Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Master-Project Report

JULEA - Transformation/Compression

vorgelegt von

Michael Blesel, Oliver Pola

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

Abstract

In this project report we will show our work of implementing a transformation object client for JULEA. We give an introduction to JULEA and data transformations and compression followed by a detailed explanation of our design and implementation choices. We highlight the challenges we faced during development and how we solved them.

We will end with benchmark comparisons between the transformation client and the standard JULEA object client and reason about how usable our implementation is for real world applications.

Contents

1 Introduction

JULEA [Kuh17] is a distributed file system, that is described in Chapter 2. Since it runs in user space, it is more easy to implement new features than in standard established file systems. This offers students and researchers the ability to experiment with file system ideas and proofs of concept.

This report describes our project to enhance JULEA with the ability to apply data transformations in general. The main use case for such a transformation is data compression to either reduce storage usage or network traffic. Having compression in mind, we try to keep the basic concepts generic, such that any kind of transformation (e.g. encryption) should be more easy to implement in follow-up projects.

The transformation feature will be a proof of concept and only offer a few simple algorithms to show the basic ideas. Our implementation then can be used as a template to further enhance JULEA, for example with multiple compression algorithms, if one wants to study their performance impact or storage capacity benefits.

The basic concepts of compression and transformations in general are discussed in Chapter 3. Then Chapter 4 lists specific challenges that need to be addressed when implementing those concepts. Our solutions to those challenges and the structure of our implementation are described in Chapter 5.

In Chapter 6 we discuss some benchmark results and, after offering some possible future work in Chapter 7, we conclude our project in Chapter 8.

2 JULEA

In this chapter we will give an introduction to the JULEA framework. We will look at the design, its benefits compared to complex modern file systems and go through the installation process and a small example application.

2.1 The JULEA framework

JULEA [Kuh17] is a flexible storage framework developed by Michael Kuhn at the University of Hamburg. It offers arbitrary I/O interfaces to applications and supports a variety of different backends. Due to its uncomplicated and quite modular implementation, JULEA lends itself to experimentation and prototyping of new and interesting file system ideas. JULEA runs in user space, which makes it much easier for developers to make modifications and to add additional functionality. It is also implemented very concise and in a minimalistic way, which makes it feasible for students and non experts in file systems to understand and to experiment with implementing their own ideas. This is a clear benefit in contrast to today's very complex and often monolithicly built file systems.

JULEA has a standard client-server model design where user applications use the client interfaces to communicate with the JULEA servers via network. Multiple clients are already implemented, like for example an object client, a key-value store and an item client that provides a cloud-like interface with collections and items in a flat hierarchy. Developers can also quite easily create their own clients and add them to JULEA. On the server side different backends such as POSIX, LevelDB and MongoDB are available. Here it is also possible for developers to include their own backends.

In this report we will show how we implement our own transformation client, which will internally make use of the provided object and KV clients. In the next sections we will give a brief overview of how to set up and use JULEA from an application.

2.2 Setting up JULEA

The installation and setup of JULEA is very straight forward. The source code can be pulled from from GitHub[1]. JULEA has only a few required dependencies such as GLib and libbson but the different clients and backends might bring their own dependencies. To make the dependency handling easier for users, JULEA provides an `install-dependencies.sh` script which can be run with different options depending on how many optional dependencies should be installed. To accomplish the dependency

[1] `https://github.com/julea-io/julea`

installation JULEA makes use of Spack [GLC+15], which is a package manager that is often used on HPC clusters. Spack provides automated building and version control of many software packages. Spack builds and installs the software locally for the user and is therefore very helpful for non root users on for example a supercomputer where they would not have access to the system package manager. Installed Spack packages can then be loaded as environment modules. JULEA's dependency script creates a local Spack installation and then automatically builds all needed dependencies.

The JULEA build process uses Meson for configuration and Ninja. The required commands can be found in the 'Quick Start' section of the JULEA GitHub repository. The last setup step is to create a JULEA config file by following the instructions in the quick start guide. After these three steps JULEA should be usable.

2.3 Creating a JULEA applications

Listing 2.1 shows a simple application using JULEA's object client. We will use this example to explain the general workflow of a JULEA application.

In the include section of the code we can see that the general JULEA header (`julea.h`) is included and we also need to include the header for the object client (`julea-object.h`). This already shows some of the modular nature of JULEA. The different clients are independent of each other and if a user for example does not want to install the dependencies for a client, they do not have to build the client and can without any problems still use the other clients.

The first thing we can see in the main function is the use of multiple GLib types. JULEA's implementation makes heavy use of GLib features (mainly for memory management) and our example codes in this report will mirror this behaviour. The code makes use of Glib's `g_autoptr`, which is essentially a kind of smart pointer that handles the cleanup for an object when it gets dropped. This Glib macro works for all types that have defined a matching Glib cleanup function, which is the case for most JULEA types and it can therefore be used to avoid the need for manual cleanup code in JULEA applications.

The first two JULEA specific types in the code appear in lines 9 and 10. First we have a `JSemantics` variable. JULEA supports different I/O semantics like POSIX or MPI-IO. We will not go into detail about that in this report and it can be generally assumed that we are working under POSIX semantics.

The `JBatch` type in line 10 is very important to the way how I/O operations are handled by JULEA. In general, and specifically for the object client, all operations on objects can first be gathered in a batch. This means that for example a write operation is not immediately executed when, like in line 21, the `j_object_write` function is called. The operations are first added to a batch and the actual write process is only run when the corresponding batch is executed with the `j_batch_execute` function. This functionality is useful from a performance standpoint since multiple operations can first be gathered and the possible overhead of for example connecting from the client to the server is minimized. In our example code we do not make much use of this benefit but

```c
#include <glib.h>
#include <julea.h>
#include <julea-object.h>
#include <stdio.h>

int main(int argc, char** argv)
{
	gboolean ret = false;
	g_autoptr(JSemantics) semantics =
		↪ j_semantics_new(J_SEMANTICS_TEMPLATE_POSIX);
	g_autoptr(JBatch) batch = j_batch_new(semantics);
	gchar data[10] = {0,1,2,3,4,5,6,7,8,9};
	gchar buf[10];
	guint64 bytes_written = 0;
	guint64 bytes_read = 0;

	g_autoptr(JObject) object = j_object_new("t_namespace", "test");

	j_object_create(object, batch);
	ret = j_batch_execute(batch) && ret;

	j_object_write(object, data, 10, 0, &bytes_written, batch);
	ret = j_batch_execute(batch) && ret;

	j_object_read(object, buf, 10, 0, &bytes_read, batch);
	ret = j_batch_execute(batch) && ret;
	for(guint i = 0; i < 10; i++)
		printf("%d\n", buf[i]);
	printf("bytes read: %ld\n", bytes_read);

	j_object_delete(object, batch);
	ret = j_batch_execute(batch) && ret;

	return ret;
}
```

Listing 2.1: A small example of an application using the JULEA object client

always execute the batch after adding one operation to make it more clear when object operations happen.

Finally in line 16 we declare our `JObject` by using the `j_object_new` function. This returns a handle to an object with the given namespace and name. It is important to notice that at this point no new object has actually been created on the server. The `j_object_new` function only gives us a handle and if an object with the given name under the given namespace already exists on the server we would now have a valid handle. In this case no such object is already in existence and we have to create one in lines 18/19. Here we can see that all functions that operate on objects take a batch as an argument. The actual object creation happens when the corresponding batch is executed. Here the client opens a connection to the server/backend and sends a message containing the required information to create the 'test' object in the 't_namespace' namespace.

In the following lines we can see the same pattern for a write and a read operation on the object. This is followed by a delete operation where we delete the physical record of the object from the server. This delete is of course optional and since we are dealing with a file system the object would otherwise be kept persistent on the server and could be accessed from different user applications at any time.

This concludes the explanation of our small example application. The provided code is complete and an example Makefile for how to build JULEA applications can be found in the example directory of the JULEA repository.

3 Compression and Transformations

In this work we consider a compression algorithm to be a specific example of a data transformation. Since we consider data compression as the most useful task of any transformations on storage systems, we mostly focus on this example.

A transformation in general can be any algorithm that changes the data. Here a user provides a before-image of the data (also called original data) and applying the transformation leads to an after-image of that data (called transformed or compressed data). The transformed data then can be used for transport or persistent storage and may have beneficial properties, e.g. being smaller in size.

It is always desired to get the original data back to the user, therefore only transformations f are considered where an inverse-transformation f^{-1} exists to compute the original data from the transformed data.

$$f(\text{original data}) = \text{transformed data}$$
$$f^{-1}(\text{transformed data}) = \text{original data}$$

Later other properties of transformations are discussed as well.

3.1 General design

The first design decision is what part of the system applies the transformation. JULEA is a client-server system, so the transformation can happen on the client or the server side, or both. We want to compare the different designs, so our implementation has to offer multiple modes.

- **Client transformation**: The transformation is applied by the client. On write, the already transformed data is transferred and the receiving server is unaware of the transformation. On read, the client inverse-transforms the data after the transfer from the server.

  When the data is compressed, the benefits are less data to transfer and less usage of persistent storage.

  The workflow for a write is shown in Figure 3.1:

  1. The user writes memory blocks A, B and wants them to be stored
  2. On a JULEA object a write-operation (w) is called for each block
  3. Given the client mode, the object now applies the transformation and generates new memory blocks A', B'

4. The transformed data A', B' is transferred to the server's memory
5. The server backend stores the transformed data A', B' to disk

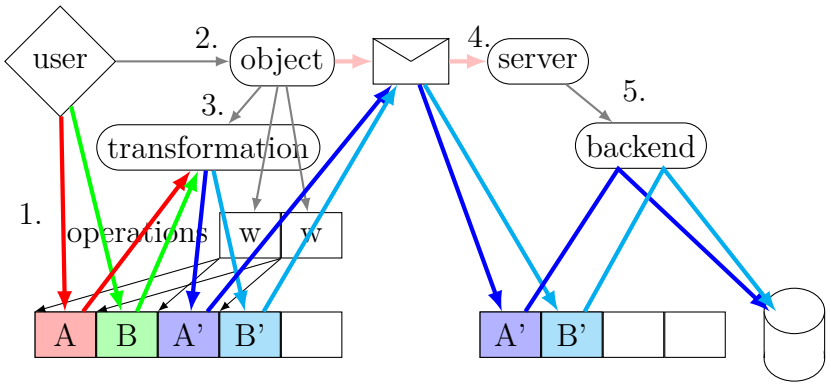

Figure 3.1: Client transformation workflow on write

On read the workflow is reversed accordingly:

1. The user requests some data from a JULEA object to target memory
2. The server has the transformed data in storage, that is now transferred to temporary memory on the client
3. From temporary to target memory, the data is inverse-transformed by the client
4. The user can access the original data in target memory

- **Server transformation**: The transformation is applied by the server after receiving it from the client, and in the other direction before sending requested data to the client. The client is totally unaware of the transformation, so this could be applied even if the client does not support transformation.

  When the data is compressed, the uncompressed larger data has to be transmitted, but the usage of persistent storage benefits from compressed data size.

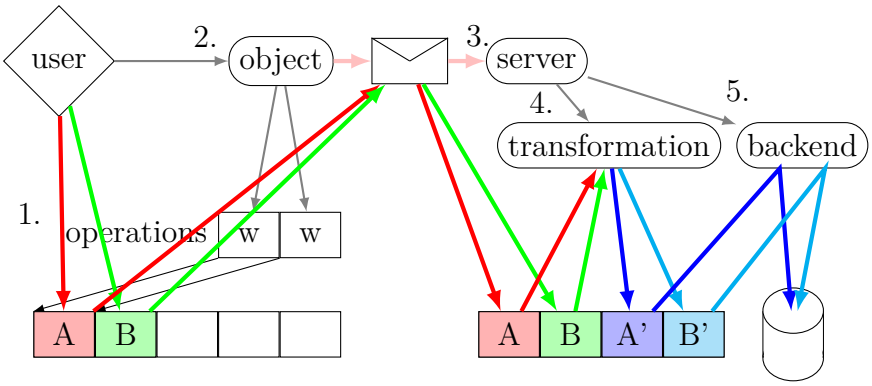

Figure 3.2: Server transformation workflow on write

The workflow for a write with server transformation is shown in Figure 3.2:

1. The user writes memory blocks A, B and wants them to be stored
2. On a JULEA object a write-operation (w) is called for each block
3. The original data A, B is transferred to the server's memory
4. Given the server mode, the server now applies the transformation and generates new memory blocks A', B'
5. The server backend stores the transformed data A', B' to disk

On read the workflow is again reversed:

1. The user requests some data from a JULEA object to target memory
2. The server has the transformed data in storage, that is loaded to server memory
3. The server applies the inverse-transformation and stores the original data in additional server memory
4. The original data is now transferred to target memory on the client
5. The user can access the original data in target memory

- **Transport transformation**: The idea is to combine both methods and apply the transformation before sending data over the network and apply the inverse-transform directly after receiving the data on the other side.

  For compressed data, this does not enhance storage utilization, because the original data is stored on disk. The compression is only applied for transportation to reduce network traffic.

3.2 Important transformation properties

To implement different transformations the following properties determine the implementation details that have to be considered.

- **Changing data size**: The most crucial information about the transformation algorithm is, whether the resulting data size is the same as the original data size or not. Memory in the resulting size has to be allocated before the transformation can be applied. Also disk space has to be allocated in the resulting size, not the original size. Of course compression algorithms aim for changing the size to the minimum possible result.

- **Context-free changes**: If the original data is the concatenation of A, B, C, some transformations f will transform each subset to A', B', C' and the overall result is

$$f([A, B, C]) = [f(A), f(B), f(C)] = [A', B', C'],$$

the concatenation of the partial results. Furthermore the transformation of B does not depend on the context A and C, so we could replace B by a new content B_{new} and the transformation $f([A, B_{new}, C])$ will then be $[A', B'_{new}, C']$.

Given this property, it might be necessary to move C', if sizeof(B') $\neq$ sizeof(B'_{new}), but A and C have not to be transformed again.

This property is not given for secure encryption transformations, where the change of one byte of the input will most likely change any byte of the output. And this is also not given for effective compression algorithms.

- **Partial access**: A transformation has the partial access property, if the data size does not change and the transformation is context-free. If then a certain byte of the original data is changed, only that changed byte needs to be transformed and the transformed data can be updated at the fixed offset of the change. And any partial update only needs to consider all the bytes that were changed.

  In this case the transformation may also be applied in-place. But no in-place update should ever be performed on write in client mode, because the client will probably continue to work with the memory blocks it wants to be written, so they must not change.

3.3 A subset of transformations to implement

The proof of concept will be shown on a small subset of possible transformations. The following methods are chosen to cover different special cases and are considered to be in increasing order of difficulty:

- **None**: Independent of how transformations are implemented, it should be possible to apply no transformation, even if a transformation client is used. This should also be performant and skip the transformation steps, especially not make a copy of the data.
- **XOR**: The XOR method simply applies a fixed binary pattern via XOR to the data. The inverse-transformation is just to apply XOR with the same binary pattern again. This is the most simple method that at least changes the data and has the partial access property, so it is chosen to be implemented first to test the concept and implementation in general.

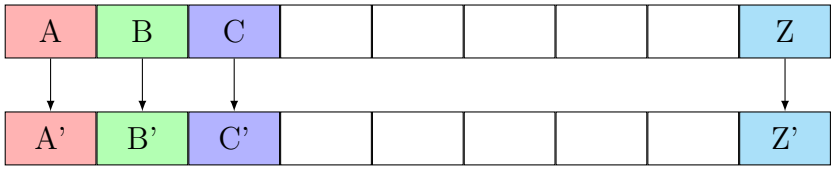

Figure 3.3: Schema of XOR transformation

- **RLE**: The run-length-encoding is often used as a very simple compression algorithm, when data contains many repetitions. The data is split into multiple sections, where each section just repeats the same byte. For each section the compressed data contains the number of repetitions and the used byte only once.

  This is a very simple method that we could implement very fast on our own. It is used as an example for changing data size. Please note that in some cases the transformed data size might be even larger than the original size. Such cases can be constructed to test the robustness of our implementation.

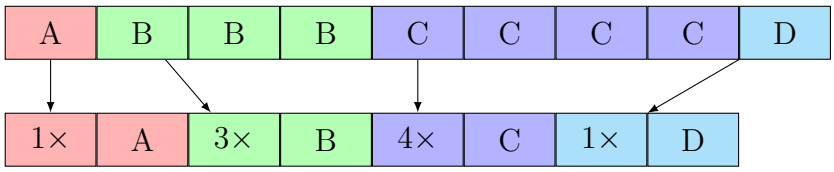

Figure 3.4: Schema of RLE transformation

- **LZ4**: Finally we wanted to implement an effective compression algorithm. For the LZ4 compression algorithm [Yan19], the library `liblz4` is used. Using a library and extending the list of supported transformations within JULEA is considered to be the way how our proof of concept can be extended to cover a larger variety of algorithms.

  The LZ4 algorithm is not explained here, but it is considered to be efficient and complex in the sense of changing one byte of input does change almost all of the output.

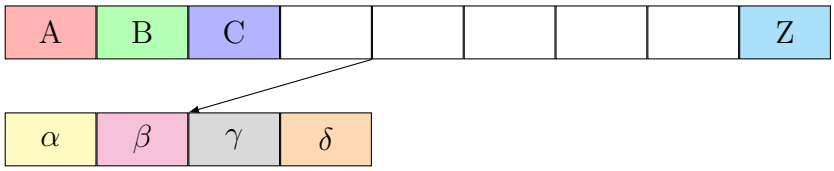

Figure 3.5: Schema of LZ4 transformation

4 Challenges

In this chapter we outline the different challenges that we faced during the planning and implementation of the transformation client. We were able to solve some of these challenges and had to make some compromises on others. Our solutions to these problems will be discussed further in Chapter 5 and Chapter 6.

4.1 General concept and structure

We started the project with the general idea of introducing transformation and compression functionalities to JULEA. Before actually starting to implement something we had to make some concrete design decision on how we wanted to integrate our new functionalities within JULEA's code base.

At first we had to decide whether we wanted to add the transformation ability to the already existing JULEA types or if we wanted to introduce a new client type. We rather quickly decided that it would be more appropriate to create our own transformation client instead of adding the it to the `JObject` type. This decision was based on multiple factors. Firstly our transformation concept needed to introduce changes to the `JObject` interface. This might have broken existing JULEA applications and would have been non-ideal. Furthermore the addition of the transformation code would probably have introduced some computational overhead to `JObject`, even if no transformation behavior was used. Another concern was the introduction of additional dependencies like for example `liblz4` to the core of JULEA, which would also be unwanted for users that are not interested in using the transformation features. Due to these reasons we decided to add a new client type called 'transformation' with its own types like `JTransformation` and `JTransformationObject`.

The next design decision was on how we wanted to provide the transformations to the user and which kinds of transformations we wanted to support. Here we decided on providing a `JTransformation` type which holds all necessary information about a transformation.

- **Transformation type:** The transformation type describes the used transformation algorithm. This could be anything from a simple XOR of the data to complex compression algorithms or even encryption of the data. It is important to notice here that depending on the used transformation algorithm different properties apply to the transformed data. For example it is important to know how the transformation will influence the size of the data. With a simple XOR the data size does not change at all but with other transformation algorithms the original

size of the data and the transformed size of the data will very likely differ. This already introduces a new type of metadata information that needs to be stored for the object to make sure that for example read operations can be executed correctly. It is also important to keep in mind that even when using a compression algorithm you are not always guaranteed that the size of the transformed data actually has decreased. This can for example be the case if a very small amount of data with a pattern that is not good for the compression algorithm gets transformed.

The other important property of transformed data is whether partial reads and writes can be executed on it. If we go back to the XOR example it is without a problem possible to read and inverse-transform just a subset of the transformed data block. This is however not possible for data that has been transformed using a compression algorithm. To read just a few bytes in the middle of a compressed data block the whole data has to be uncompressed first and then the read on the wanted bytes can be performed. This property has a strong impact on the implementation of a transformation object and also brings some serious performance concerns.

- **Transformation mode:** The transformation mode describes where the actual transformation of the data takes place inside JULEA. We looked at three specific scenarios here.
  1. **Client mode:** In this mode all transformation and inverse-transformation of the data is handled on the client side. This means that the server can be unaware that it even stores transformed data and it specifically does not need to know about the used transformation algorithm. The client is completely in charge of handling the needed metadata and doing the correct transformations. This however poses the problem that even a client that has not created a transformation object needs to be able to get access to the transformation information metadata to be able to work correctly with the object. It also keeps all computation effort of executing the transformation algorithms on the client side which could in theory be detrimental to the performance of applications but on the other hand also be beneficial to the server if it is often under high workloads.
  2. **Server mode:** In this mode the server takes complete care of all transformations and inverse-transformations. This makes the clients able to act mostly unaware of the transformation and just interact with the transformation object as if only the original untransformed data was stored on the server. In this scenario the server needs to take care of the objects metadata and it needs access to this information to correctly handle read and write requests from the clients. This approach might in some cases produce more network load because uncompressed data is sent to the server and it is only compressed there. For specific read patterns and transformation types it could also lessen the amount of data that needs to be sent. If for example a client just wants to read the first few bytes of a big amount of compressed data the server can do the inverse-transformation of the whole data block and just send back the

needed bytes. If the same scenario would take place in client mode, the server would need to send the whole compressed data block back to the client which then in turn inverse-transforms it and only uses the first few bytes of the sent data.

3. **Transportation mode:** Transportation mode is used to lessen the network load between client and server. In this mode all data that has to be sent is compressed beforehand and gets decompressed at arrival. This is a very specific mode for niche use cases and was not regarded very much by us.

As we can already see the different combinations of transformation types and modes will have a big influence on how to implement the transformation client best and especially on how the object's metadata has to be stored and managed. We also tried to plan our design of the `JTransformation` type in a way that it would be easy to add additional transformation algorithms in the future, since we could only implement a few our self in the scope of this project and future users might have very specific needs for the type of transformation that they want to use.

Our last design decision was that we wanted to keep the interface of our newly introduced `JTransformationObject` type as close as possible to the already familiar `JObject`. This keeps the workflow with our own client very close to using the already existing JULEA objects and it should be quite easy to migrate functioning code that uses `JObject` to use our client instead.

4.2 Handling partial object writes and reads

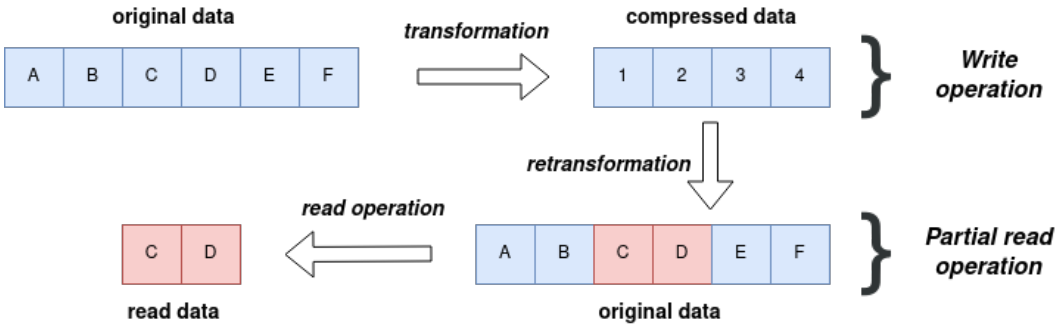

Figure 4.1: Workflow of a partial read on compressed data

As we have already mentioned in the previous section depending on the transformation type it is not possible to just access a substring of a transformed data block to do a correct inverse-transformation on it. In Figure 4.1 we can see an example of a partial read on an object that uses a compression algorithm as transformation. We can see that the original data consists of six blocks which get compressed into four blocks by an initial write to the object. In the second step a client sends a read request for the two

middle blocks of the original data. Since it is not possible to partially decompress the transformed data to get the original content of the requested blocks, the application first needs so inverse-transform the complete data. After decompression we get the original state of the data back and can now answer the read request and provide the requested data blocks. An even worse scenario would be a partial write to the object. In that scenario the whole data would again have to be decompressed first. Then the write can be executed and the whole resulting data has to be compressed again and written to persistent storage.

As we can clearly see this whole process can have a grave influence on the performance of an application. Especially if the I/O access patterns of the application contain many small partial reads and writes to a big object it could get very problematic.

The other important thing for the implementation of this process is that additional buffers are needed to store the intermediate processing steps of the data. In the given partial read example JULEA needs to provide a buffer that is big enough to hold the complete uncompressed data from which the actual data of the actual read request can be sent to the user application.

To mitigate some of these problems we quite early during development decided that some chunking of the object's data was needed to avoid cases where a huge object needs to be completely inverse-transformed to accomplish a small partial read. We will go into more detail about this in Section 5.4.

4.3 Metadata management for object persistence

As we have already discussed, the transformation objects need metadata information to be used correctly. The used transformation algorithm and also information about the original and the transformed size of the object are necessary for performing the correct read and write operations on an object. As long as we are only dealing with one client that creates the object and then also deletes it during the client's lifetime this is a simple task and this information can be stored on the client side inside the transformation object type. However this is not how objects in a file system are normally accessed by general applications. The stored transformation objects must be able to be used by every client that requests them as long as they are existent on the server. Therefore the required metadata also has to be stored persistently on disk and every client needs to load it automatically when requesting a reference to a transformation object. To solve this we had two different ideas in mind when designing our transformation client.

1. **Store metadata inside the object:** The first idea was to store the required metadata together with the object data. This would for example mean that each transformation object's data starts with a prefixed header which stores the metadata. With this solution each access to the object would first need to read the object header from disk and then extract the different metadata attributes from it. Furthermore each modification of the object would also need to overwrite the header with the new metadata. This approach comes with its advantages and drawbacks. The clear advantages is that the metadata and the object data

are always stored at the same place and it makes it trivial to find the matching metadata for an object. A drawback of this approach are the aforementioned many small read and write operations on the object that are required. Another small problem is the fact that in such an implementation the exact size of the header data block always needs to be known since each write and read operation needs to take the header offset into account to reach the correct object data bits. This would make it harder to add or remove new metadata attributes after the fact and also make it difficult to support a varying number of metadata fields for different types of transformations.

2. **Store metadata and object data separate:** With this method the metadata of a transformation object is stored in a different place, like for example the JULEA key-value store. This approach of course requires the clients to be able to find the correct place where the metadata of the current object is stored. The main advantage of this version is an easier access to specific metadata fields by using the features of JULEA's key-value store. It also solves the problem of having different metadata attributes for different types of transformation objects better than the previous method. The main drawback of this approach is the requirement of using the key-value client with every transformation client application.

In the end we decided to implement the second approach, mainly to avoid the necessary multiple reads on the objects mentioned in the first approach and because we felt it would be easier to expand on if the need for more specific metadata for different transformation objects would arise. It would be interesting to do some measurements for both methods to get an idea about which version would provide better performance.

4.4 Concurrent object access

One last challenge was how we would handle the concurrent access of multiple clients on the same object. This scenario can produce a multitude of errors for clients. The concurrent read access on an object should be unproblematic and has no need of specific safeguards but concurrent writes or reads and writes can lead to undefined behaviour. These problems also get amplified when metadata is taken into account and could even lead to crashes. If for example client A writes to an object and with that changes its transformed size and client B tries to read from that object at the same time but before the metadata of the write operation has been updated the whole inverse-transformation step would fail.

The most common way to avoid these concurrency problems is the usage of locks. Each operation that modifies the object should claim an exclusive lock on the object and its metadata. However at the time of our transformation client implementation JULEA does not yet provide locking functionality and the problems described above could also occur for the normal object client. Therefore we did not yet solve this problem in our implementation and can therefore not guarantee correct behaviour for concurrent transformation object accesses.

5 Implementation

In this chapter we will discuss our implementation of the transformation client. We will show the final state of our implementation and talk about our design decisions and how we tried to solve the challenges presented in Chapter 4.

5.1 General structure of the transformation client

The implementation of our transformation client mainly consists of three new JULEA types:

- `JTransformation`
- `JTransformationObject`
- `JChunkedTransformationObject`

We will go into detail about them in the following sections after we explain the general concept and workflow of our client.

Our main idea behind the provided interface for the transformation client was to keep it as close as possible to the regular JULEA object client. The transformation object types provide a nearly identical API with only some added parameters that are needed to specify the wished transformation type and mode.

In Listing 5.1 we can see the same example code as in Listing 2.1 with the only difference being that it uses a `JTransformationObject` instead of the regular JULEA `JObject` type. If we compare the two examples we can see that the only real difference, besides now using the `JTransformationObject` API functions, is at the object creation in line 18. To create a new transformation object we need to specify the aforementioned transformation type (as in which transformation algorithm to use) and transformation mode (server- or client-side transformation). Everything else behaves exactly the same as the regular object type from the perspective of the user.

We tried to keep the transformation logic as hidden as possible from the user as to not create any confusion or implementation effort. This should make it very easy to try out our transformation client with already existing JULEA applications. All that needs to be done is to exchange all `JObject` references and functions with `JTransformationObject` ones and the application should run exactly the same as before on a surface level.

We will now go on with a deeper look into the inner workings of our newly introduced types.

```c
#include <glib.h>
#include <julea.h>
#include <julea-transformation.h>
#include <stdio.h>

int main(int argc, char** argv)
{
	gboolean ret = false;
	g_autoptr(JSemantics) semantics =
		↪ j_semantics_new(J_SEMANTICS_TEMPLATE_POSIX);
	g_autoptr(JBatch) batch = j_batch_new(semantics);
	gchar data[10] = {0,1,2,3,4,5,6,7,8,9};
	gchar buf[10];
	guint64 bytes_written = 0;
	guint64 bytes_read = 0;

	g_autoptr(JTransformationObject) object =
		↪ j_transformation_object_new("t_namespace", "test");

	j_transformation_object_create(object, batch,
		↪ J_TRANSFORMATION_TYPE_LZ4, J_TRANSFORMATION_MODE_CLIENT);
	ret = j_batch_execute(batch) && ret;

	j_transformation_object_write(object, data, 10, 0, &bytes_written,
		↪ batch);
	ret = j_batch_execute(batch) && ret;

	j_transformation_object_read(object, buf, 10, 0, &bytes_read, batch);
	ret = j_batch_execute(batch) && ret;
	for(guint i = 0; i < 10; i++)
		printf("%d\n", buf[i]);
	printf("bytes read: %ld\n", bytes_read);

	j_transformation_object_delete(object, batch);
	ret = j_batch_execute(batch) && ret;

	return ret;
}
```

Listing 5.1: A small example of an application using the JULEA transformation client

```
1  struct JTransformation
2  {
3      JTransformationType type;
4
5      JTransformationMode mode;
6
7      gboolean partial_access;
8  };
9
10 void
11 j_transformation_apply(JTransformation* trafo, gpointer input,
12                 guint64 inlength, guint64 inoffset, gpointer* output,
13                 guint64* outlength, guint64* outoffset,
14                 JTransformationCaller caller)
15
16 void
17 j_transformation_cleanup(JTransformation* trafo, gpointer data,
18                 guint64 length, guint64 offset,
19                 JTransformationCaller caller)
20
21 gboolean
22 j_transformation_need_whole_object(JTransformation*,
23         JTransformationCaller);
```

Listing 5.2: The important parts of the JTransformation interface

5.2 The JTransformation type

The `JTransformation` type is the only part of our code that we had to explicitly add to the JULEA `core`. This requirement arose because of the server side transformation mode. In this mode the transformations and inverse-transformations need to be handled by the backend, which therefore needs to know about the `JTransformation` type. The only other way to solve this problem would have been to make the whole JULEA backend depend on the transformation client which seemed much more problematic.

In Listing 5.2 we can see a summary of the most important parts of the `JTransformation` interface. As we have already discussed in the previous chapter a transformation needs attributes for the transformation type and mode. These need to be set at creation time and will govern what transformation algorithms are used by the object and whether the transformation takes place on client- or server-side. Depending on the used transformation it can also be determined whether partial read and write operations on the object need to retransform the whole object data every time or if parts of the data can be retransformed correctly on their own. This behaviour is indicated by the `partial_access` boolean in line 7. If we take a look at lines 10-19 we can find the functions that handle the actual data transformation.

- **The apply function**: This function is the core part of the `JTransformation` type. It takes a pointer to the data and some length and offset information and returns the transformed data in the `output` pointer together with the maybe

modified length and offsets values. Whether the apply function should transform original data or inverse-transform already transformed data back to the original is decided by the `caller` argument. This enum lets the user specify what kind of transformation operation should be performed as for example a read (where the data would be inverse-transformed to its original state) or a write (where the transformation algorithm is applied in the original data to yield the for example compressed data).

The `apply` function is very general by design so that it should support any kind of actual transformation algorithm that could be applied to data. The specific transformation type is given implicitly with the pointer to a `JTransformation` and the implementation of the function will then redirect to the corresponding transform and inverse-transform function for the used algorithm. We implemented it that way to make it easier for future users to add implementations of their own transformation algorithms. To add support for a new transformation type it has to be added as an option to the `JTransformationType` enum and transformation and inverse-transformation functions have to be added to the `JTransformation` implementation. This process is very straightforward and needs to be done in the `j_transformation_apply` function of `JTransformation`. Here the transformation call is handed over to the appropriate function, determined by a switch statement over the transformation type the current object is using. The user simply has to add a new case for the added transformation type and provide one function, that applies the new transformation algorithm and returns the transformed data in an output buffer, and also provide one function that does the same for the inverse transformation.

- **The cleanup function**: This function is necessary to deallocate the transformation data buffer that is created by the `apply` function. This buffer is necessary for both read and write operations. In the case of a write operation the transformation client needs to take the data from the provided data pointer and apply the transformation algorithm to it, thereby modifying the original data. This modification can of course not be done in-place since this would invalidate the data buffer provided by the user by replacing the original data with the transformed data. Therefore the `apply` function needs to allocate a new buffer for the transformed data which is then written to disk by the backend. If this buffer is not freed at some point JULEA would start to leak memory for every operation on a transformation object. In the case of a read operation a new buffer is also necessary in some cases since we might need to read the whole object and retransform it before being able to provide the result of a partial read request from the user. The buffer that holds the complete object would therefore in many cases be larger than the provided read buffer from the user which means that in this case also the transformation client needs to create an intermediate buffer that needs to be freed at some point.

  It should be noted that the use of this extra buffer for read operations is only necessary in the case of transformations that do not support partial reads. If the

```
1  struct JTransformationObject
2  {
3      guint32 index;
4      gchar* namespace;
5      gchar* name;
6      gint ref_count;
7
8      JTransformation* transformation;
9
10     JKV* metadata;
11
12     guint64 original_size;
13
14     guint64 transformed_size;
15 };
16
17 void
18 j_transformation_object_create(JTransformationObject* object, JBatch*
       ↪ batch,
19         JTransformationType type, JTransformationMode mode)
20
21 void
22 j_transformation_object_status_ext(JTransformationObject* object,
23         gint64* modification_time, guint64* original_size,
24         guint64* transformed_size, JTransformationType*
              ↪ transformation_type,
25         JBatch* batch)
```

Listing 5.3: The important parts of the JTransformationObject interface

transformation can be done on any arbitrary subset of the data, no additional buffer that needs to temporarily store the whole object is necessary. In that case the cleanup function has no work to do and simply returns immediately.

Those free operations can not always be done directly in the write or read function of the transformation object because the pointers might need to be valid until after the function end. Fortunately the JULEA object type already provided `_read_free` and `_write_free` functions where cleanup for operations on the object can be done. In our implementation of the `JTransformationObject` we added calls to this cleanup function to them.

The `JTransformation` type is not really used by itself in the transformation client but it will be a central part of the `JTransformationObject` implementation which we will take a look at now.

5.3 The JTransformationObject type

The `JTransformationObject` is the core type of our transformation client. It behaves like the traditional `JObject` from JULEA with the exception that the data is stored transformed by the backend. In Listing 5.1 we have already seen an example of how it can be used by an application. In this section we will go more into detail on how the transformation operations are integrated into write and read operations on the object and how the key-value store is used to store the metadata. In Listing 5.3 we show the important attributes and interface functions of the type.

The first four attributes of `JTransformationObject` are identical to `JObject` and behave the same way. Following them we can see that the object contains a reference to an instance of the previously described `JTransformation` type. This attribute contains all necessary information about the type and mode of transformation which are set for this object. It is either set at creation time of the object or in the case of an application acquiring a reference to an already existing transformation object the transformation information will be loaded from the key-value store and the transformation attribute is set. The following `metadata` attribute is a reference to a JULEA key-value store entry which is used to always load and store the most recent transformation object metadata. In general before each operation on the object the key-value store is checked whether it contains newer metadata, which will then be loaded and after each operation that modifies the object's metadata, the new data is stored again in the key-value store. The association between the current object and its corresponding metadata object is handled by concatenating the objects namespace and name which will yield the name of the key-value store entry for its metadata.

The last two attributes are variables for the object's original and transformed size. Original size in this case means the size of the user data in its normal form and the transformed size indicates the size of the transformed data stored by the backend. As we have already discussed it depends on the transformation type whether and how these values might differ from each other.

Below the `JTransformationObject` declaration we have highlighted the two interface functions that differ in their signatures from the interface of the normal `JObject`. First is the `_create` function which we have already mentioned in our previous example. It contains two additional attributes for the transformation type and mode that need to be declared at the creation of a transformation object. The `_status_ext` function is an extension to the regular status function which will deliver additional information that concerns transformation objects like the original and transformed sizes and the used transformation type and mode. The regular status function still exists for the `JTransformationObject` and will return the same information as for `JObject`.

We have now seen the differences between `JTransformationObject` and `JObject` and how to use it. In the rest of this section we will take a closer look at the write and read operations on a transformation object and how their workflow in our implementation using different transformation type and mode configurations. We will first discuss the client-side and then go into the server-side transformation mode.

5.3.1 The client-side read operation

For a read operation on a transformation object some preparation steps and decisions have to be made before any actual data can be requested from the backend. We will now summarize the workflow of our implementation.

1. **Load current metadata**: The first step of each read operation is to request the object's current metadata from the key-value store and to update the corresponding attributes in the local object reference. This is necessary if we want to allow multiple clients to access the same object. The object might have been modified by another client since the last operation of the current client on the object and therefore we can never be sure that our local version of the metadata is the most recent one.

2. **Check which transformation mode is set**: In the next step we need to check the set transformation mode of the object to find out if the client should even do any transformation work or if it should just forward the read request to the server.

3. **Check if transformation type allows partial reads**: We now need to find out whether the transformation type of our object allows partial reads. If this is the case and our read operations just need a part of the objects data we can just go ahead and make this request to the server followed by inverse-transforming the received data and returning it to the user. In the following we will discuss the more complicated case where partial reads are not possible and the whole object data is needed to perform the retransformation of the data.

4. **Perform the read and inverse-transformation**: If we assume that partial reads are not possible we first need to make a request to the backend to read the whole object into a temporary buffer. From the object's metadata we know its transformed size and use it to modify the original read request from the user to a read of all of the transformed data. Now we can hand a reference to the transformed data to the previously described `transformation_apply` function of `JTransformation` and receive a reference to the inverse-transformed data back. Now that we have the object data in its original form we just need to copy the requested part of the data into the read buffer that was provided by the user and the read operation has been performed correctly. The only thing that remains is to make sure to free all allocated temporary buffers.

As we can see there might be a significant overhead for read operations on transformation object that can not support partial reads since we always need to read the whole object from the backend. Our solution to reducing the performance impact of this problem is to not always transform the complete object data but to divide it into multiple chunks that are all transformed individually. We will discuss this closer in Section 5.4.

5.3.2 The client-side write operation

Write operations can get a bit more complicated then read operations in the case of no partial writes. Like in the last section we will now give a summary of the necessary steps to perform a write on a `JTransformationObject`.

1. **Load current metadata**: Same as before we again have to refresh our metadata before performing this operation

2. **Check which transformation mode is set**: In this step we again have to check if the client is responsible for the transformation or not.

3. **Check if transformation type allows partial writes**: Also same as before if partial writes are allowed the only thing that has to be done is applying the transformation to the user write data and then writing the resulting transformed data to the backend. We will again go on under the assumption that we have a partial write request and the selected transformation type does not support partial writes. If partial writes are supported by the used transformation, the next two steps can be skipped and the write data can immediately be transformed and written to the backend.

4. **Read the whole object**: If partial writes can not be done we first need to read the whole object to later add the new data and transform everything again before doing the actual write to the backend. This is done by simply calling the transformation object's read function on all transformed data. This is again stored in a temporary buffer. Doing it this way already gives us the data in its original form because the inverse-transformation is performed by the read function.

5. **Add new write data and perform the write**: Now that we have the original data we can insert or add the data of the write operation and again apply the transformation to the whole buffer and actual write to the backend can now finally take place. Again all temporary buffers need to be freed to not introduce memory leaks.

6. **Update metadata**: Since a write is a modifying operation the metadata of the object will have changed and it needs to be updated and written to the key-value store. Listing 5.4 shows the metadata struct of a transformation object. It contains the basic type and mode information about the transformation object and the original and transformed size of the object's data. The first two fields are never updated by a write operations since they need to be specified at the object creation and can not change after that. The two size parameters will have changed after the execution of a write operation and therefore always need to be updated for future operations on the object to work correctly.

Even more than for the read operation this write implementation can come with some severe overhead. If partial writes are not possible each write operations brings with it a read of the whole object.

```
1 struct JTransformationObjectMetadata
2 {
3     gint32 transformation_type;
4     gint32 transformation_mode;
5     guint64 original_size;
6     guint64 transformed_size;
7 };
```

Listing 5.4: The metadata struct for a transformation object

5.3.3 Server-side operations

The general workflow in which order reads, writes, transformations and inverse-transformations have to take place does not change for the server-side operations but we will give a short summary of how and where we implemented them.

First of all we added new `J_MESSAGE` entries for transformation object operations to the JULEA server. To be able to perform transformations the server needs to have access to an object's metadata. If we wanted to solve this the same way as the client-side transformation this would mean that the server would need to connect and depend on the key-value client. This did not seem like a good option and therefore we decided on just sending the required information with the read or write JULEA messages. Therefore we added the `JTransformation`, `original_size` and `transformed_size` attributes to each message. In the first step of parsing a received message the server will check the transformation mode of the sent `JTransformation`. If the object uses client mode no further action has to be done by the server and the code of the regular `JObject` is run. If on the other hand the object works in server mode we call modified version of the `j_backend` write or read functions. We added those functions to the backend code and they pretty much implement the workflow for reads and writes described in the last two sections.

The last thing that needs to be solved was to update the objects metadata after a server-side write operation. Since we decided not to add a key-value client dependency to the server the metadata updates can not take place here. We instead added the necessary changes in metadata to the server's reply messages and handle the metadata update on the client upon receiving the reply.

5.4 The JChunkedTransformationObject type

We have now seen our basic implementation of a transformation object but as we have discussed major overhead can occur for partial read and write operations in specific situations. If we think about a client application that for example performs a lot of small write operations on a big object it is not really feasible to have complete reads and writes on the whole object occur every time. Therefore we have to find a solution for this problem and at least reduce the amount of overhead for operations on the transformation object. This is what the `JChunkedTransformationObject` type tries to do.

```
1  struct JChunkedTransformationObject
2  {
3      guint32 index;
4      gchar* namespace;
5      gchar* name;
6      gint ref_count;
7
8      JTransformationType transformation_type;
9
10     JTransformationMode transformation_mode;
11
12     JKV* metadata;
13
14     guint64 chunk_count;
15
16     guint64 chunk_size;
17 };
18
19 void
20 j_chunked_transformation_object_create(JChunkedTransformationObject*
       ↪ object,
21         JBatch* batch, JTransformationType type,
22         JTransformationMode mode, guint64 chunk_size)
23
24 void
25 j_chunked_transformation_object_status_ext(JChunkedTransformationObject*
       ↪ object,
26         gint64* modification_time, guint64* original_size,
27         guint64* transformed_size, JTransformationType*
               ↪ transformation_type,
28         guint64* chunk_count, guint64* chunk_size, JBatch* batch)
```

Listing 5.5: The important parts of the JChunkedTransformationObject interface

Instead of transforming the whole object data with one application of the transformation algorithm we decided to introduce chunking to the data. The general idea is that the object consists of multiple fixed size chunks of transformed data. If a small partial read or write operation now occurs it is sufficient to just read and inverse-transform the data of one or a few chunks to correctly perform the operation and we do not need to read the whole object.

In Listing 5.5 the most important parts of the `JChunkedTransformationObject` interface are shown. Most of the attributes are identical to the ones already explained for the `JTransformationObject`. Of most interest should be the last two, `chunk_count` and `chunk_size`. The chunk count indicates how many chunks are currently part of the object and the chunk size is the attribute that determines the original data size of a chunk. We decided on performing the chunking on the original data instead of the transformed data. This may lead to the transformed chunks not having identical sizes but makes it much easier to handle the chunking and later offset and size calculations

for write and read operations. From a file system perspective it probably would be more efficient to have the resulting transformed chunks be a specific and constant size but this is a difficult task for transformations algorithms that compress the original data because you would need a way to accurately predict how much of the original data compresses to the set chunk size or you would have many transformed chunks that need empty padding bits at the end.

We again highlighted the `_create` and `_status` functions as the only functions that have a modified signature from their `JObject` versions. The create function now has the additional `chunk_size` parameter where the user has to set the wanted chunking size. The `_status_ext` function now also return information about the chunk count and size.

The implementation of `JChunkedTransformationObject` is quite straight forward. Since we already have the transformation logic implemented in `JTransformationObject` we decided to build the chunked object around those objects. Each chunk consists of an own instance of a `JTransformationObject`. The main task of the chunked transformation object is managing its chunks and to translate the offset and length values of incoming read and write operations to individual operations on the matching chunks.

The metadata consists of the transformation type and mode and the chunk count and size, which are again stored in the key-value store. The chunk objects are automatically created if the size of the chunked object grows and they are named in the form of: `<object namespace>_<object name>_<chunk index>`. Going with this pattern it is quite trivial to compute and identify the necessary chunk object for an incoming read or write operation. The operations are then simply passed along to the correct chunks and the resulting data for a read operation is assembled in the read buffer or it is split and and individually sent to the corresponding chunks for a write operation.

This concludes the description of our implementation of the transformation client. We will now take a look at some benchmark results and discuss which of the challenges from Chapter 4 we managed to solve and what future work is still open to improve the client.

6 Results and Benchmarks

In this chapter we will discuss a few benchmarks for our transformation client. We will compare the write and read performance to the normal `JObject` to find out how much the overhead cost the transformation of the data and the additional metadata management tasks bring with them. We will also briefly compare the performance of the different transformation algorithms provided by the transformation client and compare transformations with and without partial read or write access.

6.1 Comparing JChunkedTransformationObject to JObject

For this benchmark we wanted to see a comparison between the normal `JObject` and our `JChunkedTransformationObject` in regard to the write and read performance for larger objects with many operations. All benchmarks were run on a home desktop machine with the POSIX backend of JULEA writing into a tmpfs.

When comparing our `JChunkedTransformationObject` to the regular `JObject` we have to take into account that the chunking operations will definitely add an overhead that impacts the performance of our implementation negatively. It could therefore be argued that a comparison to JULEA's distributed object instead would make sense. However our implementation does not use chunking to spread the object over multiple servers but instead uses it to increase the performance of partial write and read operations. It does therefore serve a different purpose and we thought it more appropriate to compare it with the normal `JObject`.

For the benchmark results that we can see in Table 6.1 the application writes in total 1GiB of data to the object in multiple small 1MiB write operations on the objects. In this case each write or read operation was executed in its own batch to simulate an application that has a high frequency of independent operations. The variable parameter in these benchmarks was the chosen chunk size for the `JChunkedTransformationObject` or alternatively the use of the standard JULEA `JObject`. The chunked transformation object is configured with using the LZ4 transformation type and the client transformation mode.

We also did two separate runs of the benchmarks with different kinds of data. In tables (a) and (c) we used a block of random data which leads to a bad compression result for LZ4 that in many cases even increases the data size by a bit. This could be seen as the worst case for a transformation object. In tables (b) and (d) we used a block of 'fixed' data where all bytes where set to the same constant value. This leads to an enormous compression rate for the LZ4 algorithm and can be seen as a best case scenario.

We only display the measured runtime of the write or read operations here and did not calculate the write/read data rates here because this benchmark is mainly focused on the comparison between the two object types and also the actual write and read speed would be highly dependent on the used hardware, backend and filesystem and would therefore not be very meaningful here. All shown results are the average values of three separate benchmark runs.

chunk size	time
4KiB	105.6s
1MiB	2.3s
JObject	0.68s

(a) Writes(random data)

chunk size	time
4KiB	119.1s
1MiB	0.91s
JObject	0.77s

(b) Writes(fixed data)

chunk size	time
4KiB	134.6s
1MiB	2.1s
JObject	0.62s

(c) Reads(random data)

chunk size	time
4KiB	128.2s
1MiB	1.10s
JObject	0.56s

(d) Reads(fixed data)

Table 6.1: Read/Write benchmarks for JChunkedTransformationObject vs. JObject

If we take a look at the results shown in Table 6.1 we can immediately see that the chunk size is a huge factor in the performance of the `JChunkedTransformationObject`. The first data point of each table used a chunk size of 4KiB which means that there will be 250000 chunks created for the 1GiB of total data and 250 new chunks for every 1MiB write operation. We can clearly see that the overhead of managing that many chunks has a huge impact on performance and makes it not really usable in practice. This was at first a very demotivating result but if we take a look at the second rows in the tables we can see that increasing the chunk size to 1MiB gave us a much more practical result. The performance is still distinctly worse than the regular `JObject` but only by a factor that a user might be alright with if the data compression aspect would be important for the application.

We will now take a closer look at the differences between write and read operations on the objects. The results show quite clearly that the time for read operations with small chunk sizes exceed the times for write operations. This makes sense if we think about how the read operation for chunked transformation objects is implemented. It first has to send read request to all affected chunks and then needs to copy all small data chunks into the read buffer before it can be returned to the user. In the second case of using 1MiB chunks this is not a factor anymore because only one chunk has to be read and copied per read operations since the read size and chunk size are equal here.

The second thing to compare are the effects of the used data. As already mentioned the random data will have a very low or even negative compression factor and the fixed

data will compress exceptionally good. If we compare tables (a) and (c) to tables (b) and (d) we can see a clear improvement of read and write speeds for the 1MiB chunk size case for the fixed data benchmarks. This is quite interesting and might show the actual impact of running the transformation algorithm. For the random data we have about a factor of 3 for the runtimes compared to the `JObject` run but for the fixed data ones this factor goes down to under 2. We will try to measure the impact of the used transformation algorithm in the next benchmark.

Overall we can conclude from this experiment that the chosen chunk size for the chunked transformation object plays a huge role in its performance. This however also leads to the conclusion that the chosen chunk size has to be highly tuned towards the size of the write and read operations occurring in the applications using this object. We suspect that this would be hard to realize for applications with widely different I/O patterns on the same object and would probably need significant fine tuning of the chunk size parameter to achieve the best possible performance.

6.2 Partial writes vs. non partial writes benchmark

In this benchmark we were interested in comparing the performance of the basic `JTransformationObject` using different types of transformations. As we have discussed before the biggest difference with transformation types is whether they allow partial writes and reads. If the transformation does not allow partial writes or reads it is always necessary to first inverse-transform the whole object before writing or reading on it. In this benchmark we compared `JObject`'s write performance using LZ4 and XOR as transformation algorithms. For a baseline we again did the same writes to a standard `JObject`. Since `JTransformationObject` performs quite bad for many successive writes we chose a rather small data size for this benchmark and only wrote 200MiB in 1MiB increments. As before we did two benchmark runs, one with random and one with 'fixed' data blocks. The testing conditions were similar to the previous benchmark and done on the same machine. Again all measurements were repeated three times and the average values are shown in Table 6.2.

transformation	time
LZ4	61.9s
XOR	0.28s
JObject	0.09s

(a) 200 1MiB writes (random data)

transformation	time
LZ4	26.8s
XOR	0.26s
JObject	0.09s

(b) 200 1MiB writes (fixed data)

Table 6.2: Comparing LZ4 and XOR transformations on JTransformationObject with normal JObject writes

The first thing to notice is that `JTransformationObject` performs very badly for many successive writes. This result was of course expected and is the reason why we imple-

mented the chunked transformation object. Since every write is preceded by a read and inverse-transformation of the whole object a massive overhead is created. If we however switch the transformation type from LZ4 to XOR the object can now support partial writes and a lot of this overhead goes away. The results are still worse than the standard `JObject` but the difference we can see here is much more bearable. The overhead that is left here is of course the XOR transformation itself and additionally the metadata management for the object. If we compare the random data to the fixed data benchmark we can see that the fixed data results are only much better for the LZ4 case. In our opinion this most likely stems from the fact that the size of the transformed data is very small in this case and therefore a lot less data has to be read from the backend before each write operation. Of course the LZ4 algorithm should also have a larger runtime than the XOR algorithm but since we are only dealing with a small amount of data here and LZ4 is a quite fast compression algorithm we don't think that this has much impact on the results here.

In conclusion these results confirm our line of reasoning of why a chunked transformation object would be necessary to achieve reasonable performance for transformation algorithms without partial writes/reads. We can also see that transformations that allow partial access only add a reasonable amount of overhead but on the other hand these kinds of transformations are probably not very useful in practice. We do not think that there exist many real world examples where such a transformation would be desirable but it should not be harmful to offer an interface for such algorithms anyway in our implementation.

7 Future Work

In this project we created a working transformation client for JULEA but we could not solve all problems completely and some possible future work and optimizations are left open.

Concurrent access:
In Section 4.4 we discussed the problem of allowing concurrent access to a transformation object by multiple clients. As we already discussed there this would require some kind of locking mechanism in JULEA which is currently not available. Since this problem can also occur for other parts of JULEA we decided on focusing our work elsewhere first and this task is still open for the future.

More transformation algorithms:
Our client currently supports LZ4, RLE and XOR transformations. This repertoire could be extended by for example many other existing compression algorithms. We tried to design the client as extendable as possible and adding a new transformation type can be accomplished quite easily by simply adding the transformation and inverse transformation functions to the `JTransformation` type. Since this project was more about creating the infrastructure for this client we did not see much use in adding all kinds of different compression algorithms ourselves but we always kept it in mind during development and adding new algorithms in the future should not be very hard.

Performance:
Performance optimizations for the `JChunkedTransformationObject` are the biggest open problems. As we have seen in Chapter 6 we can get to a reasonable performance level if the chunk size parameter is tuned correctly for the occurring write and read operations on the object, but if this is not the case the performance can be quite bad. Of course a transformation object will never be as fast as the normal `JObject` because of the always existing overhead, but it should still be possible to get closer to `JObject`'s performance. The problem of always having to read and inverse-transform a whole compressed chunk before performing an operation on its data seems non avoidable but as we have seen the biggest performance degradations occur when the amount of chunks gets large. It is probably still possible to optimize the read and write processes of the chunked transformation object a bit more and also find ways to decrease the current metadata management overhead.

8 Conclusion

In this report we discussed our project of adding a transformation client to JULEA. As we have seen we managed to create a working proof-of-concept implementation that supports multiple different kinds of transformations. The user interface for our client is very similar to the already existing object client and should be easy to use for everybody that is familiar with JULEA.

In Chapter 7 we discussed some of the problems and optimizations that are still left open. Our main concern here lies with the performance. As we have seen a user should always use the `JChunkedTransformationObject` over the `JTransformationObject` if the transformation algorithm does not support partial reads and writes. To get a reasonable performance the chunk size has to be tuned to the occurring I/O patterns of the applications using the object. If this can be done our client performs reasonably well. However we think that there are still some optimization opportunities in the code base, especially in the management of chunked transformation objects with a large number of chunks.

Overall we can say that we implemented a working proof-of-concept for the transformation client which could be used in the future to explore the application of data transformations or more specifically compression algorithms for file systems.

Bibliography

[GLC+15] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack package manager: bringing order to HPC software chaos. In *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*, pages 1–12, Los Alamitos, CA, USA, nov 2015. IEEE Computer Society.

[Kuh17] Michael Kuhn. JULEA: A Flexible Storage Framework for HPC. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, pages 712–723, Cham, 2017. Springer International Publishing.

[Yan19] Yann Collet. LZ4 - Extremely fast compression. `https://github.com/lz4/lz4`, 2019. (accessed 09/2020).

Appendices

List of Figures

List of Tables

List of Listings