu` and `nbd` null backend implementations.
lstopo	Target machines' `lstopo`.
plots	Generated from `data/**/*.csv` by `scripts/**/plotters`.
scripts/aws	AWS EC2 scripts for running experiments.
scripts/fio	Fio command generator, plotters and data scrapper.

D.2 Requirements

- GNU Make
- Go 1.16 or newer
- nbdkit 1.24 or newer (with plugin development support)
- BDUS 0.1.1
- Python 3.7 or newer
- Boto 3
- Fio
- Flebench
- jq

D.3 FIO Experiments Usage

```
cd scripts/aws
```

```
# Generates all FIO parameters combinations and run them on
# AWS EC2 instances
../fio/generator/cmdgen.py \
    --rws read \
    --bss 4k 8k \
    --iodepths 1 8 16 \
    --runtime 60 \
    --numjobs 2 4 8 \
    --direct 1 \
    | \
    shuf \
    | \
    ./runner.py \
        --instance m5.large \
        --storage bs3-null nbd tcmu

# Generates csv file from data in RESULTS_DIR
./scrapper/csvgen.sh RESULTS_DIR > fio.csv

# Plots sensitivity graph from fio.csv.
./plotters/buse/sensitivity.py --rw write fio.csv
```

Appendix E

Thesis Text

E.1 Requirements

- GNU Make
- LaTeX (TeX Live recommended)
- Pandoc 2.14 or newer
- Biber

E.2 Usage

Run `make` for generating the thesis to `thesis.pdf`.