Projekt-Bericht

BlueStore backend for JULEA

vorgelegt von

Johannes Coym

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	M.Sc. Informatik
Matrikelnummer:	6693524
Betreuer:	Jun.-Prof. Dr. Michael Kuhn
	Kira Duwe

Hamburg, 2021-03-24

Contents

1 Introduction

The goal of this project is to create a new backend for JULEA, which is based on the BlueStore from Ceph to store the objects in a real object store. For an introduction, we first have to take a look at the common storage interfaces like Portable Operating System Interface(POSIX) and Parallel Input/Output (I/O) Libraries like Message Passing Interface-IO (MPI-IO) and Net Common Data Format (NetCDF). Afterwards, we continue with an explanation of BlueStore [Cepd] and JULEA, and with the knowledge of the normal storage interfaces before, we will then understand why we require an object store in JULEA.

Next, the second chapter will show how the implementation of the backend and its library was planned, while the third chapter will then go into the implementation itself. In the end, the last part will contain some benchmarks to see how the new backend performs in comparison to the existing backends.

1.1 Common Storage Interfaces

To start this project, we first need to take a look at how storage systems in HPC are set up to understand the motivation of the project. At first, we will, therefore, take a look at POSIX and the VFS, which are the basics for most storage systems in Unix.

The VFS, short for Virtual File System [Wikb], is the layer in Kernel space that provides the structure and interface for file systems. The file systems also operate inside the VFS, and from the VFS the data either gets transferred over Direct I/O or the page cache to the block devices. To the applications, the VFS provides the universal POSIX interface so the access to the file systems is always the same, no matter which file system is used.

1.1.1 POSIX

POSIX [Wika] is a standardized interface to manage access from applications to different file systems and also the operating system. Its development was started in 1985 to set standards independent from which OS you are using. In 1988 it was set as the standard for Unix for the first time, and afterwards, it was continuously revisioned to add more features and adapt to new technology. The newest revision is from 2017, and it provides standards for programs, services, tools (e.g. echo, du), the C language, file and network I/O and much more.

For this project, the focus only lays on the file I/O part and to take an example of how POSIX works, we will first take a look at the two different write functions. The first one is just called `write` and is used to write serially as it has a file descriptor that is located at a specific byte of the file and while writing the file, the file descriptor also gets moved forward. Due to this movement of the file descriptor, using write from different threads would result in these threads writing the data mixed up.

To avoid this with multi-threaded applications there also exists the `pwrite` function. This function writes the data to the file without moving the file descriptor, instead, the threads get one shared file descriptor when opening the file. This file descriptor is then used with an offset to set the point where the data should be written, so multiple threads can write at different places of the file.

Another interesting aspect of POSIX are the semantics which are very strict to ensure reliable and reproducible results when reading. These semantics say that a file has to be locked until it is secured that every `read` call has read every data that has been written until this point, which can be problematic for larger distributed file systems. In the following block is the POSIX 2008 specification for the `write` function which describes this semantic.

> After a `write()` to a regular file has successfully returned:
>
> – Any successful `read()` from each byte position in the file that was modified by that write shall return the data specified by the `write()` for that position until such byte positions are again modified.
>
> – Any subsequent successful `write()` to the same byte position in the file shall overwrite that file data.

Such strict semantics can often be an issue in HPC [Loc]. These semantics are often not wanted for HPC since performance is often more important than forced consistency. Usually, a program would secure its consistency in a file, so the POSIX semantics are not needed but would have an impact on the performance. Additionally, POSIX often is more difficult to handle than high-level I/O libraries, and POSIX also does not provide thread safety with parallel accesses on a file.

1.1.2 MPI-IO

MPI-IO [Gro16] is a part of MPI, which provides a low-level I/O interface similar to POSIX but optimized for parallel applications, which can also be distributed over several nodes. Another big difference compared to POSIX is also that the consistency semantics are much more relaxed than with POSIX.

MPI-IO works by using the normal MPI Communicator to collectively open and close files, but the write is then independent for each process. It also provides write functions just like the POSIX ones, the equivalent of `write` would be `MPI_File_write_shared` which uses a shared filehandle. `MPI_File_write_at` would then be the equivalent of `pwrite` and uses an offset to provide thread-safety.

Additionally, MPI-IO also provides collective I/O, which writes parallel on all processes and collects the data to write it more efficiently in larger blocks. With collective I/O each process is required to run one of the collective write functions, which are described in the following.

The first one is `MPI_File_write_all` which just provides a normal synchronous write. `MPI_File_write_at_all` is the second function which integrates a seek for thread safety, just as `MPI_File_write_at`. The last function is running the writes serially with a shared filehandle and is called `MPI_File_write_ordered`. The following figure shows how collective I/O with MPI works.

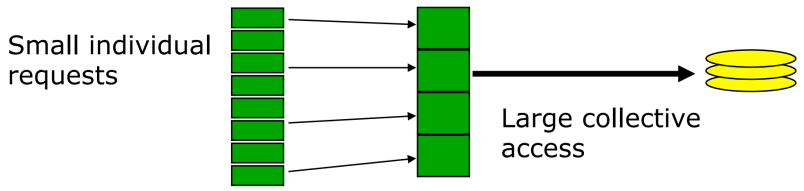

Figure 1.1: Collective I/O with MPI-IO [Gro16]

The default semantics of MPI-IO are much more relaxed than the POSIX semantics, but that also means that two writes at the same time and place in a single file do not provide a reliable or reproducible result. Additionally, writes from other processes do not require to be visible directly as the data may also still lay in page cache without being written. The only way to enforce the writes to be visible while using the default semantics is to call `MPI_FILE_SYNC` manually. If stricter semantics are required, MPI-IO also offers an atomic mode with much stricter semantics, but just like with POSIX, this comes with a bigger performance impact.

The advantages of MPI-IO, compared to POSIX, are the threadsafe data access and parallelized I/O directly integrated by MPI. Also, the more relaxed consistency semantics can be an advantage for HPC applications if strict semantics are not required. But, just as POSIX, MPI-IO also has the problem of a low-level interface with it being more difficult to handle than high-level I/O libraries like NetCDF which will be discussed in the following chapter.

1.1.3 NetCDF

NetCDF [UCAb] is a collection of libraries which provides easier access to HDF5 files using the NetCDF-4 file format. It also is accessible in different programming languages like e. g. C, C++, Fortran, and Python. NetCDF is writing serially on default, but NetCDF-4 also supports the integrated parallelization of HDF5, which is based on MPI-IO.

Initially, NetCDF started as an own file format for scientific data, which was based on the CDF format by NASA, created in 1985. These classic NetCDF formats are still supported by the current NetCDF libraries, but as the most current NetCDF format,

NetCDF-4, is based on the HDF5 format, we first need to take a look on HDF5. As these classic NetCDF variants only write serially, there also exists Parallel NetCDF (PnetCDF), which provides parallel I/O for these formats and is also based on MPI-IO.

HDF5 [Groa] is a file format specialized to large scientific data with a similar structure as a file system. In an HDF5 file, four important object types exist to describe the file and its contents. The first one is the file itself, which can only contain groups. Groups are like folders that could contain another group, but also could contain datasets. A dataset contains a homogenous array of any dimensions, which typically would contain larger data. A dataset typically also contains one or more attributes which consist of small data like metadata attached to the dataset.

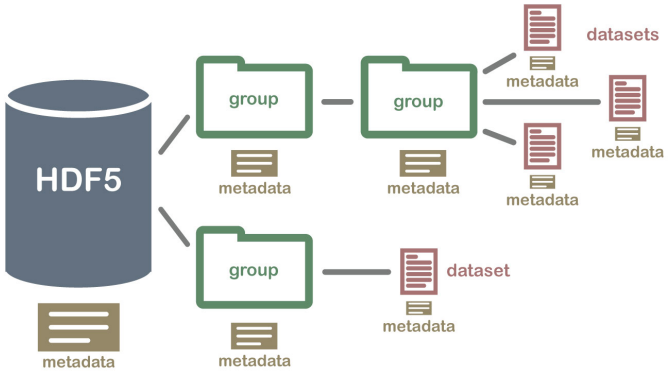

Figure 1.2: Example structure of an HDF5 file [Sci]

Based on this, NetCDF created the NetCDF-4 format, which builds HDF5 files that are fully inside of the HDF5 standards [UCAa]. So NetCDF-4 files can always be opened with HDF5, but not all HDF5 files can be opened with NetCDF because not all HDF5 features are supported. One example would be the looping groups that describe loops in an HDF5 file, which would be covered by the HDF5 standard, but NetCDF-4 only supports a tree structure without loops. An example of looping groups can be seen in figure 1.3.

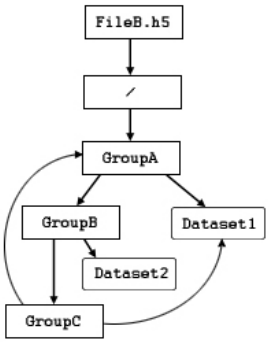

Figure 1.3: Example of a looping group in an HDF5 file [Grob]

When using NetCDF in an application, you only directly use NetCDF, but the typical I/O stack then looks like in the following figure 1.4.

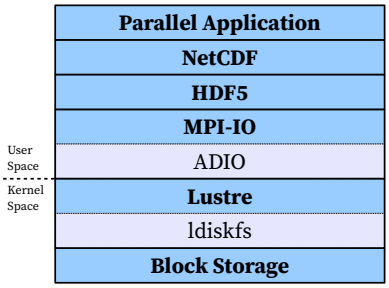

Figure 1.4: Typical I/O stack with NetCDF [Kuh17]

1.2 JULEA

JULEA [Kuh17] is a flexible storage framework for HPC which targets to simplify storage systems in HPC. A key feature of JULEA is the ability to have specific storages for data objects in multiple available object backends and metadata in several available Key-Value Stores and databases. Another key feature of JULEA, are the multiple included clients (e.g. HDF5) with the support to add even more.

For this project, we are creating a new backend for the object client, which is already designed like an object store but has no real object store as a backend. Currently, the available backends like POSIX only store the objects in a folder structure, which contradicts the simple and performant approach of a real object store. Before starting up JULEA, the config then provides the option to specify the backends that should be used, where the BlueStore shall then be selected.

Additionally, JULEA provides a much simplified I/O Stack, which is much smaller than the typical one, which was previously shown in figure 1.4, as shown in the following.

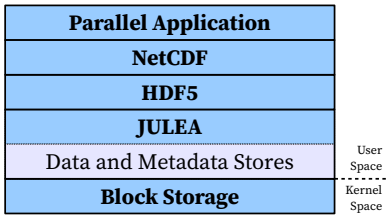

Figure 1.5: Typical I/O stack with JULEA [Kuh17]

1.3 Ceph and BlueStore

To have a look at the BlueStore, we first have to take a look at Ceph [Cepa]. Ceph is a distributed and highly scalable file system written in C++ [Cepc] and includes multiple different storage backends. Ceph also supports separate metadata servers, just like JULEA, but with Ceph these only act as a cache or to save the metadata of the file system, while JULEA e.g. supports to save attributes of HDF5 files to the metadata servers as well. Additionally, Ceph uses five different daemons for monitoring, object storage, metadata, HTTP gateways, and management, but the approach of Ceph focuses heavily on scalability, so it's built more for bigger installations.

To access the data, Ceph is completely based on RADOS and, as shown in figure 1.6, supports different ways of accessing the file system. The first is the librados, which is a library supporting different programming languages to give direct file system access to applications. The second way is the RADOS gateway which requires its own daemon and provides access via HTTP to the file system. The third way build on top of librados, just like the RADOS gateway, and provides a distributed block device with an own Linux kernel client. The last way is an own POSIX based file system with own clients for the Linux kernel as well as FUSE to use it as a normal file system.

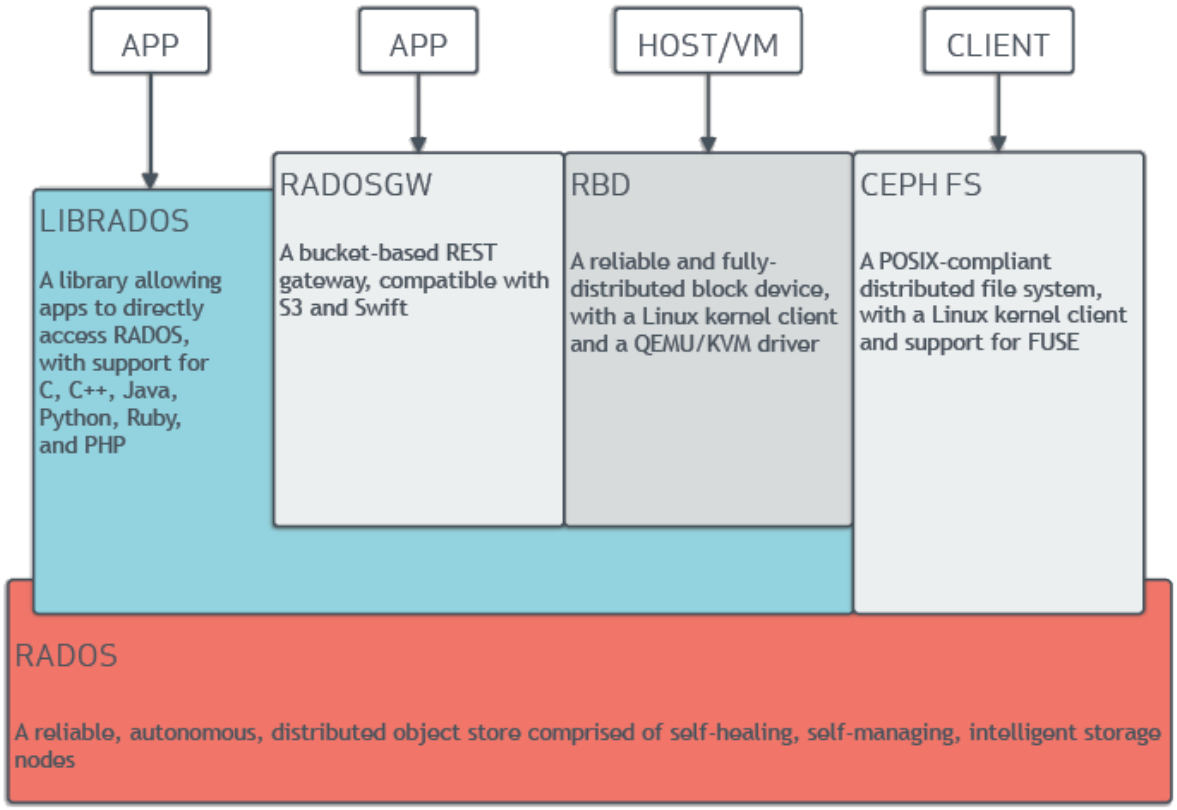

Figure 1.6: Ceph I/O Stack [Ros]

The more interesting part for this project is the object storage daemon, which includes the classic "FileStore", built like a typical file system, and the "BlueStore" which is a modern object store. As the BlueStore is an object store, it has a lot fewer features than a classic file system, as it only has to save the data blocks and where each object is saved, but e.g. it does not provide folder structures in the object store itself. Figure 1.7 shows how this different feature set translates into up to 20 percent additional performance for 4 KiB random writes.

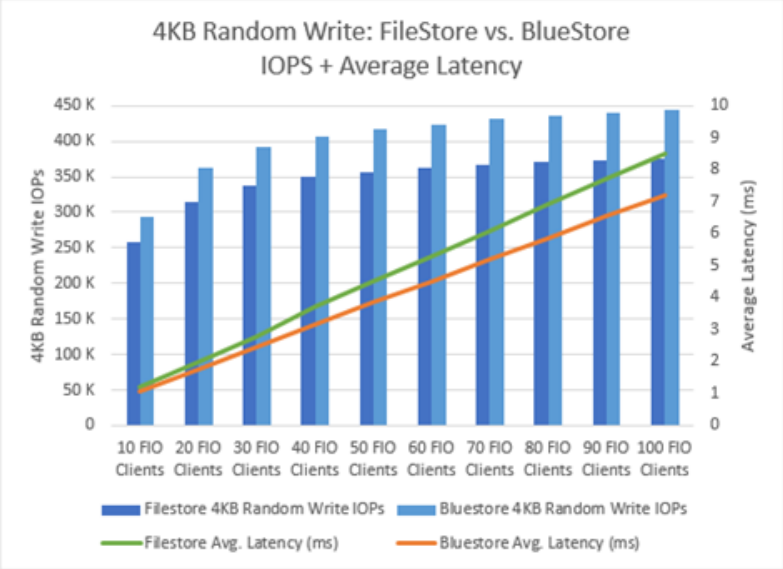

Figure 1.7: FileStore vs. BlueStore Performance [Mic]

Under the hood, the BlueStore writes the data blocks directly onto the block device, while the FileStore writes the data to a file system like XFS before it gets onto the block device. For the metadata, BlueStore is using RocksDB, which can run on a separate drive or the same drive as the data. The metadata is needed so the BlueStore can store in which blocks the data of an object is stored. The following figure also shows a comparison of how the structures of BlueStore and FileStore differ.

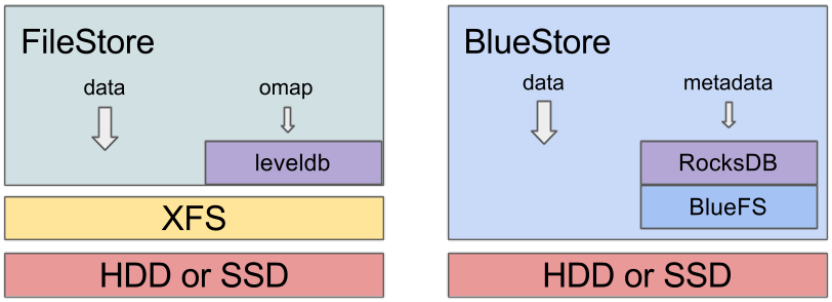

Figure 1.8: BlueStore vs. FileStore Performance [Cepb]

2 Planning

To utilize the strengths and performance advantages of the BlueStore in JULEA, the BlueStore first needs to be extracted from Ceph. As the BlueStore is a central part of Ceph and is relying on many internal functions of Ceph, it can not be separated completely. Instead, a plugin needs to be created, which provides an interface to all of the essential BlueStore functions that JULEA needs to access.

Additionally, this plugin requires to be written in C++, while it needs to provide a C interface for JULEA. Due to this, the plugin functions will only use C data types for their interface, with other data types, like the reference to the BlueStore, handled by a void pointer. As JULEA should only access this plugin directly, everything of Ceph that needs to be initialized also has to be handled by the plugin.

JULEA then requires a specific backend for the BlueStore, which accesses the plugin previously created. The backend should mostly be a wrapper for the plugin, as the interface for the plugin can already be created with a similar structure as the template for JULEA backends. The internal structs of the backend will also be used to store the void pointers for the BlueStore and its collection, which are needed for all functions accessing the BlueStore.

The plugin for the BlueStore will also provide a small test, which can be compiled and run independently from JULEA. This test is written in C, so the access is similar to the access from JULEA, but without the requirement of running a full JULEA installation. In addition to this test, the plugin can also be tested using the integrated test suite of JULEA, when it is used as the object backend in JULEA.

3 Implementation

3.1 BlueStore Library

The first part of the library is a struct that holds the information on the collection. This struct only contains the collection ID and the handle of the collection. In the functions using the collection, a reference to it will be used with a void pointer used for the C interface.

The second part of this library consists of the BlueStore operations, which are required to start and shutdown the BlueStore correctly. To launch Ceph itself, the `julea_bluestore_init` function first initializes Ceph, and afterwards, it creates the object of the ObjectStore and returns it. This can also be seen in the following listing where `global_init` and `common_init_finish` begin and end the initialization of Ceph. The arguments of `global_init` also define how Ceph should be launched, e. g. `CEPH_ENTITY_TYPE_OSD` says Ceph to launch as an object store daemon and `CINIT_FLAG_NO_MON_CONFIG` tells Ceph not to launch any monitoring. After the initialization, `ObjectStore::create` then creates the object store, which is returned as a void pointer to the C interface.

```
1 void *julea_bluestore_init(const char* path) {
2     vector<const char*> args;
3     ObjectStore* store;
4     cct = global_init(nullptr, args, CEPH_ENTITY_TYPE_OSD,
        ↪ CODE_ENVIRONMENT_UTILITY,
        ↪ CINIT_FLAG_NO_MON_CONFIG);
5     common_init_finish(g_ceph_context);
6     store = ObjectStore::create(g_ceph_context,
        ↪ string("bluestore"), string(path),
        ↪ string("store_temp_journal"));
7     return (void *)store;
8 }
```

Listing 3.1: Ceph Initialization

Using `julea_bluestore_mkfs`, the store can then be used to create the BlueStore filesystem onto the block device and initialize the database. This function is only a wrapper, just like `julea_bluestore_mount` and `julea_bluestore_umount`, which are used for mounting and unmounting the object store.

Additionally, the BlueStore also requires the use of collections where the objects are stored, so the third part of the plugin consists of the collection functions. These functions are used to create and open a collection, as well as to sync the data to the storage. The create and open functions of the collection are also the functions, which create the struct containing the collection, which was described previously. The data of this struct gets set then to the collection and the collection ID, and the struct is then returned as a void pointer to the struct. This pointer then has to be provided to every function, which operates on objects, as these operations require the corresponding collection.

The function `julea_bluestore_create_collection` is also the first function using the transactions from Ceph, as shown in listing 3.2. The transactions are always bound to a specific collection, but they have to be queued using the ObjectStore. The ObjectStore also has a specific function to queue multiple transactions at a time, these transactions have to be collected in a vector beforehand to utilize this function. Additionally, the transactions can be used with a mutex to ensure the data is readable from other functions or even securely saved on the storage device. For this use case, the callbacks `on_applied_sync`, `on_applied`, and `on_commit` exist. The first two callbacks are very similar and only ensure subsequent reads can see the data, with the sync variant being a little stricter as it is running in an ObjectStore executing thread, with the normal variant running in a separate thread. The `on_commit` callback is then the variant used to ensure the transaction is durably written to the storage device, so the data is crashproof. This is also the consistency level used by the flush function, which is used in the last collection function, called `julea_bluestore_fsync`, which is a simple wrapper of flush and provides basic functionality like `fsync` from POSIX.

```
1  void *julea_bluestore_create_collection(void* store) {
2      ObjectStore* ostore = (ObjectStore *)store;
3      BSColl* coll = new BSColl;
4      coll->cid = coll_t();
5      coll->ch = ostore->create_new_collection(coll->cid);
6      {
7          BlueStore::Transaction t;
8          t.create_collection(coll->cid, 0);
9          ostore->queue_transaction(coll->ch, std::move(t));
10     }
11     return (void *)coll;
12 }
```

Listing 3.2: Create Collection

Furthermore, the collection functions currently have a fixed collection ID and only use the default ID. This part of the code could be adjusted later, for something like JULEA's namespaces which could get their own collections.

There also are the file operations, and the first of those is `julea_bluestore_create`. The create function creates an object using the name, which is provided in the parameters of the function. At first, this function only creates an object, using the name given in the function parameters. Afterwards, the touch function is used for the object within the transaction, and the transaction is queued. Finally, the object gets returned for subsequent function calls. The full function is shown in the following listing.

```
1 void *julea_bluestore_create(void* store, void* bscoll,
    ↪ const char* name) {
2   ObjectStore* ostore = (ObjectStore *)store;
3   BSColl* coll = (BSColl *)bscoll;
4   ghobject_t *obj = new
      ↪ ghobject_t(hobject_t(sobject_t(string(name),
      ↪ CEPH_NOSNAP)));
5   ObjectStore::Transaction t;
6   t.touch(coll->cid, *obj);
7   ostore->queue_transaction(coll->ch, std::move(t));
8   return (void *)obj;
9 }
```

Listing 3.3: Object Create

The corresponding function to delete an object from the object store is called `julea_bluestore_delete` and provides a very similar structure with just two changes. The differences of this function are the `t.touch` call, which gets replaced by `t.remove`, and the object being provided in the parameters, instead of the object being created. To get the pointer to the object for the parameters, the objects do not require to be created using the create function first, as for existing objects, there also is an open function. Using the open function, a pointer to the object can be retrieved, as this function is only a wrapper for creating the `ghobject_t`, like in line 4 of the create function.

For I/O operations, the write and read functions are quite different, as only the write function uses a transaction. As the listing 3.4 shows, the `julea_bluestore_write` function recieves the ObjectStore, collection, and object and the creates its transaction. Then, it needs to create a bufferlist to convert the char array of data it receives from the C interface, to a bufferlist, which is the required datatype by the Ceph interface. When the data is stored in the bufferlist, the write function then gets called for the transaction, the transaction is queued, and the length of the bufferlist is returned to show how many bytes were written.

```
1  int julea_bluestore_write(void* store, void* bscoll, void*
     ↪ object, uint64_t offset, const char* data, uint64_t
     ↪ length) {
2    ObjectStore* ostore = (ObjectStore *)store;
3    BSColl* coll = (BSColl *)bscoll;
4    ghobject_t* obj = (ghobject_t *)object;
5    ObjectStore::Transaction t;
6    bufferlist bl;
7    bl.append(data, length);
8    t.write(coll->cid, *obj, offset, bl.length(), bl);
9    ostore->queue_transaction(coll->ch, std::move(t));
10   return bl.length();
11 }
```

Listing 3.4: Object Write

The read function `julea_bluestore_read` is quite a bit simpler than the write function, as the ObjectStore offers a direct read function. Therefore, no transaction is required, as shown in the following listing, which makes this function significantly shorter than the write function. The read function also creates a bufferlist, in which the data is returned. Using this bufferlist, the read function of the ObjectStore is called directly and fills the bufferlist. The bufferlist then gets converted to a char array and is put into the `data_read` pointer to return the data.

```
1  int julea_bluestore_read(void* store, void* bscoll, void*
     ↪ object, uint64_t offset, char** data_read, uint64_t
     ↪ length) {
2    ObjectStore* ostore = (ObjectStore *)store;
3    BSColl* coll = (BSColl *)bscoll;
4    ghobject_t* obj = (ghobject_t *)object;
5    bufferlist readback;
6    int ret = ostore->read(coll->ch, *obj, offset, length,
       ↪ readback);
7    *data_read = readback.c_str();
8    return ret;
9  }
```

Listing 3.5: Object Read

The last function is the status function, which is a fairly simple wrapper of the stat function, provided by the object store. Like the other file operation functions, it simply gets the object using `make_object` and then calls `ostore->stat` and returns the stat struct this function is providing.

3.2 JULEA Backend

With this BlueStore library, the next step is a backend for JULEA, which utilizes this library to store the data in an object store. The object backend of JULEA already has a predefined structure consisting of functions for `init`, `fini`, `create`, `delete`, `open`, `close`, `status`, `sync`, `read` and `write`. Of these functions, `close` will only be an empty shell in the BlueStore backend, as it is not used by BlueStore.

Since the library was already designed with the backend in mind, most of the other functions also are pretty simple wrapper functions, like the write function in 3.6. Additionally, as the write function also shows, the backend also requires the use of a struct (`JBackendData`) containing the path, the pointer to the object store and a pointer to the collection, so they are available in all functions. Furthermore, there is another struct called `JBackendObject`, which contains the path of an object, as well as the pointer to the object retrieved from the plugin.

```
1  static gboolean
2  backend_write(gpointer backend_data, gpointer
       ↪ backend_object, gconstpointer buffer, guint64 length,
       ↪ guint64 offset, guint64* bytes_written)
3  {
4      gsize bw = 0;
5      JBackendData* bd;
6      JBackendObject* bo;
7
8      bd = backend_data;
9      bo = backend_object;
10
11     j_trace_file_begin(bo->path, J_TRACE_FILE_WRITE);
12     bw = julea_bluestore_write(bd->store, bd->coll,
           ↪ bo->obj, offset, (const char*)buffer, length);
13     j_trace_file_end(bo->path, J_TRACE_FILE_WRITE, length,
           ↪ offset);
14
15     if (bytes_written != NULL)
16     {
17         *bytes_written = bw;
18     }
19
20     return (bw == length);
21 }
```

Listing 3.6: Backend Write

The `backend_fini` function is also pretty simple, as it only unmounts the BlueStore and frees the memory of the struct, but `backend_init` is a bit more complicated. This

function first initializes the `JBackendData` struct and fills the path variable of it with the path. Afterwards, the init function of the BlueStore library is called to initialize Ceph and then comes a check if the `mkfs_done` file exists in the mount location. If this is the case, the BlueStore only gets mounted, and the collection gets opened. If `mkfs_done` does not exist, the init function assumes the filesystem is not created so far, so `mkfs` is called first, then mount, and finally, a new collection gets created.

```
1  static gboolean
2  backend_init(gchar const* path, gpointer* backend_data)
3  {
4      JBackendData* bd;
5      gchar* mkfs_path;
6
7      bd = g_slice_new(JBackendData);
8      bd->path = g_strdup(path);
9      mkfs_path = g_build_filename(path, "/mkfs_done", NULL);
10
11     bd->store = julea_bluestore_init(path);
12
13     if (access(mkfs_path, F_OK) == 0)
14     {
15         julea_bluestore_mount(bd->store);
16         bd->coll =
               ↪ julea_bluestore_open_collection(bd->store);
17     }
18     else
19     {
20         julea_bluestore_mkfs(bd->store);
21         julea_bluestore_mount(bd->store);
22         bd->coll =
               ↪ julea_bluestore_create_collection(bd->store);
23     }
24
25     *backend_data = bd;
26
27     return TRUE;
28 }
```

Listing 3.7: Backend Init

3.3 Ceph Setup Scripts

The next essential parts are the shell scripts, which help to install and setup Ceph to use it with JULEA.

3.3.1 install-ceph.sh

The first script is `install-ceph.sh`, which creates a Ceph build from Ceph's GitHub repository [Cepc]. First, the paths of the JULEA and the Ceph directory are defined, with Ceph getting installed into a subfolder in the dependencies folder. Then, Ceph gets cloned from its GitHub repository into the Ceph directory, and then in the Ceph directory, all dependencies of Ceph are being installed. In the next step, `sed` is called to deactivate `malloc_usable_size` in RocksDB, as this can cause problems with the BlueStore Plugin. Afterwards, Ceph's CMake is run with Bluestore and BlueFS toggled on, but also tests, Rados Gateway, Manager, and Fuse turned off to reduce the size of the Ceph installation. Finally, the script moves into the build directory and starts the compilation with eight threads, which can be adjusted to the number of cores of your machine.

```
1  #!/bin/bash
2
3  SELF_PATH="$(readlink --canonicalize-existing -- "$0")"
4  SELF_DIR="${SELF_PATH%/*}"
5
6  . "${SELF_DIR}/common"
7
8  CEPH_DIR="$(get_directory
      ↪ "${SELF_DIR}/..")/dependencies/ceph"
9
10 git clone --recursive --single-branch
      ↪ https://github.com/ceph/ceph.git "${CEPH_DIR}"
11 cd "${CEPH_DIR}"
12 ./install-deps.sh
13 sed -i "s/  add_definitions(-DROCKSDB_MALLOC_USABLE_SIZE)/#
      ↪  add_definitions(-DROCKSDB_MALLOC_USABLE_SIZE)/g"
      ↪ "${CEPH_DIR}/src/rocksdb/CMakeLists.txt"
14 ./do_cmake.sh -DWITH_BLUESTORE=ON -DWITH_BLUEFS=ON
      ↪ -DWITH_TESTS=OFF -DWITH_RADOSGW=OFF -DWITH_MGR=OFF
      ↪ -DWITH_FUSE=OFF
15 cd build
16 make -j8
```

Listing 3.8: Ceph Install Script

3.3.2 prep-ceph.sh

The second script is the preparation script, which takes two input parameters to prepare the mount location for Ceph. These input parameters are the directory, where the BlueStore gets mounted, and the block device, where the BlueStore is supposed to store its data. So, using `/tmp/bluestore` and `/dev/sdb` as an example folder and block device, a typical script call would be `./scripts/prep-ceph.sh /tmp/bluestore /dev/sdb`.

This script then checks if the folder is existing, to either create the folder or to clear all files from the folder. When the folder is then created or cleaned up, the script creates a symlink to the block device in the folder, which is required by Ceph for the BlueStore. With this symlink, the folder then is correctly prepared, so JULEA can be started, and `mkfs` should work correctly.

```
1  #!/bin/bash
2
3  SELF_PATH="$(readlink --canonicalize-existing -- "$0")"
4  SELF_DIR="${SELF_PATH%/*}"
5  SELF_BASE="${SELF_PATH##*/}"
6  BLUESTORE_DIR=$1
7  BLKDEV=$2
8
9  usage ()
10 {
11     echo "Usage: ${SELF_BASE} BLUESTORE_DIR BLOCK_DEV"
12     exit 1
13 }
14
15 test -n "$1" || usage
16 test -n "$2" || usage
17
18 if [ ! -d "$BLUESTORE_DIR" ]; then
19     mkdir "$BLUESTORE_DIR"
20 fi
21 if [ "$(ls -A $BLUESTORE_DIR)" ]; then
22     rm -r "$BLUESTORE_DIR"/*
23 fi
24 ln -s "$BLKDEV" "$BLUESTORE_DIR/block"
```

Listing 3.9: Ceph Preparation Script

4 Benchmark

The benchmarks, to test the performance of the BlueStore against the performance of the POSIX backend, were run on a single hard drive locally on a node. For the BlueStore backend, a 100 GiB looping device was created, as the backend requires an own block device, while the POSIX backend ran directly on the hard drive. The node used consisted of an AMD Opteron 6344 CPU, 128 GiB RAM, and a Western Digital WD1003FBYZ hard drive. For compilation gcc 9.3.0 was used, and Ceph was built from the newest code from the Ceph GitHub Repo [Cepc], as of February 18, 2021.

The first benchmarks are for the file operations create, delete, status, and the batch variant for each variant. In the batch variant, JULEA collects all operations in a single batch and executes them at once, while the normal variant executes each operation on its own. Also, only the benchmark of the object backend was used, as the distributed-object and item clients behave very similarly.

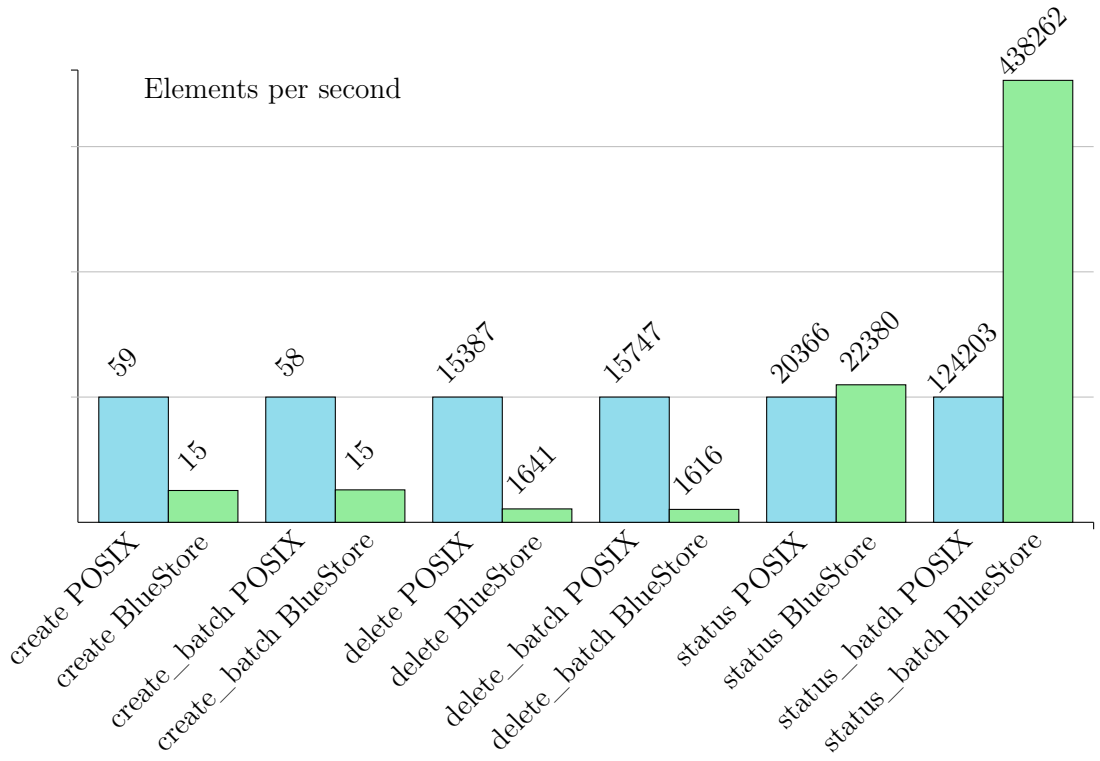

Figure 4.1: Create, Delete, and Status Performance

As these benchmarks show, the creates and deletes are significantly slower for the BlueStore backend than the POSIX backend. This might be due to a bit of an overhead of using a batch of JULEA, as well as a transaction of Ceph. Looking at the status benchmark, BlueStore is even faster than POSIX, especially in the batch variant. This also supports the assumption of the transactions slowing things down, as the status function does not use a transaction by Ceph. To mitigate this overhead, some deeper integration between batches and transactions might be required in the future.

For the read and write benchmarks, even more benchmarks were run, since POSIX and BlueStore behave differently with larger or smaller block sizes. So each benchmark was run five times each with six different block sizes, to show the performance differences. The first benchmark is the read benchmark.

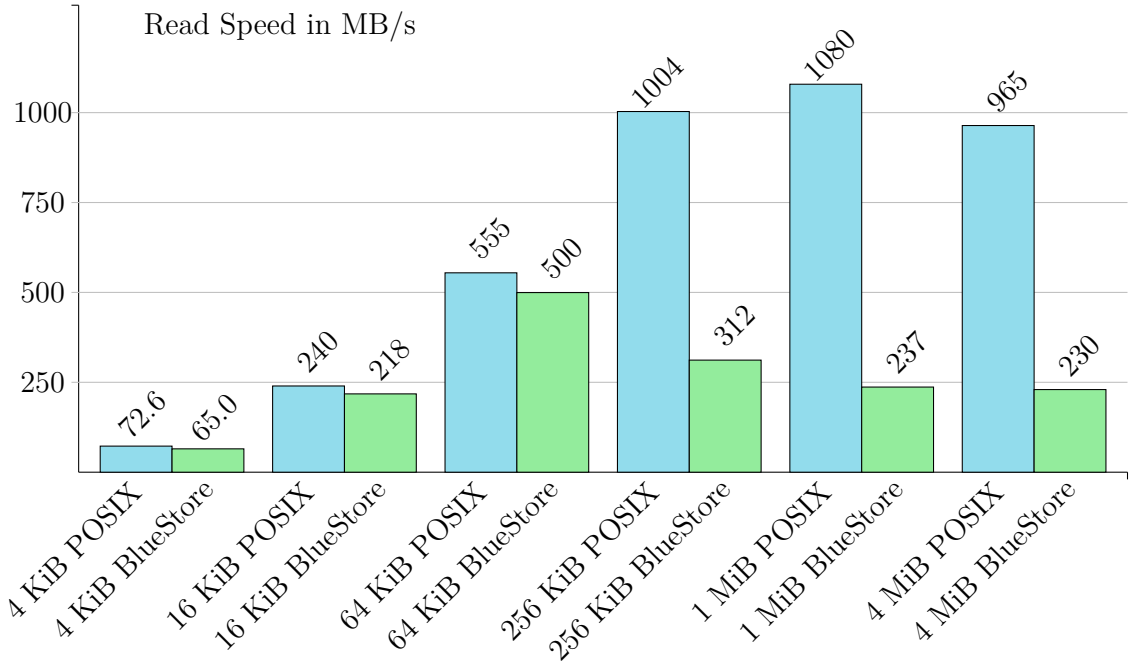

Figure 4.2: Read Performance

The read benchmark also shows a bit slower performance for the BlueStore backend, but while the difference is only about 10% for the small block sizes, it is much more for the larger block sizes. Especially for the larger block sizes, the performance reaches about 1 GB/s, which are impossible speeds for a hard drive, so there definitely is caching involved. And looking at the speed of the BlueStore, it seems like its cache reaches its storage limit with the 64 KiB benchmark, as the following sizes drop in performance. POSIX, on the other hand, can hold up the speed much longer, so it seems to have a larger read cache, which boosts the results of the larger block sizes.

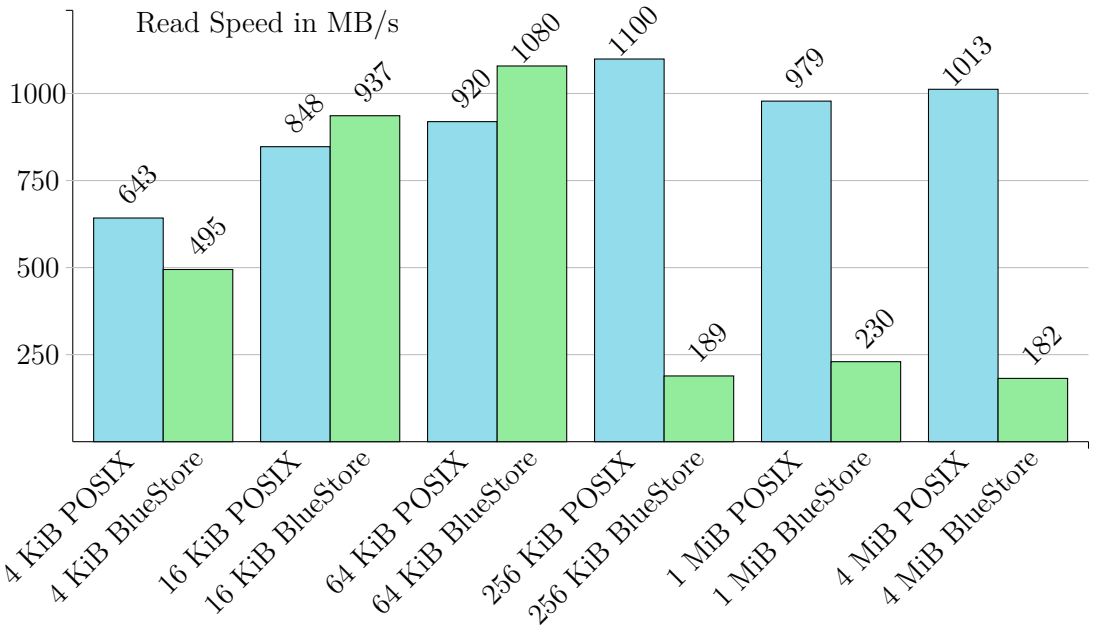

Figure 4.3: Read Batch Performance

Using the batch variant of the read benchmark, the situation looks quite similar, but here BlueStore even outperformed POSIX slightly with 16 and 64 KiB block sizes. With the larger block sizes, the same problem with caching kicks in, which gives POSIX a significant lead. Additionally, this benchmark works much better with small block sizes, since the blocks are executed at the same time and not every single one on its own.

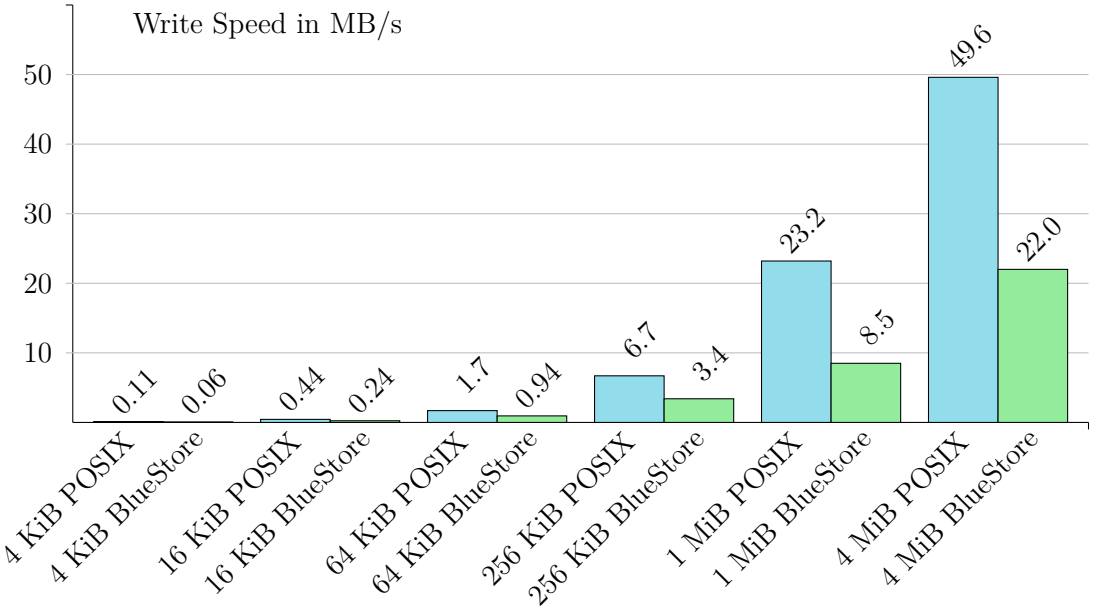

Figure 4.4: Write Performance

As this figure is showing, the normal variant of the write benchmark scales quite good with the block size. The scaling is similarly good between POSIX and BlueStore, but the BlueStore backend is consistently about half as fast as POSIX. With this difference in performance, it seems like there still is much room for optimization, which might partially have to do with the transactions.

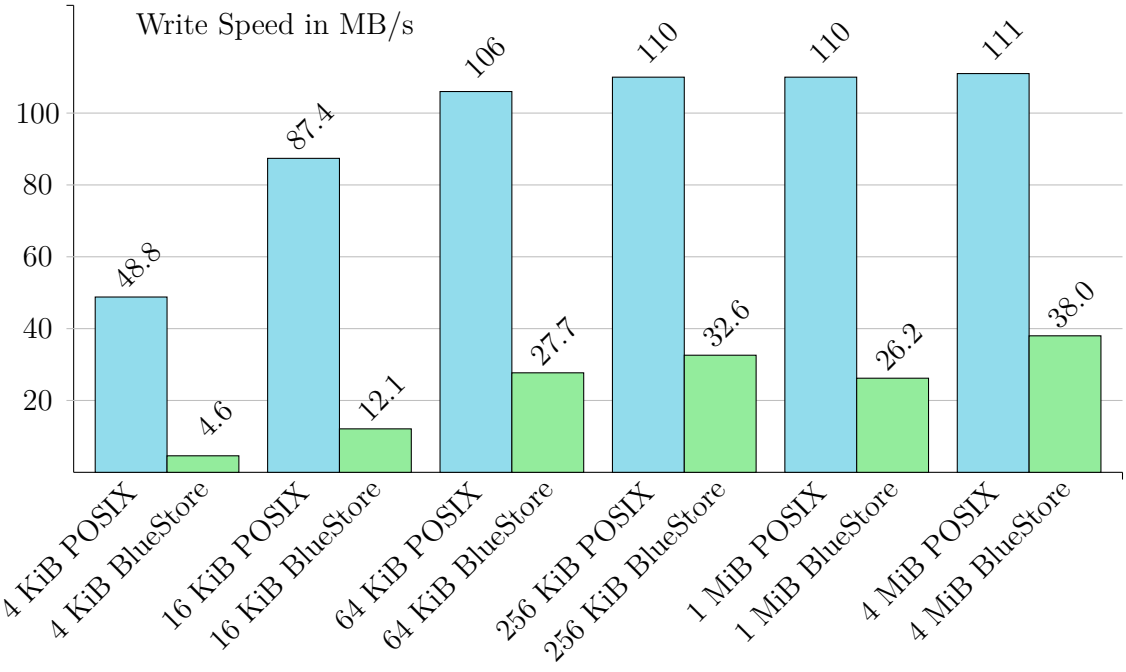

Figure 4.5: Write Batch Performance

Switching to the batch variant, the POSIX backend gains even more over the BlueStore backend and is about three times faster for the larger block sizes. For the very small block sizes, the difference is even more extreme, which, once again, supports the assumption of the transactions slowing things down. With the batch variant, POSIX simply has to execute a single batch and can write non-stop, while the BlueStore backend still has a single transaction for each write to execute. This is even more visible with the small block sizes, where the drive speed has less impact and more time gets spent with computation.

In conclusion, the performance currently is not quite on par with the standard POSIX backend, but it works as a proof of concept of adapting Ceph to JULEA. This project provided an interface to use the BlueStore in JULEA without using a full-blown Ceph setup, and with some optimizations, this might also outperform POSIX due to the advantages of an object store.

Bibliography

[Cepa] Ceph. *Ceph*. URL: https://ceph.io/ (visited on 10/06/2020).

[Cepb] Ceph. *FileStore vs. BlueStore*. URL: https://i0.wp.com/ceph.io/wp-content/uploads/2017/08/filestore-vs-bluestore-2.png (visited on 10/06/2020).

[Cepc] Ceph. *GitHub: Ceph*. URL: https://github.com/ceph/ceph/ (visited on 10/06/2020).

[Cepd] Ceph. *New in Luminous: BlueStore*. URL: https://ceph.io/community/new-luminous-bluestore/ (visited on 10/26/2020).

[Groa] HDF Group. *HDF5*. URL: https://www.hdfgroup.org/solutions/hdf5/ (visited on 10/26/2020).

[Grob] HDF Group. *HDF5 Looping Group*. URL: https://support.hdfgroup.org/HDF5/doc1.6/UG/Images/Group_fig2,8.jpg (visited on 10/09/2020).

[Gro16] William Gropp. *Lecture 32: Introduction to MPI I/O*. 2016. URL: http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf (visited on 10/09/2020).

[Kuh17] Michael Kuhn. "JULEA: A Flexible Storage Framework for HPC". In: *High Performance Computing*. Ed. by Julian M. Kunkel et al. Cham: Springer International Publishing, 2017, pp. 712–723. ISBN: 978-3-319-67630-2.

[Loc] Glenn Lockwood. *What's So Bad About POSIX I/O?* URL: https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/ (visited on 10/26/2020).

[Mic] Micron. *FileStore vs. BlueStore Performance*. URL: https://www.micron.com/-/media/client/global/images/blogs/content-images/2018/blog_image_ceph_bluestore_3.png (visited on 10/06/2020).

[Ros] Red Hat Ross Turk. *Ceph I/O Stack*. URL: https://raw.githubusercontent.com/ceph/ceph/master/doc/images/stack.png (visited on 10/06/2020).

[Sci] Neon Science. *HDF5 Structure*. URL: https://www.battelleecology.org/sites/default/files/images/HDF5/hdf5_structure4.jpg (visited on 10/09/2020).

[UCAa] UCAR. *Interoperability with HDF5*. URL: https://www.unidata.ucar.edu/software/netcdf/docs/interoperability_hdf5.html (visited on 10/26/2020).

[UCAb] UCAR. *NetCDF*. URL: https://www.unidata.ucar.edu/software/netcdf/ (visited on 10/26/2020).

[Wika] Wikipedia. *POSIX*. URL: https://en.wikipedia.org/wiki/POSIX (visited on 10/26/2020).

[Wikb] Wikipedia. *Virtual File System*. URL: https://en.wikipedia.org/wiki/Virtual_file_system (visited on 10/26/2020).

List of Figures

List of Listings