Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

# System profiling and data aggregation for smart compression in Lustre

Pablo Correa Gómez

MIN-Fakultät

Fachbereich Informatik

Arbeitsbereich Wissenschaftliches Rechnen

Program of Study: Intelligent Adaptive Systems

Matrikelnummer: 7251797

Date: 17.08.2022

First Examiner: Jun.-Prof. Dr. Michael Kuhn

Second Examiner: Prof. Dr. Thomas Ludwig

Supervisors: Anna Fuchs, Jun.-Prof. Dr. Michael Kuhn

# Acknowledgements

This work would have been in a much different shape without the constant laughs and support from Till. And might have never been finished without all love and care that Juli put into me when I was not able to do it myself. This thesis is specially dedicated to them.

But the thesis is also a culmination of years of study and a continuous shift to a new career path. This would not have been possible without my family, Jorge, Guille, and Rodri, which were always there through the years, the ups, and the downs.

Indirectly, thank you to Georgia, the ultimate reason for my move to Hamburg. And to Anton, Till, Pia, Rohan, and Stefi, which unknowingly convinced me to stay. Also to Annika and Sven, which provided the most useful advice and consultation when the doubts were at its peak. Finally, thank you to Simon and everybody at Combine which approved the remote work before it was a thing, and allowed me to sustain myself for the whole of my thesis.

# Abstract

In High-performance computing (HPC) setups, the IO and data transfer can be a big part of the processing requirements of scientific applications. When that is the case and they become a bottleneck, the application performance can degrade. This problem is expected to become more common since CPU processing has been for many years and continues growing at a faster rate than network or storage speed. Moreover, the imbalance between different machines with different roles in the setup and applications' inefficiencies make this problem worse.

In this thesis, compression is considered a solution to this problem. Compression allows trading the excess in computation power for a reduction in the data size, both for IO and transfer. However, static compression can potentially result in a similar set of inefficiencies as those that it aims to solve. For this reason, I propose to extract and analyze information from the HPC setup, introducing a collection and decision-making process that makes compression *smart*. The integration point for compression is the parallel filesystem, which is the piece of software that, in HPC, takes care of the data transfer. For this work, Lustre, the most popular filesystem among big HPC deployments [57], is the filesystem of choice.

In consequence, this thesis analyses a typical Lustre setup to identify and extract the components that would take part in the smart compression. Those components are studied to obtain the metrics relevant for compression. Later, the required process for smart compression is considered, and two relevant decisions, the location of the compression, and the algorithm configuration to use are analyzed in detail. The analysis assesses the key metrics for each decision and possible ways to integrate their calculation. Finally, to prove the relevance of smart compression, a small set of experiments show both the benefits of compression and the dangers of a wrong configuration.

# Contents

# 1. Introduction

High-performance computing (HPC) is a field of computer science that aims to solve advanced computing problems. HPC setups are designed to accommodate applications that solve complicated scientific tasks, such as particle physics simulations or molecular chemical interactions. For this reason, they are usually designed differently than regular computer systems. Instead of placing together all the system components (CPU, GPUs, memory, and storage) in generic machines, HPC setups split them into specialized nodes. Therefore, a great amount of computational power (CPUs, memory, and GPUs) is aggregated together into *computation* or *client* nodes. These nodes are usually interconnected and placed together, allowing a great communication bandwidth between them. This makes it possible for scientific applications to run efficiently split among multiple nodes. However, computation nodes usually do not have storage or their storage is very limited. Instead, specialized *server* nodes host all the storage. These nodes often have more limited computational capabilities but support great amounts of storage per node. The clients and servers are connected through a dedicated network which allows for the data flow. Therefore, when clients need to write some data (be it for persistent storage, for it not fitting in their memory, or for any other reason), the data is transferred through the network to the servers. When clients later need some data from the servers, they request it and the data is sent back through the network. The system consisting of the clients, servers, network, and necessary auxiliary machines is usually referred to as a *cluster*.

In HPC clusters, the piece of software which takes responsibility for the data transfer between clients, servers, and storage is usually a parallel filesystem. The filesystems are those pieces of software in computer systems that take care of the data storage and access. They are the ones that write the data to disk, keep a record of the location for different pieces of data, and, in general, allow transparent access to files and directories in a computer system. A parallel filesystem is a filesystem that can be mounted and accessed concurrently by multiple machines, and its storage is possibly distributed in different ones. Therefore, they are a perfect fit for typical HPC clusters. Out of those parallel filesystems available, Lustre is the most popular one among big clusters [57] and the system of study in this thesis.

However, the work of the parallel filesystems is often limited by the network interconnection between clients and servers, or by storage operations. When these become a bottleneck, application performance can degrade. To overcome this issue, compression is seen as a solution. Data compression consists in using computation capabilities to reduce the size needed to express certain data. Therefore, at the cost of computational resources, the bottlenecks could be reduced. This feature is already available in Lustre through one of its backends, but only applies to the servers and its configuration is static, losing efficiency. In consequence, applying compression to Lustre is relevant in the scientific space for different reasons:

- First, while most of the components involved in High-performance computing have been steadily growing exponentially over the last 50 years [47, 36] the growth speed has been different for different components. Such difference creates a performance gap between CPUs and other components. Compression can help leverage the gap by using CPU resources to reduce the requirements on the other elements.

- Second, in HPC clusters there is usually a big imbalance between the power and configuration of servers and clients. Servers have the purpose of receiving data and storing it. Therefore, they are usually specialized machines with a big capacity for IO and memory for caching, but less proportional CPU power. Clients, on the other side, are designed to run scientific computations and have usually a lot more CPU power than servers and enough memory. The number of clients and servers is also usually not even, but there are a lot more clients than servers, which act as a shared resource. With these ideas in mind, compression in the clients seems more appropriate, considering that additional network resources could be saved. Some previous engineering work [29] already considered this approach.

- Last, it is still common that scientific applications which run in HPC setups are often making poor usage of resources. Some possible hypotheses for this situation are the complexity of the programs, the fact that they are mostly developed by non-computer-scientists, and the complexity of the optimizations. Therefore, by obtaining system information and taking decisions that reduce the processing time and increase the resource usage, the results from this thesis can help those tools improve their overall performance without updating them.

However, just compressing is not enough. Static compression configuration like the one now available and in development [29] in Lustre requires the intervention of system administrators and lacks flexibility. Depending on the workload of different applications and properties of the data they write, compression can have different impacts, even potentially degrade the performance. For example, in the case of a very well-optimized application that uses all client resources and has no network bottleneck, compression in the client could potentially slow it down.

Therefore, this thesis aims to make compression *smart*. To do so, it will aggregate system metrics and data properties with user-provided hints and real-world knowledge about the algorithms being used, and take decisions accordingly. This should allow obtaining the maximum possible benefits from compression while keeping the flexibility to avoid it when it is of no help. The system metrics and data properties, together with information about compression algorithms, are used to predict the best-fitting compression to apply. The user-provided hints are integrated to allow users to provide information about their preferences, the data, or its future usage, which would be otherwise lacking. Altogether, this work fits into a bigger effort to improve the compression status in Lustre to increase the efficiency of scientific applications running in HPC clusters.

Additionally, this thesis is unconventional for the space it covers within the scientific effort of the Computer Science field. In general, the field of Computer Science provides scientists with the possibility of easily executing manipulation experiments. In an already built Computer Science system, or when a multitude of those exist, computer scientists can edit or modify single variables and reach causal explanations. This is, however, unfeasible to do in this thesis for two reasons:

- On one hand, the complexity of the system of study is much higher than any regular Computer Science setup. Lustre is designed to be a core part of High-performance computing setups and it is being used in a majority of the 100 fastest computers in the world [57]. It should be clear, that accessing and doing manipulation experiments on setups of such magnitude is a problem. Both for the lack of resources and the complexity of the changes.
- On the other hand, there is a big lack of previous or related work on compression for HPC setups. In consequence, the necessary bits to build on to reach the ability to execute manipulation experiments are missing. At this moment, there exists no direct and already published previous scientific work in this field. And although some previous engineering work [29] to build upon exists, it is not ready to be deployed and experimented with.

A visual representation of the different locations in science between this work and most Computer Science thesis is presented in Figure 1.1. The red dot represents the location of this thesis while the blue dots represent common Computer Science experiments. This thesis is a system-specific and mostly analysis and description-based work, while typical manipulation-based experiments from Computer Science represent causal explanations of different degrees of generalization.

In consequence, this work mostly aims to do an exploratory analysis and yield hypotheses, which future authors can build upon. For that reason, this thesis is organized as follows: First, in Chapter 2, some necessary background about the tools and technologies used in this thesis will be presented. Then, Chapter 3 contains a discussion of the components involved in the compression workflow of interest for this thesis and the information which can be extracted from them. Afterwards, Chapter 4 uses the knowledge gathered in 3 to deepen the discussion into the decision-making process, its requirements, needs, and possible solutions. It is this chapter that contains most of the yielded hypotheses, assumptions, and requirements for future work on this topic. Chapter 4 is the core of the theoretical discussion happening in this thesis, supported by some small experiments and the previous work in Chapter 3. After the theoretical outcome of the thesis is finished, Chapter 5 presents a small exploratory setup where some knowledge about the hypotheses is gathered and the behavior of a very specific system is studied. It provides the necessary practical foundations for later work to improve and iterate the previous discussions. It also contains not only a small set of experiments to challenge the most basic ideas from Chapter 4 but also some discussion on the findings and technical limitations of the setup of interest in this thesis. Finally, this thesis concludes with Chapter 6, a discussion of possible future work that can build on the hypotheses proposed and the knowledge gathered in this thesis, and also includes a small conclusion section.

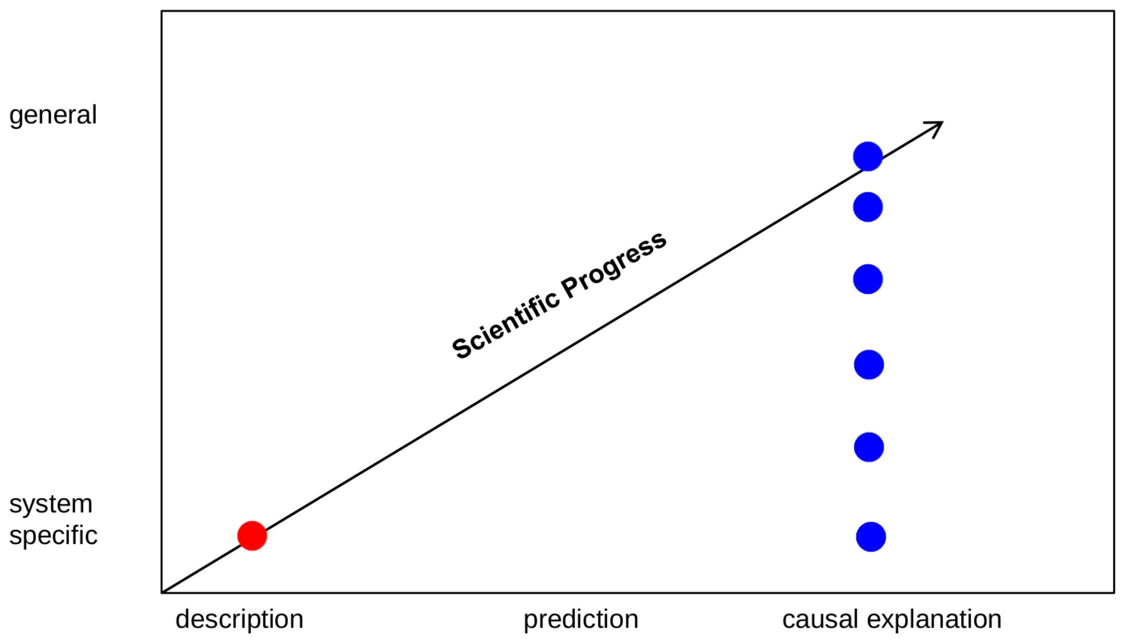


Figure 1.1.: The scientific progress[Sven Magg, Research Methods Lecture]. The red dot represents the location of this thesis; the blue dots, typical Computer Science experiments.

# 2. Background

For a better understanding of the concepts and the work developed in this thesis, it is important to provide the reader with some necessary background of the tools and technologies that will be discussed. For that reason, in this chapter, I introduce Lustre, the filesystem central to the development of this thesis. Afterwards, I will discuss compression in Computer Science, and in most detail some of the algorithms which will take part in the core discussions of this thesis. Hooking up both concepts, an introduction to the current status of compression in filesystems will be discussed. Later, I will briefly introduce the Linux kernel, since it is a key component for the metric collection discussed and executed during this thesis. Finally, I will conclude with an overview of related work that might be useful as motivation for this thesis, as well as for follow-up works.

## 2.1. Lustre

Lustre is a parallel filesystem commonly used in High-performance computing setups. Its architecture consists of clients (usually HPC nodes), management and metadata servers (MGSs and MDSs), and object storage servers (OSSs) hooked up through some network that might contain Lustre routers. The purpose of the whole system is to allow computational nodes (clients), which usually lack storage, to store the data they produce. OSSs serve that purpose by receiving the data from the clients and storing it persistently. Then, they can provide the data back to the clients when requested. The other components (MGSs, MDSs, and routers) serve the purpose of the management, organization, and correctness of the communication. To understand Lustre's architecture and common deployments, it is important to further describe these components:

- **Lustre clients**: the clients are the data generators and consumers. They are most often computations nodes without any kind of storage, but can also be desktop machines or visualization nodes that simply mount a Lustre filesystem. They usually have very limited storage, compared to the Terabytes to Exabytes of any usual Lustre deployment.
- **Lustre Management Server**: the Management Server (MGS) stores the configuration for the Lustre filesystem, providing that information to all the nodes in the cluster. At each point in time, it is always a unique machine. To avoid a single-point-of-failure for the whole cluster, it can be configured with replication to a standby instance. Such an instance will get active in case of failure of the main instance and is common in high-availability setups. The persistent data is stored in the Management Target (MGT).

- **Lustre Metadata Servers and Targets**: the Metadata Servers (MDS) manage the metadata and information about all the files and directories in the filesystem. That information is stored in Metadata Targets (MGT) which are the persistent storage backend for the MDSs and might be accessed by multiple MDSs. The MDSs, in consequence, make the information in the MDTs available to all the nodes of the cluster through the network.
- **Lustre Object Storage Servers and Targets**: the Object Storage Servers (OSS) manage the network access for I/O operations on the Object Storage Targets (OST). These OSTs are the final storage for the user data. The user data can be stored in one or more objects and replicated across different OSTs. These settings are user-configurable to tune performance and reliability. Both the OSSs and the OSTs support replication to increase the reliability and availability of the system.
- **Lustre routers**: on big-scale deployments, traffic might flow through different network hardware types, e.g: both Ethernet and Infiniband, and routing might be necessary given the high amount of components. Lustre routers take care of both doing the necessary routing between the different components, and managing routing and traffic transformation needed to interconnect networks built from different network technologies.

Figure 2.1 represents graphically all the components and their interconnections of an example scalable Lustre deployment. In such a cluster all the components already discussed can be identified: clients, MGSs, MGT, MDSs, MDT, Lustre routers, OSSs, and OSTs. Additionally, it is visible that both the MGS and the MDS support a fallback replica, while some of the OSSs also support replication. There are also a variety of OSTs with different replication and fallback configurations, and two different network technologies interconnected through Lustre routers. This should allow the reader to get an idea of the complexity of a traditional Lustre setup.

### 2.1.1. OSS backends

Lustre's Object Storage Servers are of special interest in this thesis, since they hold the data which will be compressed, and need to be able to *understand* it. However, the OSSs do not directly manage the storage. Instead, they attach to one of two different backends:

- **ldiskfs**: an ext4 fork with custom modifications and improvements for Lustre's workflow. It is tightly integrated with Lustre, but supports less storage per storage node than OpenZFS and lacks certain advanced features.
- **OpenZFS**[1]: the Open Zetabyte Filesystem, is a filesystem and volume manager with a focus on integrity and many advanced features [6]. It is a completely independent

[1]OpenZFS is the official naming for the version of the filesystem developed as a community-based project and available for Linux (formerly also known as *ZFS on Linux* or ZoL), FreeBSD and IllumOS. The original ZFS code was developed at Sun Microsystems as a free and open-source project and then continued its development under Solaris [7]. In 2010, Solaris stopped the open-source releases and development [7].

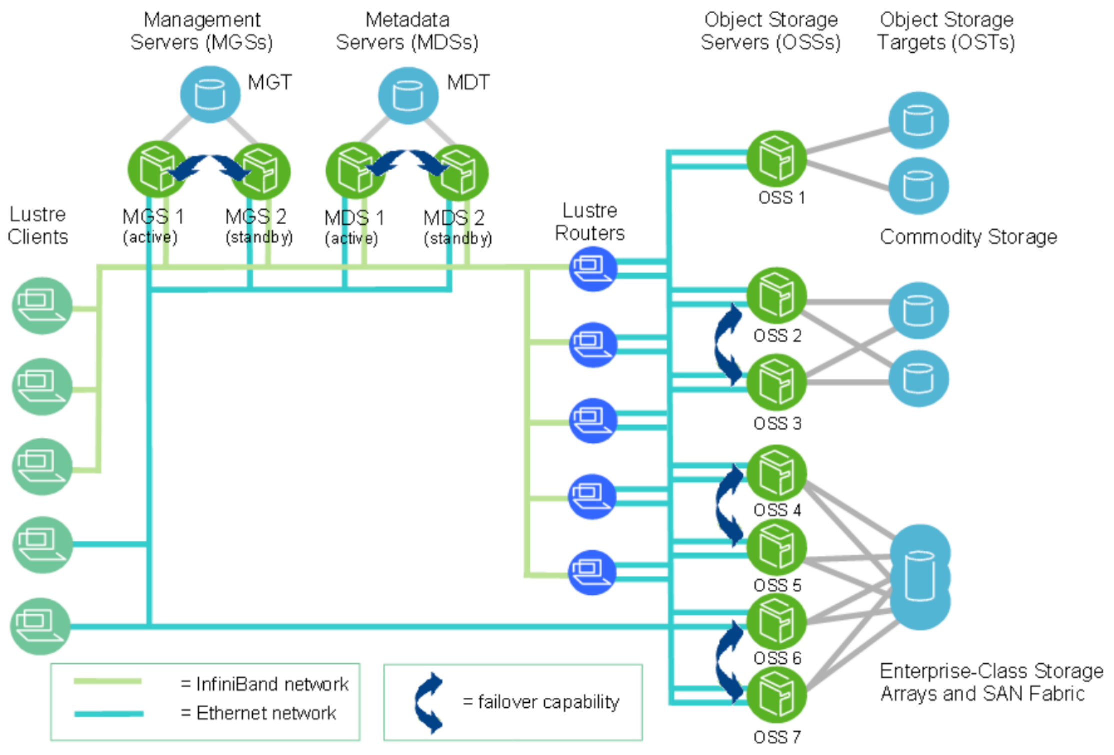


Figure 2.1.: An example of a Lustre cluster at scale [55]

project, but supports much more storage per node than ldiskfs (up to a ridiculous amount unreachable by current technology) and has a set of advanced features and performance tweaks far bigger than ldiskfs. Including but not limited to: transparent compression, transparent encryption, data checksumming, or volume management. It will be discussed in more detail in Subsection 2.3.1.

In this thesis, the only backend of interest is ZFS. The most important reason is that ZFS supports data compression while the ldiskfs backend does not. To build smart compression features into Lustre, having a backend that already supports compression provides a higher degree of flexibility which is not otherwise possible.

## 2.1.2. ladvise interface

One additional remarkable Lustre feature is its `ladvise` interface. This interface allows users to provide the filesystem with specific bits of information about the data they are using.

Currently, although the grounds of the filesystem are the same for all the forks, ZFS and OpenZFS are distinct projects developed by different entities. For simplicity, and given that in this thesis the work is carried exclusively under Linux, I will refer to the official OpenZFS [6] available for Linux and FreeBSD as ZFS. However, it is important to understand the differences between the projects and avoid confusion in discussions that involve a wider background.

Its goal is to improve resource handling by integrating user knowledge into the filesystem processing pipeline.

Users can use the `ladvise` interface through a user-space command-line program [3] and a well-documented library [2]. This allows to embed the information in programs but also allows users to provide such information about a specific group of files or a directory.

The provided data can be processed at the clients or the servers, although the interface has the potential to allow both processes. Currently, `ladvise` implements two groups of *advice*:

- Regarding information about the usage of certain data. These are implemented on the server-side, but are only functional when ldiskfs is the backend:
  - `willread` advise requests that the data is pre-fetched into the server cache. This could speed up future reads and make sure that commonly used data stays in the cache.
  - `dontneed` advise requests that some data be removed from the server cache. Allows to clean up unnecessary data from the cache.
- Regarding locking behavior. Can be used to warranty exclusive use of a certain file or to avoid locking at all:
  - `lockahead` advise is used to warranty the locking used over a file.
  - `noexpand` advise is used to disable lock behavior on a specific file descriptor.

This interface has the potential to be used as a way for users to provide compression-related information to the *smart* process developed in this thesis.

## 2.2. Compression algorithms

Compression algorithms are programs used to reduce the size needed to represent certain data [72]. They are widely used in informatics for a multitude of tasks: from media encoding (images, music files, videos) to genetic sequences or simply to reduce the size of a certain file.

When using and comparing compression algorithms, those are often characterized by two metrics: compression ratio and throughput. The evaluation of these metrics is usually tied to the data being compressed and therefore does not exist absolute values for every possible compression algorithm. Especially, since the use-case and kind of data can often change the behavior of the algorithms.

- The compression ratio is a measure of the ability that the compression algorithm has to reduce the size of the data representation. It is measured as the relationship between the size of the data before and after compression, and can be expressed in two different ways: as a reduction factor, presented in Equation 2.1 and as a reduction percentage, presented in Equation 2.2. Both are commonly used in literature, thus the need to present them here. In general, the greater the compression ratio is, the better.

$$compressionRatio = \frac{\text{Data size before compression}}{\text{Data size after compression}} > 1 \tag{2.1}$$

$$compressionRatio(\%) = \frac{\text{Data size after compression}}{\text{Data size before compression}} * 100 < 100 \tag{2.2}$$

- The throughput represents the time needed for a certain piece of data to be compressed or decompressed. It is a measure of the time taken to first compress the data and then write the compressed data back to its original location (be it disk, memory, or some remote storage). In consequence, this metric presents also a correlation with the compression ratio. The greater the compression ratio, the fewer data has to be written, and the faster the overall transfer happens. The throughput is measured in bytes per second (bps) or its multiples, e.g: Mb/s. It is also important to mention, that the throughput varies for compression and decompression since usually algorithms have different behaviors when compressing than when decompressing data. The exact calculation for the throughput can be seen in Equation 2.3.

$$throughput(bps) = \frac{\text{Data size before compression}}{\text{Time needed for compression or decompression}} \tag{2.3}$$

**Lossless and lossy compression** Within compression there exist two big groups of algorithms according to the results of the compression: lossless and lossy algorithms [72].

- Lossless algorithms are those which, after compressing and decompressing some data, can reproduce the same representation of the data as before the data was compressed. They usually make use of redundancy within the data to avoid unnecessary representation. In consequence, there is no information loss, and the data integrity is never compromised. These are the algorithms used when dealing with data whose representation is important.

- Lossy algorithms, on the other hand, cannot reproduce the representation of the data after decompression. These algorithms *lose* some information during the compression and decompression process. They have often a higher throughput, but they cannot be used when the integrity of the data is important. They are commonly used for media encoding, where some quality loss is acceptable, e.g: MP3 audio or JPEG image encoding.

In this thesis, I will only discuss in detail lossless compression algorithms. The compression requirements for Lustre are such that the integrity of the data must always be warrantied. This is especially important for scientific applications, whose data shall not be modified. In consequence, lossy algorithms are not appropriate for the task.

However, there exists a great variety of lossless compression algorithms. For this thesis, and since the work will be developed around Lustre and ZFS, only those applicable to this combination are of interest. ZFS currently supports gzip, LZ4, LZJB, zstd, and a custom zero-reducer algorithm known as ZLE [5]. Out of those, for this study, I only consider LZ4 [17] and zstd [21]. Those are the most modern ones, and overall the best-regarded algorithms within the ZFS community. Additionally, there also exist technical reasons, which will be further explored in Chapter 4.

### 2.2.1. LZ4

LZ4 is an algorithm designed for fast decompression speed. Originally designed with that single purpose [17], newer releases brought new features.

The first, is the ability to configure the relationship between throughput and compression ratio through *levels*. These levels are sometimes referred to as the *LZ4 fast* algorithm. The levels can be used to tune the relationship between the throughput and the compression ratio. The higher the level, the higher the throughput and the lower the compression ratio. Additionally, the decompression speed also increases the higher the level is.

The second is a new mode labeled *high compression* (HC). It uses the same format, but the algorithm provides much higher compression ratios at the cost of much lower throughput and is also configurable with levels. However, these levels have the opposite impact to the regular levels, where the higher the value, the lower the throughput and the higher the compression ratio. However, this mode is special in the sense that it warranties a constant decompression speed, equivalent to that of the original LZ4 algorithm, regardless of the level used for compression.

These different variations or modes of operation are still considered the same basic algorithm. That is possible because LZ4 establishes a standard compression format [18] that must be followed by any LZ4-compatible implementation. Additionally to the regular compression format, LZ4 also establishes a streaming or *frame* compression format aiming for streaming applications. Although of little use in this thesis, it is useful for streaming applications where data is compressed and decompressed and the beginning and end of the endpoints.

A more detailed analysis of the performance and characterization of the algorithm will be carried out in Chapter 4, Subsection 4.3.2.

### 2.2.2. zstd

zstd is the reference implementation of the Zstandard algorithm [21]. Such implementation is included both in ZFS and in Linux and thus is the one that will be discussed throughout this thesis. Originally designed at Facebook, zstd was born as a modern replacement for gzip, a general-purpose compression algorithm widely used. Similar to gzip, it provides a different set of levels to tune the compression ratio to throughput relationship, with higher levels achieving greater compression at a lower speed.

It is also important to note that some zstd levels require a relevant use of memory for operation. As it will be discussed in Subsection 4.3.2 and observed in Section 5.2, that is remarkable for some of the highest levels. This is even warned by the zstd authors [21], which recommend users to only use levels greater than 20 where the speed is not a concern but maximum compression ratio is required. It must also be noted, that similar to the LZ4

HC algorithm, zstd was designed to have a constant decompression throughput, regardless of the level used for compression.

Later versions of zstd introduced a new mode of operation, zstd-fast, whose levels act in the opposite direction to the regular ones. In this case, the greater the level, the lower the compression ratio and the higher the throughput. It is important to note that the zstd-fast configuration does not warranty a constant decompression speed across its levels.

Another remarkable fact from zstd is that it introduces new algorithms [19] for the calculation of the *entropy* or variability within the data. These algorithms are those which identify the redundancy existent in the data to reduce its size without data loss. These new algorithms feature a faster speed and better redundancy detection than previous counterparts and are part of the zstd success.

## 2.3. Compression in filesystems

Compression is a very rare feature within filesystems. For example, at the time of the latest update of zstd code within Linux, release 5.16 [68], there were approximately 80 different supported filesystems. The update of the zstd code [68] shows that out of them, only 4 had compression support:

- **btrfs**: The most general-purpose filesystem out of those which support compression within Linux. It is an advanced filesystem with a similar set of features to ZFS [64] but built into the kernel. It is, however, a much more recent project and some features are still unstable or not implemented [10].
- **f2fs**: Still a general-purpose filesystem optimized to be used on flash memory. It has no special features outside supporting encryption and compression.
- **pstore**: Highly-specialized filesystem to support filesystem access to a certain set of advanced hardware properties.
- **squashfs**: Also a specialized filesystem, where everything is always compressed. It is a read-only filesystem that compresses both its content and its metadata and is designed to provide a compact way of using read-only data.

This shows that compression support within filesystems is a very uncommon feature. Despite its great advantages: mainly reduced storage space, and increased storage bandwidth, it is rarely implemented. One important reason is that filesystems are critical pieces of software in a computer system, and extensive changes can be potentially dangerous to the stability of the system. Therefore, only a few advanced or specialized filesystems make use of it.

However, there is still another existent filesystem with compression support: ZFS. Unfortunately, due to licensing reasons [63], ZFS is not part of the Linux kernel, but instead supported as a different project and built against Linux. Since this is the filesystem of most interest in this thesis, it will be discussed in more detail below.

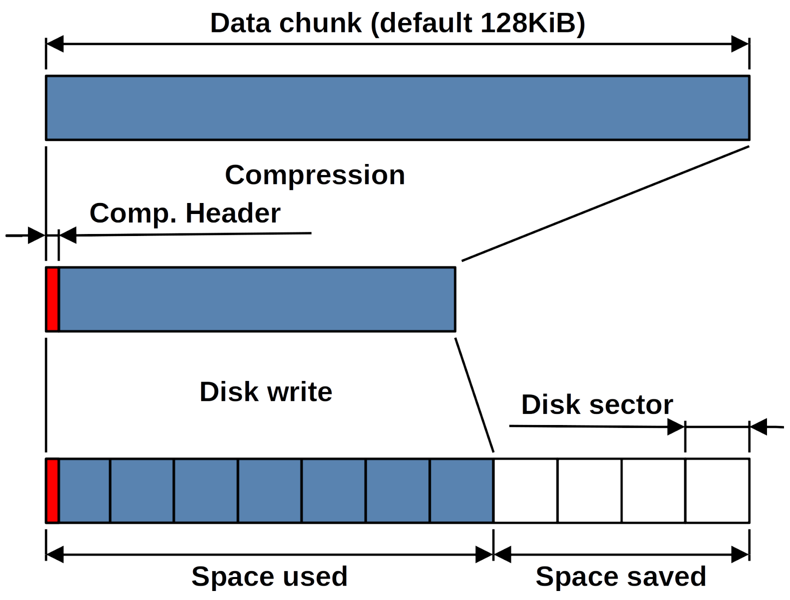


Figure 2.2.: ZFS compression process

### 2.3.1. ZFS

ZFS is a filesystem and volume manager with a focus on integrity and many advanced features [6]. It puts together a multitude of features traditionally not part of the filesystems, including drive aggregation, drive redundancy management, and management of logical volumes. Among its most remarkable features, one can find: checksumming of all data to warranty integrity, transparent built-in compression, transparent built-in encryption, and the ability to execute plain filesystem transfers maintaining the integrity checks.

However, ZFS is relevant for this thesis not due to all of its advanced features, but *only* due to its compression feature and it being compatible with Lustre. The fact that ZFS supports compression opens a door for compression support in Lustre.

**ZFS compression** As it has already been discussed, ZFS currently supports gzip, LZ4, LZJB, zstd, and a custom zero-reducer algorithm known as ZLE [5]. However, regardless of the algorithm in use, the process to compress and write data to disk is identical for all of them. A graphic representation of the process can be seen in Figure 2.2. The process begins with compressing the data and adding a custom header at the beginning. Once the data is compressed and the header attached, they are both written to disk. From this process, it is important to highlight two ideas:

- First, the header exists to store metadata like the algorithm used, the level, or the size of the compressed data. The storage of this metadata (the header) together with the data itself allows an external program to write data that is compatible with ZFS internal structures. Therefore, ZFS can understand and interpret the compressed written data. This is a great advantage that can be exploited in the implementation of compression in Lustre.

- Second, the actual benefits of the compression ratio can be limited by the relationship between the size of the data chunks and the size of the drive sectors. Drives are logically and physically split into small chunks known as sectors, usually of size 512bytes or 4KiB. These sectors can only be read and written atomically. In consequence, space savings can only happen in multiples of the sector size. Therefore, any compression unable to save at least the size of one sector will be unable to save space.

**The Adjustable Replacement Cache (ARC)** The ARC is a caching mechanism implemented by ZFS which is also of interest in this thesis. It is relevant for this work because it is a memory-intensive mechanism that uses a sizable amount of server resources. Therefore, if the smart compression ever happens in the servers, the memory usage taken by the ARC must be taken into account. By default, it is set to occupy half of the system memory [52].

The purpose of the ARC is to make use of extensive caching to help speed up IO. On one hand, it can make sure that the filesystem metadata, e.g: which parts of which file go to which disk, remains in the cache after some file has been read or written, speeding up its future uses. On the other, it can keep both recently used and commonly used data in the cache, adjusting them according to the workload of the system, hence its name: *Adjustable Replacement* Cache.

Additionally, the caching can be expanded from system memory to fast secondary storage. When heterogeneous persistent storage is available, e.g: some fast SSDs complementary to many slow HDDs, ZFS can leverage it. By using the fast secondary storage as an additional cache, the cache size can dramatically increase at a small performance penalty. This secondary storage is known as Level 2 ARC (L2ARC), and its existence can increase the caching potential, reducing the criticality of the in-memory ARC.

## 2.4. The Linux kernel

A kernel is the program within an operating system that manages and controls hardware resources. It is, therefore, the part of the operating system that allows *interaction*. As a consequence of hardware-resource management, the kernel is required to execute other complementary tasks, such as memory management; process accounting and scheduling; or I/O operations.

Linux is one of the multiple kernel programs that have been developed throughout Computer Science history. Originally developed and distributed by Linus Torvalds in 1991, it is now one of the most successful free software projects to date, and a great example of collaborative development. Its great popularity makes it the kernel of choice for a multitude of operating systems (RedHat Linux, Debian, Ubuntu...), commonly known as *Linux distributions*[2].

[2]Although in common language Linux is used to denote those operating systems which use Linux as their kernel, in this thesis Linux will always only refer to the kernel.

The Linux kernel is built using a hybrid architecture that makes use of two different paradigms: monolithic and modular design. Its monolithic part contains the basic building blocks which are needed to run the kernel and form what is known as *Linux core*. Its modular part consists of smaller programs known as *kernel modules* which extend Linux's functionality. They are most commonly used to extend hardware support, but they can have any purpose. This modularity is what allows both Lustre and ZFS to work with Linux. Since they are both programs that manage and interact with hardware resources, they must interact with the kernel. In consequence, although developed independently, they are built as kernel modules and executed as part of the kernel.

However, apart from being the foundation for the development and execution of Lustre and ZFS, Linux is also relevant for this thesis for the metric collection purpose. To be able to make a smart decision-making regarding compression, it is needed to know and understand the status of the Lustre cluster at any specific point in time. Such status is defined, in a sizable amount, by the properties and status of hardware resources. As a resource manager, Linux is both managing those resources and collecting metrics about them. Therefore, for the analysis of a Lustre system and implementation of the basic metric collection (both of which will be discussed in Chapter 3) Linux plays a very relevant role.

## 2.5. Related work

As has been discussed in the Introduction, the position of this thesis within the research space of Computer Science lies on the exploratory side (See Figure 1.1). One of the main reasons is the limited amount of previous work for compression in HPC. Although there exists some previous work, a system integration and study of this size has, to the knowledge of this author, not been executed. At the same time, the field of adaptive compression does get more research focus. Some of that work is certainly useful to understand the possibilities that exist for smart compression in Lustre and build on previous knowledge.

**Compression in HPC** Research for compression in HPC has been mainly focused on power and data reduction factors. Power usage is a relevant metric in HPC since it reduces the operational cost and avoids carbon emissions. Although it might be counter-intuitive due to the CPU resources needed for compression, compression can help reduce the power usage. Ferreira et al. [28] conducted extensive research showing that a measurable part of power consumption in HPC is the consequence of IO operations. Therefore, using the CPU to reduce the data size, and with it, the IO, has the potential to also reduce the power consumption. That same rationale lead Chasapis et al. to study the consequences of applying compression to a Lustre setup [14]. They use Lustre with ZFS as the backend and apply different compression algorithms to evaluate power use. They conclude, that compression, when correctly configured, can help reduce power usage. However, they also warn about the potential harm when too expensive algorithms are in use. For that same reason, Kuhn et al. use decision trees [46] to select the algorithms which best fit the data written by applications. They show that it can use such a decision algorithm to select the best-fitting configuration according to the data.

However, data reduction does not only impact power or performance, but also the costs for HPC clusters. For that reason, Kuhn et. al study the consequences of compression for climate data [45]. They mathematically analyze the consequences of compression and their whole impact on the cluster and its costs. They conclude that compression is a promising technology able to improve the performance and reduce the costs of dimensioning and running an HPC cluster.

### Adaptive compression

Early approaches to smart or adaptive compression considered heuristics and prediction algorithms, usually based on some kind of entropy. Early work by Sucu et. al [44] considers an algorithm that analyzes current and previous compression results to select the best-matching algorithm. Their approach applies to regular network P2P communication, and thus the data is sequentially streamed, which is a simpler use case than the highly scalable, block-based, parallel IO typical of Lustre setups. Still, more modern research keeps considering algorithmic approaches based on heuristics and entropy relevant. In 2020, Yamagiwa et. al [75] propose a real-time entropy coder to compress streaming data without stalling the process. Their approach provides lower compression ratios than common algorithms but has the advantage of working in real-time.

However, the problem of predicting the expected compression ratio of certain data or its best-fitting algorithm is complex. As will be discussed in Chapter 4, it requires several inputs with complex correlations. Therefore, some modern approaches have started considering Machine Learning (ML) and Neural Networks (NN) as feasible algorithms to solve this problem. Plehn et. al [58] produce an exploratory approach using ML for compression in HPC environments. One of the challenges of ML algorithms is gathering enough data for training. For that reason, one of their main purposes is to design a data-collection and training pipeline which fits the data and application in turn. Their results show the potential benefits of compression but would need to be validated by future research. A similar but more specific approach by Yamagiwa et. al [76] also shows promising results. They use an aggregation of Principal Component Analysis and train a NN to predict the best-matching compression algorithm for image lossless compression, also with promising results.

One additional blind spot in the research in the compression field is the characterization of the algorithms according to metrics different from throughput and compression ratio. This characterization is important to understand the CPU and memory requirements of an algorithm, in a system where other processes are executing. In [59], Froening et. al discuss these issues for a great number of algorithms. They put the focus on power usage and the possibility of offloading some of the computations to external processors. Their work shows great differences in the resource usage for different algorithms, supporting the idea that they are relevant to monitor.

# 3. System profiling and data collection

The goal of this thesis is to set the conditions for smart decision-making in the context of Lustre and compression algorithms. For that purpose, in this chapter, I analyze all the components involved and the properties of the data generated by HPC programs. The aim is to find all the metrics that might be relevant for the said task.

Therefore, Lustre must be analyzed to pick the components which are relevant for data compression. Once those components have been identified, I dig into available Computer Science descriptions, documentation, and monitoring tools, to find all possible metrics which describe them. Scientific documentation for this step will be incorporated when available. However, it is important to note that current research does not focus on metric description and analysis, but rather on the evaluation of specific metrics for a specific use case. In consequence, it is rare to find recent papers on this field, and most documentation refers to conferences, manuals, or technical descriptions.

Additionally, I will also analyze the application-generated data and the application IO patterns. The data is modified by compression and the IO patterns reflect *how* the data is being used. Since the data is the target of compression, they are both relevant for a smart decision-making.

Finally, I present a small proof-of-concept implementation to prove the viability of the discussion. On one hand, I implement some basic metric collection. That is done by creating a kernel module in Lustre which collects some relevant metrics and exposes an API for other modules to use it. On the other hand, I describe all the components involved in the `ladvise` interface and provide an example of how it could be expanded to consume input from users for compression purposes.

**Lustre components** Lustre and its components have been previously introduced in Section 2.1, where a commonly-used scalable architecture was presented. However, for this exploratory analysis, considering a highly complex and scalable architecture is unfeasible. Therefore, a simplified architecture presented in Figure 3.1 is used instead. In this architecture, the MGS and the MDS are co-located, there is a single interconnection between clients and servers, and the routers are merged into the network. This simplification is possible because the scale of the system does not change the data flow for the compression work. Still, the data flowing from the clients to the servers can transverse the Lustre routers (abstracted in the simplified architecture) and simplifications could have implications in the data collection and analysis process. Therefore, although this simplification is appropriate for this exploratory study, future work could consider the study of scaled-up architectures.

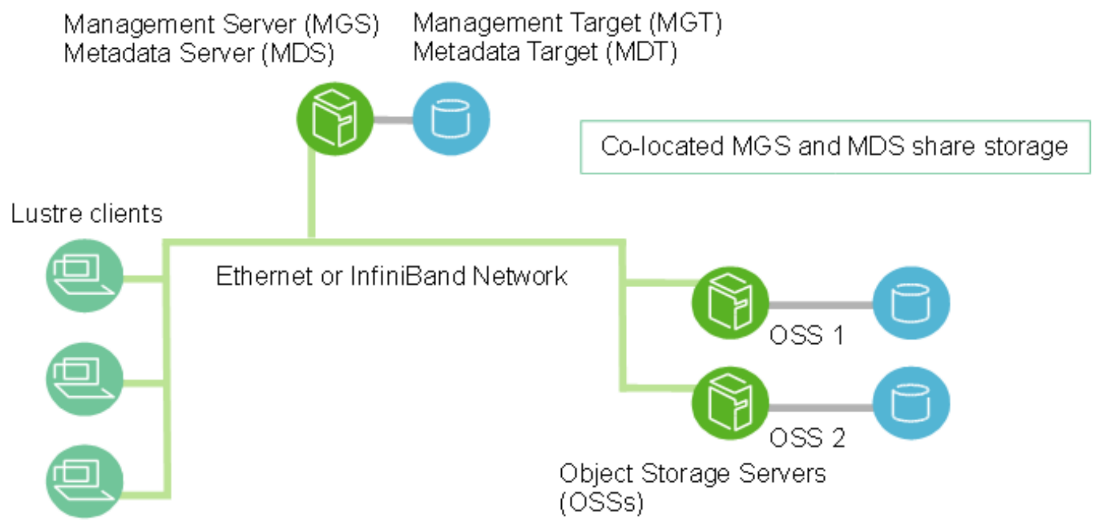


Figure 3.1.: Simplified Lustre architecture [55]

**Compression workflows** To be able to analyze the components in Figure 3.1, the expected workflows in the system need to be described first. In any regular Lustre installation, clients run scientific applications which consume and generate scientific data. When the applications write, their data is sent by the clients to the OSSs. This is done in coordination with the MDSs, which store the filesystem metadata for future use. Once the data arrive at the OSSs, it is finally stored persistently in the OSTs. When the applications read, the clients request the data from the OSSs, which fetch it from the OSTs and provide it to the clients. This is again executed in coordination with the MDSs, which contain the necessary metadata to execute the task successfully. In this process, the MGS does not play any role, apart from providing the MDSs and clients with information about other nodes. There exist also one additional workflow, known as `short_io`, where the data is written and read directly to and from the MDSs, respectively. This workflow is an optimization to the regular workflow applicable only to small files (by default smaller than 16KiB) which fit into the MDSs. This is effective due to the size of this data being similar to that of the metadata. This `short_io` workflow can be tuned or disabled by administrators. It is then decided by Lustre at runtime whether to use it or not, depending on the data size and the configuration.

At this point, it is also important to mention some additional limitations which impact the development of this work and apply to the compression workflow:

- Out of the two OST backend types available, ZFS and ldiskfs, only ZFS is considered. ldiskfs lacks any kind of compression support, while ZFS compression features exist for a very long time and are stable and well tested. Although it would have been possible to implement compression purely in Lustre to allow any backend to be used, that is well beyond this research. Therefore, reusing ZFS implementation becomes the most sensible alternative.

- Compression not considered whenever Lustre decides to use the `short_io` workflow. There are multiple reasons for this: First, although the MDSs also support ZFS as MDT backend, it is quite common for deployments to use ldiskfs instead. The reason is that the small nature of the metadata fits better the ldiskfs backend purpose. Second, even if ZFS is in use, compression should in principle be turned off. The metadata and `short_io` data are too small for compression to make a difference. From the technical perspective, in modern 4KiB block drives, 16KiB writes would need a compression ratio of at least 1.33 (reduce data to 12KiB) to save space. Additionally, the smaller the write, the more amount of space is lost due to redundancy in the ZFS raidz model. Overall, and although it could be technically possible, there seems to be no motivation to enable compression for the metadata.

In consequence, from the simplified architecture presented in Figure 3.1 and the following discussion, it can be concluded that the only components relevant for this study are: the clients, the OSSs, and the interconnection between them.

**Compression impact on components and data** Although all the Lustre components which take part in the compression have been identified and can provide some useful information, they are too coarse. For example, a generic client component has too many parts (CPU, memory, network connection) to be directly approached. Therefore, a more fine-grained view of the system is needed to provide a more detailed assessment. This fine-grained view should allow gathering a lot of information about the different components. To extract that information, an analysis to identify those components which are used for the compression workflow takes place. For that, first, the impact of compression algorithms on the system and the data must be studied. Then, the application-data flow can be analyzed on a more detailed level.

**Compression algorithms impact** In the scientific literature, scientists characterize compression algorithms mostly by their compression ratio [33, 60] and throughput [60] (or its inverse, transfer speed). In rare cases, scientific literature also discusses CPU [33] and memory, which are otherwise common metrics in non-scientific reports [17, 4, 21]. One possible explanation is that, while users and compression algorithm designers have a great interest in the resources' usage, those are very complicated to approach scientifically. For example, the algorithm throughput as described by Equation 2.3 contains a complex relationship with the CPU and memory performance due to the usage of a time variable.

Out of these, in this chapter, I consider compression ratio a data property (and one more reason to study the data flow) and take CPU and memory as key components which deserve a detailed study. As will be discussed in more detail in Subsubsec 4.3.2 in Chapter 4, the algorithm throughput is usually much higher than the bandwidth available in the whole system. Therefore, when considering the metric collection in this chapter, the focus for those components which the data just traverses will be on their bandwidth properties.

**Data flow** Since the properties of the data generated and consumed by the clients change during compression, those components which the data traverses have to be studied in more detail. For this purpose, I follow the workflows described before, but with a more fine-grained view of each of the components.

During data generation and writing, the data first moves from the CPU to the client memory. There, it might be compressed to a different memory location if compression happens in the client. Afterwards, the data is transferred to the network, which moves the data to the server. In the server, the data is directly placed in memory. If not-compressed, it will be compressed through the CPU and the compressed bytes placed again in a different memory location. Once the data is verified to be compressed and it is requested to be committed to persistent storage, it is given over to ZFS. During both reading and writing, ZFS places the data in a cache (ARC) that might be supported by fast secondary storage (L2ARC). Finally, the data is written to disk. A representation of such architecture can be seen in Figure 3.2.

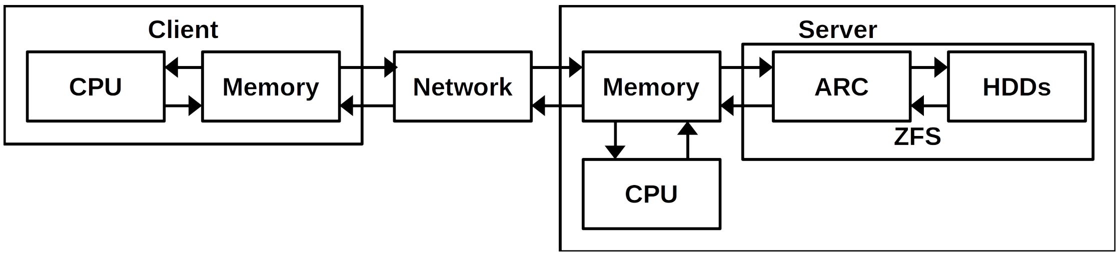


Figure 3.2.: Data flow between client CPU and persistent server storage

During data reading, the clients request the data from the server, and then the same process happens in reverse order. First, ZFS fetches the data from disk and places it in the ARC in compressed format (if not already cached). If the data is to be decompressed in the server, then the data goes through the CPU for compression. Otherwise, it is sent to the network from the memory. The client moves the data from the memory to the network, decompresses it through the CPU if necessary, and then back to memory the CPU makes use of the requested data.

As all the described components need to be studied. In the following, I will exhaustively iterate through each of them. The goal is to characterize the components and identify possible metrics that can help with smart decision-making. I will also discuss the properties intrinsic to the application-generated data (which is the subject of compression) and the IO patterns (or *how* the data is read and written). As the data generation and consumption events are always triggered by clients, for every metric the generation location shall also be considered. Client-generated metrics are directly available by clients and don't require any further processing. However, those metrics generated in the servers might be subject to limitations. These metrics need to be sent from their generation location to the clients, which requires additional data management; takes longer time than just requesting client metrics; and increases the network bandwidth. A further study of the impact of remote metrics takes place in Subsection 5.4.1 in Chapter 5. Lastly and for completion, a group of metrics marked as *cluster metrics* will also be considered. These metrics represent values from the cluster architecture that have been abstracted in Figures 3.1 and 3.2, but that are relevant for the discussion. These are metrics that often represent complex relationships between components and configurations representative of big Lustre deployments as shown in Figure 2.1. The integration of these metrics into the decision logic is certainly challenging, and would likely require further integration with the MGS. However, a discussion of such metrics is still relevant for completion and future work purposes.

## 3.1. CPU

Modern CPUs are very complicated pieces of hardware. They contain a multitude of modules for the execution of different instructions, speculative modules to guess branch decisions, and multiple levels of caching. A block diagram example of such an architecture is presented in Figure 3.3. There, we can observe in orange 4 different caches to store commonly used data, instructions (or micro-operations, which are low-level instructions), or both (shared cache). The blocks which interpret, fetch, and schedule instructions are colored in green. These modules are characteristic of modern hardware, which can both execute multiple instructions at a time, and schedule operations in a different order than supplied, to improve the effective use of resources. In purple, we can find the blocks which execute the operations. There is a multitude of these, specialized in a certain set of operations such as integer operators (ALU), divisions, or multiplications. There must be enough of these modules to avoid limiting the execution of programs on a single group of operations. The data is loaded and stored from these execution blocks from the light blue blocks which contain the registries. These are specialized memory modules with extremely fast read and write access (typically in the order of nanoseconds for modern hardware), which act as a proxy between the memories and the execution blocks. Finally, there are also two yellow blocks, these are the instruction and data Translation Look-ahead Buffers (TLB), which do the address translation necessary for multiple programs to run in parallel.

Characterizing such complicated hardware is a complex problem that strongly depends on the level of detail and the approach taken. Linux and its tools provide resources to tackle this problem in two different ways:

- **Low-level metrics**: these are metrics specific to the different components within a CPU which the kernel has no specific control over. They are usually instead collected and exposed directly by the hardware, although the kernel can configure its collection. Such configuration can be done with the assistance of *perf* [71], a kernel tool designed specifically for this purpose. *perf* can configure the Performance Counters or Performance Monitoring Units, which are the hardware self-managed registers that expose hardware metrics. Some examples of the metrics that can be collected through this process are the count of instructions per clock, the number of caches hits and misses for every single cache, and the number of failed branch predictions [71].
- **High-level metrics**: these are metrics with a higher scope and that can typically be

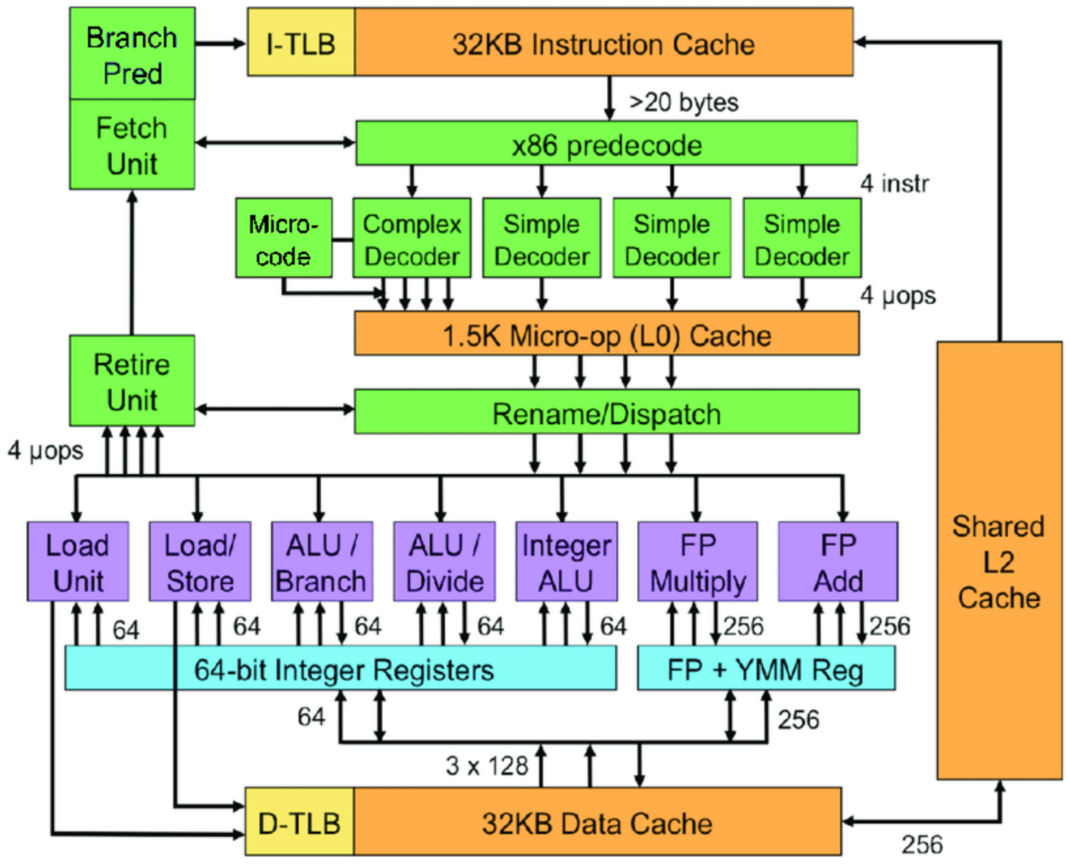


Figure 3.3.: Sandy Bridge CPU block diagram [31]. There are blocks which execute different kinds of instructions in purple (ALU/Divide and FP Multiply), fetching and queuing blocks in green, different types of cache in orange, fast registers in light blue and buffers for address translation in yellow.

directly used and collected by the kernel. They represent a certain hardware status, but the kernel has full control over them, their status, and configuration (when possible). Some good examples are the number of CPU cores in use (which the kernel can enable or disable, use or stop using at its liking), or the frequency at which they operate (controlled by the power-state drivers in the Linux kernel). Based on these metrics, it would even be possible to define additional derived metrics [11], but such task will be deferred until Section 4.3.

Since there is no limitation to the programs which users can run in the Lustre cluster, it is important to do a hardware characterization that is valid among a great variety of circumstances and workflows. For this reason, low-level metrics will be mostly dismissed. This group of metrics is usually only representative of a specific workflow and it is very hard to use them to answer generic questions, e.g: can LZ4 level 10 be used with the current resource-usage status? For the same reason, high-level metrics which represent the status of the hardware being used should play, instead, a big role in the decision-making process. Unfortunately, using high-level and dismissing low-level metrics has the drawback of a precision decrease for the metric collection. While low-level metrics are counted instance by instance, higher-level metrics are usually computed by polling. This has the advantage of reduced performance impact at the cost of more coarse and, under some workflows, less accurate numbers. However, since this work does not aim at studying and profiling everything that happens within the Linux kernel, but rather to get a picture of how HPC programs are running, this approach is considered acceptable.

For the remaining study, I will differentiate between static and dynamic CPU metrics for two reasons. First, the collection process and information provided by the metrics change depending on whether they are a static value (or very seldomly updated) or a changing number with a highly-expected variability. Second, the CPU is one of those metrics which needs to be collected both in the clients and in the servers. As the decision-making process cannot take place simultaneously at the clients and the servers, some metrics have to be sent over the network. This might pose some limitations on dynamic metrics, which depend on multiple factors (like collection frequency) which will be further studied in Subsection 5.4.1 in Chapter 5. In consequence, the following CPU metrics are considered:

- Static CPU metrics: The most remarkable and commonly used static features for CPU characterization are core number, CPU clock frequency, and CPU architecture. Due to the possibilities for frequency scaling in modern processors, the CPU clock frequency can also be considered (and will be further discussed) under dynamic metrics. Still, the lower, upper, and reference clock frequencies are static metrics set in stone at manufacturing time. All these metrics (cores, frequency, and architectures), combined, can provide an accurate approximation of how much computing power is there available in a computer system. However, out of those, I disregard the CPU architecture for two reasons: it is very hard to accurately evaluate platform differences, which must be done for all the relevant algorithms, and, in practice, HPC clusters are mostly homogeneous or compromised of a few different CPU models. Therefore, the inclusion of the CPU architecture would require a great characterization effort, at very little benefit.

- Dynamic CPU metrics: Due to the dynamic nature of CPU use, these are the majority of the metrics that can be collected from them. For this kind of metrics, I look into Linux resources and documentation, as well as common monitoring tools used in the Linux-based ecosystem. Linux currently provides two groups of metrics related to CPUs: the number of runnable processes [42] and system times [37]. The first metric, also known as CPU load, counts the number of running and ready-to-run processes and averages it over a certain time period. It is a metric commonly used to identify the CPU pressure a certain system is under. Unfortunately, the common average periods are usually too long (1, 5, and 15 minutes) for the use case in this thesis, where changes in application behavior should be identified with precision. The second metric can be used to calculate the amount of time the CPUs in the system stay idle without anything queue on them. This is a very useful metric, often used in monitoring tools like top or monitoring systems like Zabbix [67]. It provides an estimate of the amount of free CPU time available in the system. Unfortunately, modern CPUs make the process harder. In modern processors, power performance states and technologies like Intel® Turbo Boost [35] can modify the CPU frequency at run-time to improve the performance and resource and power usage. In consequence, the same percentage of idle time in two separate computers might mean different amounts of CPU power available, depending on the base and turbo frequencies the CPUs are designed for. Therefore, when calculating idle times, the CPU frequencies must also be taken into consideration.

## 3.2. Memory

Memory available to computer systems in HPC is composed of multiple memory modules. These modules can be characterized in great detail by properties like memory channels, latency, or speed, but also by the usage the kernel does of them. Additionally, since the memory is often the bottleneck in the CPU-memory communication, the memory-CPU bandwidth (represented by the CPU-memory interconnections in Figure 3.2) shall also be considered in this Section. Although still complex, memory modules are much simpler and are less configurable hardware than CPUs, making the split between low and high-level metrics unnecessary (indeed, most low-level metrics are not discoverable or only discoverable through the CPU).

To find possible memory metrics that can be of use in the system depicted in this thesis, I look at two of the most common systems to extract memory-related information in Linux: the `/proc/meminfo` file exposed by the kernel [43] and `dmidecode` [1], a utility program used to extract information from the system BIOS:

- `/proc/meminfo` **metrics**: provides human-readable information about the amount of space allocated for each of the different kinds of memory within Linux. It also exports some additional statistics like the amount of memory per CPU or the size of the page table. A non-exhaustive list of the types of memory includes the total amount, the free amount, the amount of buffered memory, the reclaimable (possible to request and use) memory by the kernel, and the cached data in memory. For the use-case in this thesis, the most relevant statistics are: the total amount of memory, since it provides the information about the maximum available capacity; the free memory, which is the memory that can be used without any interference with the system; and the available memory, which is the sum of the free memory, and that other memory that could be potentially freed (mostly buffered or cached). This latest value provides the maximum amount of memory that can be allocated, while its difference from the free memory informs about the additional memory that would need to be freed for the maximum allocation to be possible.

- `dmidecode` **metrics**: are mostly static metrics about the hardware setup. Although it can provide much more information than just memory metrics[1], for this thesis only

[1]See the SMBIOS specification: `https://www.dmtf.org/standards/smbios`

the memory ones are of interest. For this purpose, `dmidecode` can provide very useful information about memory configuration, mainly documented under *Memory Device (Type 17)* structure in the SMBIOS interface. Under such structure lay important information such as the physical memory size, speed, type, or cache. All this information can be very useful to accurately evaluate the memory power a certain system has, for example, to make the decision on *where* to compress.

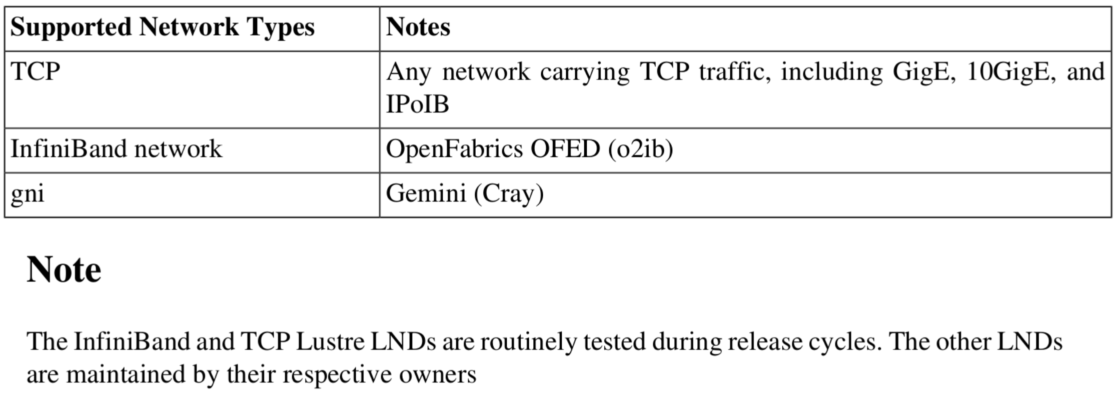
As the memory component is present both in the clients and in the servers, the discussion on the variability of the metrics also applies. In general, every metric extracted from `dmidecode` can be considered as a static metric, since hardware configuration and BIOS information will hardly ever change. On the contrary, everything from `/proc/meminfo` apart from the total memory is dynamic, since kernel memory usage varies with the use that applications do of the system.

**CPU-Memory bandwidth** The CPU-memory bandwidth is part of the data workflow presented in Figure 3.2 and shall be analyzed. As with any communication process, bandwidth can be limited by any of the elements which take part in the communication. When the limit lies on the CPU side, such maximum bandwidth could be obtained from the CPU or motherboard's datasheets and externally provided. Unfortunately, for CPUs to be the limit, all the memory slots must be filled with the maximum possible memory with the maximum possible specifications, which is rarely the case. In the most common case where memory is the limit for the bandwidth, there is no possibility to directly extract or calculate such a limit. Instead, the best possible alternative is to use memory-bandwidth benchmarks [49, 34] which exist for this purpose. In practice, research on performance bottlenecks for HPC applications suggests that CPU-memory bandwidth is rarely the bottleneck for scalable HPC jobs [26]. Therefore, due to the complexity of its calculation, and the research which suggests it is rarely a bottleneck in HPC, for this thesis, I dismiss the CPU-memory bandwidth as a relevant metric.

## 3.3. Network

Network design, structure, and interconnections in HPC is a greatly complex topic aiming to provide the maximum possible bandwidth and speed for the connection between clients and servers. As has been discussed in the introduction to this chapter, the network complexity in this thesis is simplified to a mere interconnection between a client and a server. Therefore, this section only considers the interconnection points between the clients and servers and the rest of the network. It is also common that even single nodes have multiple interfaces with different *goals*, e.g: data transfer, server failover, or administration and management. When computing the different metrics, such *goals* should be taken into account. For example, the network being fully available in the administration interface does not mean that such interface can be used for data transfer due to the complexities of Lustre networking. However, given the need to limit the complexity of network metrics and configuration collection, I will also dismiss and ignore the goals that different interfaces might have. They are instead considered part of future work.

In consumer electronics, the network interconnection points (interfaces) are mostly exclusively built around Gigabit Ethernet (GbE), which has been widely studied and should be familiar to the reader. However, network requirements in HPC are more demanding and other technologies are often used. Since this thesis focuses on Lustre, only TCP and Infiniband networks will be studied. Those are the only ones supported by Lustre, as seen in Figure 3.4.



| **Supported Network Types** | **Notes** |
|---|---|
| TCP | Any network carrying TCP traffic, including GigE, 10GigE, and IPoIB |
| InfiniBand network | OpenFabrics OFED (o2ib) |
| gni | Gemini (Cray) |

**Note**

The InfiniBand and TCP Lustre LNDs are routinely tested during release cycles. The other LNDs are maintained by their respective owners

Figure 3.4.: Table and comment from Lustre's manual which show the supported network types [55]. LND stands for Lustre Networking Driver

The main source of network-related metrics is, again, Linux. The kernel takes care of all the network communication and interface management, including the collection of runtime statistics [41], and is, therefore, the best place to look for metrics. Additionally, there exist specific tools for manipulation of the interface'´s configuration and properties, although they are protocol specific. We can find `ethtool` [38] to manipulate and inspect the status of TCP networks (despite its name, `ethtool` can be used to manipulate any interface whose driver implements `ethtool` support) and `ibstat` [62] for the equivalent in InfiniBand networks.

The kernel network-related statistics are all collected and stored in the `rtnl_link_stats64` structure, which is exported through the `/proc/net/dev` file and often consumed by monitoring tools like Zabbix. The said structure stores and accumulates counts of the total amount of bytes, errors, and successful and errored packages, both in sends and receives, among other data. Although some post-processing is needed, regularly polling this structure provides information about the accumulated bandwidth and errors for a specific network device for a certain time.

The kernel statistics provide some dynamic information about the system. However, static description of the hardware as done with CPU and memory requires also the usage of the `ethtool` and `ibstat` tools already introduced. These tools are just user interfaces to discover and manipulate information in the kernel which is either not exposed or hard to do so. In practice, some of the information they provide can be also extracted from the kernel. Still, they are very useful to *find* and *discover* metrics that might be of interest.

With different formats and different names, they both provide information about whether a certain interface has something connected at the opposite end, whether it is in use, the speed characterization of the hardware, and the speed available and configured by Linux. Out of these, the most interesting is the speed configured by Linux, which provides the maximum possible data transfer rate that can traverse a specific interface.

Finally, given that network metrics are generated both at the clients and at the servers, it is relevant whether they are dynamic or static. As already mentioned, those metrics related to the network status should be dynamic, while the hardware configuration and characterization metrics as exposed by `ethtool` and `ibstat` can be considered static. With the simplification that dismisses intermediate network points, only the server or client metrics are considered. However, in a non-simplified scenario, router network metrics would also be relevant and probably dynamic information that would have to be transferred to the decision-making location.

## 3.4. Storage

Storage-related metrics are those metrics that are related to the ZFS backend and the storage media which holds the persistent data (usually arrays of HDDs). Therefore, there are two distinguishable groups of metrics: those related to ZFS configuration and those related to hardware setup and characterization.

**ZFS configuration** The ZFS configuration can have a very big impact on a system's performance. It is mostly static, and would rarely change during deployment, but contrary to static CPU, memory, or network metrics, it is extraordinarily complex [52]. Given this, it is very hard to estimate the maximum possible performance for a certain static configuration. Therefore, I limit the collection of ZFS configuration metrics to those metrics related to the ARC. The ARC is especially relevant for this thesis because it can hold and interpret compressed data sent by the clients and read from disk. In consequence, the greater the compression ratio for the data, the more data will fit. Additionally, the ARC can be dynamically expanded or shrank depending on the memory available on the OSSs, and is a metric that should be taken into account when deciding on the aggressiveness of the compression. Having relevant data be evicted from the ARC for a lack of space can have a huge impact on the system performance, and must be taken into account.

The use of L2ARC backed by fast secondary storage (usually fast SSDs) can help reduce the impact of a shrinking ARC. Its existence and size respective to the whole pool size, together with its configuration (whether it is allowed to store complete pages or just metadata) shall also be taken into consideration. A faster and bigger L2ARC reduces the impact of a small ARC, together with the need to aggressively compress data to improve caching.

**Hardware setup** The hardware setup consists of the information about the existence of fast secondary storage (like the SSDs typically used for the L2ARC) and the configuration of the HDDs which hold the persistent data. These drives can be in different quantities and configured and organized in various ways. Altogether, their setup and characteristics define the maximum theoretical bandwidth between the server and the storage. At this stage, there is a similar problem as that with the CPU-memory bandwidth. Since exact values are unknown, estimating the maximum possible disk bandwidth is only possible with benchmarks. However, given that the hardware setup will rarely change, might be similar for the different servers, and is of great use, it is likely appropriate to run such a benchmark. More details about the possibility and process to execute these benchmarks can be found in Section 4.3.

**Hardware characterization - IO metrics** Finally, it is important to obtain metrics to profile the storage system and its dynamic behavior, since the greater its usage, the more relevant compression becomes. These metrics should be related to how, how much, and how fast ZFS is reading and writing from the disks. Similar to other dynamic metrics, Linux collects such information and provides it by exporting the `/proc/diskstats` file [40]. This file contains valuable information about the amount of data read and written, the amount of time spent on IO tasks, or the number of IOs in progress.

However, since the ZFS configuration impacts how the disks are used, the `/proc/diskstats` by itself is not enough. For example, some partitions or disks might be unused or used for different purposes than others. Meaning their usage (or lack of) wouldn't be especially relevant. For this specific reason, ZFS provides a small utility, `zpool iostat` [54], which aggregates the information from ZFS and the kernel and presents it already merged. An example of its output can be seen in Figure 3.5. Therefore, to be able to properly analyze the IO metrics, a similar process must be followed, where the ZFS configuration is obtained, and only metrics for the relevant disks or partitions are collected. The actual implementation of this process is challenging and a possible future work path, but is considered out of scope for this thesis.

```
pablo@kgpe-d16:~$ zpool iostat -vv
                  capacity     operations     bandwidth
pool            alloc   free   read  write   read  write
--------------  -----  -----  -----  -----  -----  -----
rpool            545G  3.09T      7     54  1.85M  3.32M
  mirror-0       545G  3.09T      7     42  1.85M  2.77M
    hdd2-part6      -      -      2     20   946K  1.39M
    hdd3-part4      -      -      4     21   952K  1.39M
logs                -      -      -      -      -      -
  ssd1-part2    14.8M  4.95G      0     11      1   557K
cache               -      -      -      -      -      -
  ssd1-part3    5.37G   203G      0      1    299   120K
--------------  -----  -----  -----  -----  -----  -----
```

Figure 3.5.: Example output from zpool iostat when the bandwidth, operations, and capacity properties are available at a per-device level. Two different HDDs with different partitions are in use for the main pool (rpool), with one SDD with different partitions being used as the L2ARC (cache) and ZIL (logs).

Regarding the remote or local creation of data, storage metrics behave differently from all the other previous ones. This group of metrics is only generated in the servers. Therefore, for client-based decision-making, they must always be sent to the network. On the contrary, for this group of metrics, it might make sense to set up server-side decision-making or aggregation protocol, since all the network communication could be then avoided.

## 3.5. Application data

As the subject of the compression, extracting information from the data which applications read and write is important for a successful decision-making. Metrics about application data are, however, intrinsically different from the metrics discussed before. For most of the metrics previously discussed, the program making use of the resources, e.g: the kernel, either directly provides the metrics or otherwise provides some abstraction from where they can be extracted. For application data, the HPC programs that create it do not provide any such interfaces.

Therefore, for this section, I take a different approach. Instead of identifying all possible metrics, I will search for *characteristics* of the data which are relevant for compression. Since the compression ratio is the compression property that has a direct impact on the data, the search aims to find characteristics of the data which influence the compression ratio. Through technical reports and scientific documentation, I find the *data entropy* and the *data size* to be those metrics relevant for compression.

**Entropy** Entropy is a scientific concept and a physics property associated with various scientific systems. It commonly represents the order, redundancy, or randomness of the system of study [73]. Applied to digital data, it denotes the amount of repetition or redundancy existent in the data. Its relationship with compression and compression ratio has been long studied [56], but new research continues to be published [76, 12] and new entropy-calculation algorithms developed [20]. Although digging into the mathematical details related to the entropy calculation is out of the scope of this thesis, some different aspects related to its relationship with compression are still relevant:

- The entropy measurement for digital data is not an absolute unique value calculated by a standard formula (as opposed to other fields, e.g: thermodynamics [8]). Instead, there exist multiple algorithms which extract redundancy from the data in different ways, achieving different results [20].
- Most lossless compression algorithms make use of some measure of the entropy for their compression calculation. The main reason is that the redundancy in some certain data can be successfully exploited to reduce the length needed to express it.
- The calculated entropy for certain data represents the maximum theoretical compression ratio achievable for lossless compression. Unfortunately, compression algorithms are often not able to reach this maximum compression ratio due to technical constraints. Still, the entropy sets an upper bound to the compression ratio that can be useful as a heuristic [75].

- The data encoding of the data matters. The same sequence of data written as a text file document or as a binary blob will have different *bit-wise* representations. Therefore, the entropy and its compressibility will change (usually being the text file more compressible).

**Data size** The size of the data being compressed matters both from a scientific and a technical point of view. From the scientific side, the size of the data has an impact on the entropy available, which is more remarkable the smaller the data is. In general, for very small amounts of data, it is usually very hard to find redundant information which can be exploited to reach a high compression ratio. For sizes smaller than 64KiB there is evidence [30] that entropy values are smaller. Therefore, it becomes harder to achieve a high compression ratio or even compress at all. From the technological side, the maximum amount of data that can be compressed is limited to the ZFS *recordsize* property [53]. This is the size at which ZFS writes data to persistent storage, and defaults to 128KiB. During writes, all data bigger than the *recordsize* will be split into *recordsize-sized* blocks, out of which redundancy needs to be extracted. Therefore, the data size metric is only useful for data smaller than the *recordsize* in use. In consequence, it is important to consider this ZFS parameter (specially if it is set smaller than the default) to avoid spending computational resources for a very small benefit.

This discussion on the data size also provides further justification for why the *short_io* path (discussed at the beginning of this chapter) has been kept out of the compression workflow.

## 3.6. IO patterns

Data-related metrics do not only span the data, but also its usage by applications and its transfer from the clients to the servers. Based on Figure 3.2, all the resources involved in the usage and transfer of the data have already been studied. However, a study of *how* the transfer happens is missing. This transfer consists of the data *read* by clients from the servers and the data *written* by the clients to the servers. When and how in the process of an application execution these actions happen is what is considered the application IO pattern.

The study of the different applications' IO patterns is relevant for this thesis for two reasons:

- Compression consumes resources that might otherwise be used by the application. However, compression and decompression of the data only happen when it is transferred between the clients and servers. Therefore, it is at those points in time that the resource usage is relevant.

- Compression and decompression and when and how applications read and write data are time-dependent asymmetric processes. Decisions taken at a certain point in time, e.g: compress with a slow but high-ratio algorithm at compression time because *now* there are plenty of resources available, have an *asymmetric* impact on the future, e.g: the system might be stalled if too much of that data is read at the same time.

Therefore, this section focuses on finding those characteristics of the IO patterns that are relevant for the compression process in this thesis. These characteristics can then be used to obtain information about the IO pattern of a program, which will be useful for the decision-making process described in Chapter 4.

To find those characteristics, I look at the programs involved in the process of sending data and at programs specifically designed for IO characterization. In the first group, there are Lustre and the Linux kernel. In Lustre, there are no direct statistics or properties related to the IO pattern. Linux, on the other hand, exposes some information about reads and writes through the `/proc/diskstat` file [40]. Unfortunately, the information provided there is very basic (amount of data read and written or number of IOs in progress, among others) and likely too coarse for the use case in this thesis. Among the second group, I find *darshan* [13] as the program with the better track record within the HPC and scientific communities. darshan is a scalable IO characterization tool for use in HPC [22] and is used actively for scientific purposes [74]. Its main goal is to provide hindsight into the IO for HPC applications. It can intercept dynamic linker calls to IO routines for metric collection, aggregate the collected information and write it to log files. Those files can later (or concurrently) be analyzed by specific tooling provided by darshan. In the following, those metrics collected by darshan [23] that can be of interest in this project will be presented and discussed:

- {POSIX,MPIIO_*}_{READS,WRITES}: The total amount of synchronous and asynchronous reads and writes, depending on their type. These are useful to understand the use that applications do of IO and the impact of the compression decisions on the overall performance. Programs with an important IO signature are more sensitive to compression impact than those which rarely do IO.

- {POSIX,MPIIO}_RW_SWITCHES: Amount of switches between reads and writes. It allows to know if the read and write processes are concurrent or rather happen in different phases. This has implications, since having a dedicated read or write phase allows the commitment of additional resources for compression. Additionally, it is not uncommon that programs that deal with a great amount of data must commit some temporary information to the servers. In this case, the data written is later read.

- POSIX_FSYNCS, POSIX_FDSYNCS, MPIIO_SYNCS, MPIIO_NB_{READS,WRITES}: Provide information about the split between synchronous and asynchronous reads and writes. These values inform about whether the application is doing sync or async IO. The difference is crucial for the use case in this thesis. During synchronous IO, the applications wait for the IO operation to finish before continuing. Therefore, the computation process completely stalls until the IO happens. In these circumstances, any speed-up or slow-down generated by compression directly impacts the application. During asynchronous IO, the application does not stop to wait for IO. In this case, compression must be careful to not take excessive resources from the application that might prevent it from pursuing its goal.

## 3.7. Cluster metrics

This group represents metrics that are not local to any of the components of the Lustre cluster, but that instead span the cluster as a whole. They can be related to the cluster organization and configuration. But also to the overall usage of nodes for applications that span multiple nodes in the cluster and communicate with each other.

An overall discussion of scaled-up clusters and their metrics is out-of-scope in this project as already justified at the beginning of this chapter. However, it might still be appropriate to discuss some simple metrics which might be relevant for this simplified study. These metrics do not directly belong to any of the previous sections or can be observed in Figure 3.2. Instead, they represent the properties of the cluster as a whole and could be obtained from different locations. For example, architectural metrics could be provided by the MGS and network metrics by Lustre routers:

- **OSS-client ratio**: This ratio is a relevant metric since deployments (even small ones) will rarely have a 1:1 client-server ratio as in the simplified scenario of this thesis. The lesser the amount of OSSs, the fewer the needed communication, and the most important would be to execute computations in the clients.

- **OSSs routable per client**: In scaled-up Lustre clusters, not all OSSs might be reachable by every client. Complex network configurations allow to split the network into different groups or assign specific servers to specific groups or clients. Therefore, the data read and written by some clients might only be ever transferred to a specific amount of servers. These servers are the ones carrying the data and the server-side load by that specific client. Although this value might differ for different clients, it is a consequence of the cluster structure and configuration and is therefore considered a cluster metric.

- **Cluster network**: Current overall network usage in the cluster. This is unlikely to be extracted directly from Lustre. However, there exist some tools like `tstat` [25] designed exactly for this purpose. Deeper integration with the system would be challenging. However, having an overall understanding of the network usage at a cluster level would be greatly beneficial.

## 3.8. Implementation

To show the viability of the system profiling discussed above, I carry a small proof-of-concept implementation. The main goal is to show that the previous discussion is not only based on hypothetical ideas but the system profiling and metric collection can be done efficiently within Lustre's code. I implement some metric collection in Subsection 3.8.1. This implementation is in no way exhaustive and only takes care of a small subset of which I consider to be the most relevant metrics for the system in this thesis. Additionally, since the information required for Sections 3.5 and 3.6 can rarely be obtained directly through Lustre or Linux interfaces, I also look into the `ladvise` interface in Subsection 3.8.2. There I will show that it is possible to extend the ladvise interface to provide Lustre with arbitrary information about the data being written. Finally, in Subsection 3.8.3 I sketch how remote metric collection would be possible, although the complete implementation is not feasible due to timing constraints.

### 3.8.1. Metric collection

The metric collection has been implemented with a module in Lustre named lmetrics, which collects all the necessary metrics periodically and provides an API for other modules to use. The implementation goals of this module are:

- To introduce a minimum system overhead while collecting metrics by making use of interfaces already available in Linux and Lustre.
- To provide consumers a non-mutable set of metrics with minimal overhead.
- To maintain an up-to-date set of metrics to give consumers an accurate picture of the state of the system.

For these reasons, the module basically consists of a global structure, a simple API which copies that structure for the API users, and an asynchronous background process for metric collection.

The global structure which stores the relevant metrics is lmetrics_vals struct, which is in turn compromised of one instance of lmetrics_s and another one of lmetrics_d, representing the static and dynamic metrics, respectively. These structures can be seen in Listing 3.1. This structure contains all the collected information in a pre-processed format, so that metrics can be consumed without additional overhead.

```c
#define MAX_LMETRICS_IFACES 8

struct lmetrics_s {
        unsigned int cpus;
        unsigned long total_mem;
        struct net_device *device;
        u32 iface_speed;
} lmetrics_s;

struct lmetrics_d {
        unsigned int idle_perc;
        long available_mem;
        unsigned int rx_perc;
        unsigned int tx_perc;
} lmetrics_d;

struct lmetrics_vals {
        struct lmetrics_s stat;
        struct lmetrics_d dyn;
};
```

Listing 3.1: structs

The modules in Lustre that might need to access the collected metrics, are those making decisions about compression. Therefore, for these modules to access the collected data and to make fast, online and smart decisions, they need to access the lmetrics_vals structure. For that purpose, the lmetrics module provides a public API which returns a copy of the current metrics, as seen in Listing 3.2. Although the current API provides to whole lmetrics_vals structure, it would also possible to extend it for users to request only specific parts of it, reducing the memory footprint.

```c
static struct lmetrics_vals metrics;

struct lmetrics_vals *lmetrics_get_metrics(void)
{
        struct lmetrics_vals *m;
        OBD_ALLOC_PTR(m);

        /* Return a copy, consumers shall not modify internal struct */
        memcpy(m, &metrics, sizeof(struct lmetrics_vals));
        return m;
}
EXPORT_SYMBOL(lmetrics_get_metrics);
```

Listing 3.2: export API for consumers

The asynchronous metric retrieval is executed through a delayed_work task, which gets rescheduled upon completion. This warranties that the metric collection happens at regular intervals. This is necessary both to warranty that API consumers get up-to-date data and to accurately calculate deltas for the metrics that require it. The current implementation as seen in Listing 3.3 considers a single recurring task where all the different metrics are updated. However, the timing could be adjusted for the different groups of metrics by scheduling multiple delayed_work tasks. The maximum precision of these scheduled tasks is a jiffie. A jiffie is a time measurement determined by the kernel configuration options. They jiffies are a software clock determined by the HZ kernel constant, whose value varies between 100 and 1000. Therefore, the timing of the tasks rescheduling should be limited to 10ms to warranty compatibility with any kernel configuration.

```c
#include <linux/workqueue.h>

static struct delayed_work task;

static void collect_metrics(struct work_struct *work)
{
        memdyn_metrics();
        cpudyn_metrics();
        netdyn_metrics();
        /* Collect dynamic metrics and update global metrics struct */
        if (!exit)
                schedule_delayed_work(&task, cfs_time_seconds(1));
}

static int metrics_init(void)
{
        one_shot_metrics();
        INIT_DELAYED_WORK(&task, collect_metrics);
        schedule_work(&task.work);

        return 0;
}

static void metrics_exit(void)
{
        cancel_delayed_work_sync(&task);
}
```

Listing 3.3: Schedule a delayed work

As it can be seen in structure `lmetrics_vals` from Listing 3.1, I have implemented collection of CPU, memory and network metrics. These are justified as the most relevant metrics, so proving the possibility to implement them is certainly relevant. Some specifics of the implementation and challenges found in the process are described below.

### CPU

Implementing CPU collection metrics involves collecting the number of CPUs in the system and calculating the percentage of time those CPUs where unused (`idle_perc`). These metrics where chosen for being intuitively the most relevant ones out of those ones described in Section 3.1. To extract the number of CPUs, good documentation is not available available online, alathough some tangential documentation [39] together with inspection of kernel sources[2] provided information on how to use and interpret the different CPU states. To extract the idle percentage (which is the metric used by programs like `top` to present the % of idle time in the system), the sources in `fs/proc/stat.c` were used. I reimplemented a simplified version of the different statistics to only store the idle time which is of interest in this work.

### Memory

Collecting memory metrics was actually uneventful. It came down to finding the correct metric out of those exported by Linux [43] and the interface which exports it. Linux keeps track of many different kinds of memory, being usually the *free* memory that which systems can use. However, when memory space allows it, Linux also does strong use of memory caching, and is not uncommon that more than half of the memory can be used for that purpose. That memory should in most of the cases be easily usable for other purposes. Because that is not always the case, Linux also provides an *available* memory value, which is the memory it considers to be fastly reclaimable, and is warrantied to be always greater than the *free* memory. Therefore, this is considered as the reference memory metric for this initial implementation[3]. To extract the value, the corresponding interface needs to be found.

[2]`https://elixir.bootlin.com/linux/latest/source/include/linux/cpumask.h#L915`
[3]This is also what is done by standard monitoring systems like zabbix

Looking into `fs/proc/meminfo.c`, which generates the `/proc/meminfo` file under Linux simplified the process.

### Network

The network implementation is by far the most challenging one. The main isssue is its complexity, since the interfaces that are of relevance need to be identified, their maximum speed extracted as a *one-shot metric* and then periodically checked to obtain their current transfer speeds. In this process, I do one great simplification. Instead of considering all the interfaces, only the one with the fastest speed will be selected. The reason for this is that clients usually only have one ethernet interface for management and one fast infiniband interface for data transfer. Since this is a proof-of-concept implementation and remote metrics are not fully implemented (for servers the 1 interface assumption will not always be accurate), this simplification should suffice at this implementation stage.

To get the implementation right, I had to iterate through 3 different approaches until I found a satisfactory solution:

**Approach 1:** Initially, I attempted to use exclusively Lustre LNet interfaces to extract the network configuration, translate LNet structures to Linux network structures and get all the data for Linux. For that, `LNetGetId` function seemed a good solution. Together with `lnet_nid_to_ni_addref`, it can provide all Lustre links in use. The problem with these interfaces is that they are only partially exported out of the LNet kernel module. More specifically, a `delref` equivalent to `lnet_nid_to_ni_addref` does not exist, which creates serious problems during shutdown. In consequence, this approach was discarded

**Approach 2:** Second, I tried to avoid the network configuration, and simply request the network links in use directly from LNet. For that, it is possible to use `lnet_inet_enumerate`. This function needs the user to provide a specific network namespace for the enumeration to take place, but that is easily solvable getting it through Linux interfaces. However, at this point a new problem became evident. To get the maximum speed and network metrics for a link, the Linux network structures for the device in question need to be obtained, but unfortunately only LNet structures are available. Up to this point, I have not found an API which can translate between them. Either because the Linux network stack is specially complex or possibly because it does not exist. Consequently, this approach is also discarded, and with it, the expectation to get the networking information directly from Lustre.

**Approach 3:** Finally, and given that I did not find any way to get the necessary information from Lustre, I fallback to requesting the information directly from Linux. Given that this implementation works under the assumption that there is only one fast interface (which is the one in use for Lustre for data), I can simply iterate over all of them and select the fastest one. This is considered sufficient for this Master Thesis, but this approach could also be combined with Approach 2, to identify the cases where multiple fast Lustre interfaces are in use. This final work for the network static configuration can be observed in Listing 3.4.

```
static void get_network_device(struct lmetrics_s *mstatic)
{
        u32 speed = 0;
        struct net_device *dev;
        struct net *ns = &init_net;

        rtnl_lock();
        if (current->nsproxy && current->nsproxy->net_ns)
                ns = current->nsproxy->net_ns;

        for_each_netdev(ns, dev) {
                struct ethtool_link_ksettings cmd;
                int ethtool_ret;
                struct in_device *in_dev;
                int flags = dev_get_flags(dev);

                if (flags & IFF_LOOPBACK) /* skip the loopback IF */
                        continue;

                if (!(flags & IFF_UP)) /* skip not UP interfaces */
                        continue;

                /* skip ifaces without IPv4 */
                in_dev = __in_dev_get_rcu(dev);
                if (!in_dev)
                        continue;

                ethtool_ret = __ethtool_get_link_ksettings(dev, &cmd);
                if (ethtool_ret == 0 && cmd.base.speed > speed) {
                        speed = cmd.base.speed;
                        mstatic->device = dev;
        }
        rtnl_unlock();

        mstatic->iface_speed = speed;
}
```

Listing 3.4: Network metric collection logic

### 3.8.2. Process new ladvise flags

In its current state, Lustre only processes a small amount of ladvise flags. Some of them are processed in the clients, while others are processed in the servers. I will cover here the process of adding a new ladvise flag to the list, even though such flag has not been implemented in practice. The goal is to both show-case the process and simplify future work of other people in this area. The new flag to be added is called *compressed* and represented by the `LU_LADVISE_COMPRESSED` enum value.

The first step is to add such a flag to the list of available flags. The current list with the addition of the compressed can be seen in Listing 3.5, which is available in the `lustre/include/uapi/linux/lustre/lustre_user.h` file.

```
enum lu_ladvise_type {
        LU_LADVISE_INVALID      = 0,
        LU_LADVISE_WILLREAD     = 1,
        LU_LADVISE_DONTNEED     = 2,
        LU_LADVISE_LOCKNOEXPAND = 3,
        LU_LADVISE_LOCKAHEAD    = 4,
        LU_LADVISE_COMPRESSED   = 5,
        LU_LADVISE_MAX
};

#define LU_LADVISE_NAMES {                                                  \
        [LU_LADVISE_WILLREAD]           = "willread",                       \
        [LU_LADVISE_DONTNEED]           = "dontneed",                       \
        [LU_LADVISE_LOCKNOEXPAND]       = "locknoexpand",                   \
        [LU_LADVISE_LOCKAHEAD]          = "lockahead",                      \
        [LU_LADVISE_COMPRESSED]         = "compressed",                     \
}
```

Listing 3.5: ladvise flags with the new compressed flag implemented

Once that is done, correctness checks for the new advice have to be run whenever some user requests them. Those checks lay in the `ll_ladvise_sanity` function within the `lustre/llite/file.c` file. Since no additional checks to the default one have been implemented for the example flag, the corresponding listing is skipped here. However, when a new flag requires extra parameters to be sent, e.g: "compressed" + "lz4", those must be validated too.

After the input has been validated, the client must decide if it executes some task or forwards it to the server. That decision making happens in the `ll_file_ioctl` function in the same `lustre/llite/file.c` file. An abbreviated representation of the relevant parts of the function can be seen in Listing 3.6. In that function, the advices which correspond to client-side tasks request the execution to the corresponding task. Those which correspond to server-side tasks will call `ll_ladvise`. That function does some preparatory steps, and then initiates a remote IO call to forward the information to the server. Such information is embedded in the remote call, and could be intercepted at any time of the process, in case the decision-making in the client would have to happen at some point during the network process.

```
        switch (cmd) {
        ...
        case LL_IOC_LADVISE: {
        ...
                switch (k_ladvise->lla_advice) {
                case LU_LADVISE_LOCKNOEXPAND:
                        rc = ll_lock_noexpand(file,
                                k_ladvise->lla_peradvice_flags);
                        GOTO(out_ladvise, rc);
                case LU_LADVISE_COMPRESSED:
                        /* Call any function for local processing */
                        /* Call ll_ladvise to forward to the servers */
                default:
                        rc = ll_ladvise(inode, file,
                                k_ladvise_hdr->lah_flags, k_ladvise);
                        if (rc)
                                GOTO(out_ladvise, rc);
                        break;
                }
        }
        ...
}
```

Listing 3.6: lavise client-side logic

Once at the server, the ladvise information is process in two different locations. First, in the `lustre/ofd/ofd_dev.c` file, generic implementations which affect all the backends are executed. The relevant code can be seen in Listing 3.7 Those implementations that require some special processing from the backends (ZFS, ldiskfs), can call `dt_ladvise` to forward the ladvise to the backends, which must implement the processing through their corresponding `.dbo_ladvise` function.

```
static int ofd_ladvise_hdl(struct tgt_session_info *tsi)
{
        ...
        for (i = 0; i < num_advise; i++, ladvise++) {
                ...
                switch (ladvise->lla_advice) {
                default:
                        rc = -ENOTSUPP;
                        break;
                case LU_LADVISE_WILLREAD:
                        req->rq_status = ofd_ladvise_prefetch(env, fo,
                                tbc->local,
                                start, end, dbt);
                case LU_LADVISE_COMPRESSED:
                        /* Call any function for server processing */
                        /* Call dt_ladvise to forward to backends */
                case LU_LADVISE_DONTNEED:
                        rc = dt_ladvise(env, dob, ladvise->lla_start,
                                ladvise->lla_end, LU_LADVISE_DONTNEED);
                        break;
                }
                ...
        }
        ...
}
```

Listing 3.7: lavise server-side logic

### 3.8.3. Remote metrics

Remote metrics can be collected making use of the same `lmetrics` kernel module describe above. The module can dynamically detect if other modules are loaded. As luster server-side modules only exist in servers, their presence is a clear identifier. Therefore, if the server-side module is detected, additional metrics can be collected. An example can be seen in Listing 3.8.

```c
#include <linux/module.h>

mutex_lock(&module_mutex);
osd_module = find_module("osd");
if (osd_module)
     /* Trigger server-side collection metrics */
mutex_unlock(&module_mutex);
```

Listing 3.8: Identifying if Lustre server-side modules are loaded

However, there is extra code necessary for the metrics to be sent and received. That imposes an additional engineering challenge: either a new communication path between the servers and the clients would need to be created; or the reuse of a current path made possible without affecting its current usage. In consequence, and to avoid diverting all the resources of this thesis to a complicated engineering challenge, such implementation is considered out-of-scope.

# 4. Smart compression design

Once the possible system metrics and data properties with an influence in compression have been discussed, we can start discussing smart compression, the core of this thesis work. As has been discussed in the Introduction, the end goal of this thesis is to make the compression configuration *smart*. This is done by integrating the system metrics and data properties discussed in Chapter 3 with user knowledge about the programs running, and expert knowledge about the implications of compression. For this purpose, smart compression is defined as compressing and decompressing data making the *most appropriate* use of resources, without penalizing the main program execution. *Most appropriate use* is specifically worded. I want to offer the flexibility for users and programs to decide what *most appropriate* means for them. For some, it might be to extract the maximum resource use, while others want a low average resource use or to reduce power usage to the minimum (e.g: using algorithms with a lower CPU footprint).

In consequence, smart compression consists of advanced decision-making biased by users' and programs' input. As the user or program knows better their workload, it should be possible for them to *suggest* a specific algorithm or compression flow. However, users and programs cannot know the specific state of the system at a certain point in time. For example: even if the user requested compression, it might be appropriate to disable it if the node executing the computations is close to running out of memory and network utilization is low. Therefore, to characterize smart compression, I will analyze three different tasks: the decisions that can be taken, the aggregation of the information which can be extracted from the system for decision-making, and the information that the users can provide.

Therefore, this Chapter is organized as follows: First, in Section 4.1, a detailed study of the decisions which are possible and the trade-offs involved are presented. Afterwards, in Section 4.2, the different implications that reading and writing have on those decisions are considered. Once those are clarified, Section 4.3 goes deeper into a theoretical discussion for smart decision-making using the data collected in Chapter 3. Finally, the different possibilities for user and program to bias the decision-making is discussed in Section 4.4.

However, before following deeper into the problem, and given that this is an exploratory study, it is important to set some hypotheses and limit the feature space of the discussion. Therefore, in the following:

- As it has already been covered, out of the two OST backends available in Lustre, only ZFS is considered. The main reason is that compression features already exist in ZFS and are well-regarded and well-tested. Since this hypothesis was applied for the whole discussion in Chapter 3 it is, therefore, a requisite to apply it also here.

- The discussion only considers the data flowing between the clients and the OSTs and assumes ZFS compression is on. Having ZFS compression on in the OSTs is a common setup, which this thesis aims to improve. Therefore, unless the clients explicitly disable compression for some specific data, it would as a last resort be compressed in the server with the ZFS-configured algorithm. It would, however, be possible for smart compression to explicitly disable compression. For example, in the ideal case where the application can make perfect use of system resources or when writing completely incompressible data.

- Only LZ4 and zstd algorithms will be discussed. In practice, only the algorithms available in ZFS [5] can be used: gzip, LZ4, LZJB, zstd, and a custom zero-reducer algorithm known as ZLE. Out of those, LZ4 is chosen for being the default algorithm in modern ZFS pools [5] and zstd for being a modern replacement for gzip, with increasing adoption in the field [21, 48, 68]. One additional consequence of using LZ4 and zstd is that both algorithms support different modes or *levels*. These allow tuning the algorithm performance, which a smart compression module can make use of and benefit from.

- During the whole discussion taking place in this chapter, one important goal will be to *smartly* saturate the system. Since, commonly, scientific applications cannot efficiently use all the available resources, the smart compression aims to increase the resource usage. Of course, this saturation must consider all of the available metrics. Simply saturating the CPU by using a very slow but high-ratio algorithm, while making the program stall and reduce the network load is certainly not a *smart* use of resources.

## 4.1. Decisions

For the compression capabilities to be considered *smart* the decisions to be taken play a major role. In the complex system which is the focus of this thesis and that has been described in Chapter 3, there is a multitude of decisions that can be made. Some examples are:

- Lustre networking configuration.
- The total pool of memory which is allowed to be used for compression.
- Compression and decompression location.
- Algorithm configuration to use for compression.
- ZFS and Lustre data-related parameters.

However, the scope must be limited to fit into a thesis project. In consequence, out of those, the compression and decompression location and the algorithm configuration to use are the only ones chosen. These two decisions answer the *where* and *how* questions for compression and can serve as the groundwork for any future approach to smart compression. Also, modifying the compression location or the algorithm configuration have a wide impact on the resources used and could potentially lead to big differences in terms of performance and throughput. Therefore, in this Section, the consequences of these decisions will be further analyzed.

However, before the analysis, it is needed to study how the decisions could take place. For that purpose, we revisit Figures 3.2 (page 20) and 3.1 (page 18) from Chapter 3. They show represent a simplified data flow, and simplified typical Lustre deployment, respectively. From them, some basic conclusions can be extracted:

- There exist two possible locations where data can be compressed and decompressed: The CPUs in both the clients and the servers.
- The data is both generated and consumed at the clients. Therefore, any communication process is always initiated from the client's side. The servers are passive entities waiting for instructions.
- The clients are where scientific applications run. Therefore, there is more information available about the data in the clients; whereas other system information is equally available to the clients and the servers.
- There exist a multitude of clients per server. In consequence, servers can be considered a shared resource.

From these points, a clear difference in the role that clients and servers play concerning compression can be observed. Additionally, it is common that client machines are more powerful and capable than servers due to the different tasks they have been assigned within the cluster. In consequence, in this work, I assign the clients the role of the decision-maker. It will be in the clients that the decision on *where* and *how* to compress will be taken.

In the following parts of this Section, I discuss the implications of the different decisions and present ideas and important considerations. Since the decisions' implementation is not possible in this thesis due to technological and time constraints, the discussion is not of great depth. Instead, it considers different ideas that shall be useful both for discussions further in this chapter and for a possible follow-up work that might consider the implementation.

### Compression and decompression locations

Data can be compressed or decompressed either in the clients or the servers. Looking again at the data flow in Figure 3.2 and the basic cluster setup in Figure 3.1, some basic conclusions can be extracted:

- If the data is compressed and decompressed at the clients, the network usage and the memory-network bandwidth both at the clients and the servers will be reduced.
- When client-compression takes place, the computing requirements are on the exclusive-use resource. When server compression takes place, the computing requirements are on the shared resource.
- In current deployments where compression happens in ZFS, the compression is a server-exclusive task.

Given these observations, a trade-off between the use of exclusive and shared-use resources can be observed. On one hand, the trade-off is biased towards the clients when network and memory-network bandwidth are of relevance since client compression can potentially reduce its requirements. This bias is also more remarkable the more compressible the data is and could also be a negative bias in the case of incompressible data. In such a case, where compression is attempted but could potentially fail, server-compression might be preferred. The speed of the compression process is less relevant when it happens in the servers since they do not execute scientific programs, and a small penalty might be acceptable. In consequence, the nature of the data also plays an important role in this decision. On the other hand, the trade-off is biased towards the servers when shared resources are of no or little use. This could potentially be more remarkable in existent installations where compression is already executed by ZFS and clusters designed and dimensioned for that use case. Although the influence of all the possible metrics will be studied in greater detail in Subsection 4.3.1, the outline is already clear. For location decision-making, both client and server computational resources, network and network-memory bandwidths, and properties of the data must be taken into account.

It is also remarkable that the collection process plays a role in this decision. In general, client metrics will have better accuracy, since the collection and consumption happen at the same place. Server metrics might add to the network bandwidth of the cluster (See Subsection 5.4.1 in Chapter 5 for an actual study of this problem) limiting their use, and the latency of such communication might be non-negligible. Therefore, this data collection process also supports a bias towards the clients due to the clients making the decision.

Additionally, up to this point compression and decompression have been given the same treatment. However, some small differences deserve further comments. First, during decompression, the data internals cannot have a negative effect, such as trying and failing compression. In the worst case, if the data is not compressed, it can be used directly without attempting decompression. Thus, the lack of penalty for incompressible data makes the decompression location decision slightly biased towards the clients. Second, decompression pipelines are usually more predictable. The compression ratio is known beforehand and the algorithms depicted in this thesis (LZ4 and zstd) have more stable decompression than compression ratios, as discussed in Section 2.2. Therefore, the location decision for decompression would usually have lower uncertainty than for compression.

### Algorithm configuration

Clients need to decide on a specific algorithm configuration for each of the pieces of data sent to the servers. Given the algorithms that this work discusses (LZ4 and zstd), such configuration is compromised of two variables: the algorithm itself (which could be LZ4, zstd, or to not compress at all), and the algorithm *levels*. Both LZ4 and zstd allow for a *level* or *acceleration* integer value to modify the trade-off between compression ratio and throughput, usually also involving different amounts of CPU and memory usage. Unfortunately, the resources needed to compress certain data and the final compression ratio achieved are hard-to-predict problems [33, 77]. In consequence, decisions about algorithm configurations are most often educated guesses and, at best, a consequence of good heuristics. Still, the metric collection and analysis developed in this thesis should allow to incrementally improve such guesses or heuristics, by repeatedly collecting usage information and mapping it to the algorithms in use.

The possibility that the data is never compressed must be considered too. It could be the case when the smart decision-making detects that there are not enough resources for compression to work or to be effective. It could also happen that the compression of the data is attempted, but failed, leading to the data flowing uncompressed through the setup.

It would also be possible to implement more complex decision-making processes, such as retrying compression in case of failures or low compression ratios achieved or recompressing already compressed data to reduce storage use . Such analysis is out of the scope of this thesis but will be further discussed in future work in Chapter 6.

Similar to the location decision, decisions on algorithm configuration are different during compression and decompression. First of all, during decompression, the algorithm cannot be chosen since it is directly determined by how the data was *previously* compressed. In consequence, it might feel that there is no algorithm configuration to take during decompression. However, another approach is to interpret that the decision exists, but was simply taken *ahead* in time. Therefore, while deciding on a compression algorithm, such a decision compromises both the compression and decompression decisions. Second of all, the *levels* for the algorithms depicted in this thesis have more stable decompression than compression costs. Therefore, although the information about how and when the data will be read is unavailable at compression time, there is one less level of complexity to consider. Overall, the lack of information; the necessity to take the decision *ahead* of time; and the constant decompression costs for different levels, should bias the use of more efficient decompression algorithms. These are most likely to provide better results during reads, when the decision cannot be reconsidered, and is probably a more conservative approach to avoid smart compression strongly negatively affecting reads.

## 4.2. The read vs. the write path

Up to this point, the processes of compression and decompression have been treated mostly the same, except for some narrow comments regarding the decisions. However, the processes of reading and writing are intrinsically different from several perspectives:

- **Data usage**: during writes the use of the data is clear, it happens *now* and only once. Reads, on the contrary, can happen none or multiple times and happen at undefined future times (from the algorithm configuration decision perspective).
- **Algorithm performance**: some compression algorithms, among others those relevant for this thesis, behave completely differently during compression and decompression. Their requirements and demands regarding CPU use, maximum throughput, and memory are different.
- **Data information**: During writes, the decision-maker, the client, has access to the data and can analyze it. This is relevant since there are data-related metrics that are

of interest, as discussed in Section 3.5. However, during reads, the client lacks access to the data, since it lies on the servers, and thus, it must be provided. Additionally, the information available is different, since the data changed after compression.

In the following of this section, the previous perspectives will be discussed in more detail. However, for the rest of the chapter, the discussion will focus on the write path and the compression process and decisions. The discussion of the compression process and its decisions is already complex enough to fill the full-fetched discussion part of this thesis. Adding the read path and the decompression perspective does not only add a proportional amount of work. It also adds a deeper discussion and analysis of the possible interconnections and ways in which one affects the other. Since that is unfeasible, a deeper discussion about these topics and characterization of the read path is considered part of future work.

### Data usage

There exists a clear difference between the usage writes and reads do of the data. Every write, even when writing the same data, is unique and produces a new piece of data on storage. Therefore, every write must consider both the location and the algorithm configuration decision. Such a process is different for reads since the same data can be potentially read multiple times (or even not read at all). For each read, even for the same data, a different location decision must be taken, as those reads will happen at different points in time, with different cluster circumstances. However, the algorithm configuration decision cannot be taken, since the data is already compressed.

This poses a great asymmetry between reads and writes. Writes need to consider not only their current circumstances but also the possible circumstances in which reads might happen. For example, it might be appropriate, at a certain point in time, to use a very aggressive algorithm during compression. However, that also conditions any future reads of the data, which might be slower and require greater resources than if the data had been compressed with a different algorithm. To solve this, there might be different alternatives: obtaining information about the future use of the data (potentially through user hints), trying to predict it, (risking creating greater harm at read-time), or taking a conservative measure to strongly benefit algorithms with fast decompression speeds. This problem and its possible solutions are of the greatest tasks that must be solved when considering a deeper analysis of the read path.

### Algorithm performance

The compression algorithms' performance is in general asymmetric from the perspectives of compression and decompression. That is mostly the case because during decompression the algorithms have fewer tasks to do, e.g: there's no need to calculate the entropy or redundancy of the data. For that same reason, although there are no technical reports or scientific literature, I would expect decompression to use less amount of resources than compression. Indeed, as a consequence of further characterization of compression algorithms later in this Chapter, some small decompression characterization was also carried out and is presented in Appendix A. The results widely support the hypothesis that decompression is faster and requires much fewer resources than compression. Additionally, at least for zstd, it is known [21] to have a constant decompression speed across all its levels.

In consequence, decompression performance is, in general, both better and more predictable. That should make the decompression process less risky for the overall cluster performance than the compression one while keeping the same benefits. This is also an additional justification to give priority to the compression study. Since the compression process is more critical, its detailed characterization and study should come first.

### Data information

During writes, for the decision-making purpose, the data can be analyzed to extract meaningful metrics, as those described in Section 3.5. However, such information is modified through the execution of compression. Therefore, it is not available at read-time. This information is relevant to knowing the size of the original data once decompressed, and to possibly predict the resources needed for decompression. Since that information is no longer available in the data itself, it must be stored by Lustre or ZFS at write-time and fetched again at read-time.

This leads to the second difference that must be solved during the read process. At read time, the data, and then the information about the compression algorithm used (and potentially the others we have discussed like the data size), lay in the servers. Since the clients have been tasked with the decision-making process, the information must be sent to the clients previous to any decision happening. Currently, reads in Lustre already compromise a pre-fetching step for clients to know which parts of the data are stored in which OSS. Therefore, the compression information could be added to that pre-fetching step. However, that also implies, that clients might not be able to make decisions about a specific read until the whole information becomes available. This can potentially delay the read process and must be taken into account. Additionally, the communication bandwidth is certainly increased by this process and should be considered when further discussing or implementing the read path.

## 4.3. Smart compression

Up to this point in the thesis, a typical Lustre setup has been covered; all of the components involved in the compression flow have been analyzed; and the decisions needed to consider such a setup *smart* have been described. This section will, finally, describe and discuss in detail the proposal for the decision-making process. This discussion will follow the same steps while discussing both the location and the algorithm decisions:

- First, analyze the metrics groups in Chapter 3. I will discuss their importance for each decision and conclude whether they are relevant for this exploratory study.
- For each of the relevant metric groups, the metrics proposed in the corresponding sections of Chapter 3 will be discussed in detail. The goal is to identify which of the

metrics can be used, their importance, and why. I will also cover some preprocessing of the metrics when necessary.

- Finally, I propose a basic calculation algorithm and process that could be followed for the first experimental implementation of the integrated smart decision-making.

The discussion will first approach the location decision. This is a conscious choice and relevant for the development of this section. The location where the data will be compressed or decompressed has a strong influence on the metrics chosen to discuss the algorithm configuration. Resources available in clients and servers might differ and the algorithm decision must take into account only the resources which will be used. Additionally, this allows simplifying the algorithm configuration decision, which is the most complex out of the two, by taking the location-dependent variables out of the scope of the discussion. It is also true that there exists the possibility that the most appropriate algorithm to use varies depending on whether client or server compression is decided. Therefore, a simultaneous decision might, under certain circumstances, be able to take better decisions than the current approach. However, since this is an exploratory work, such an approach will only be considered in future work in Chapter 6.

### 4.3.1. Location decision

To decide where the compression should take place in a *smart* way, the algorithm should take into consideration all the data available. At this point of development, where the assessment of what might or might not be important is mostly theoretical, a methodical approach is needed to avoid unnecessary bias by the author. Therefore, as a first step, every group of metrics in Chapter 3 is analyzed. It is then assessed whether they are relevant or not for the location decision. For the groups that, according to Figure 3.2 (page 20), have instances both in the clients and in the servers, both must be taken into consideration.

CPU and memory both in the client and in the servers are metrics of special relevance. The CPU and the memory are the resources used by the compression algorithms to compress data. For location decisions, many metrics from these groups are of relevance. To be able to make a *smart* informed decision about whether to compress at the clients or the servers, the overall availability of computing resources for both of them must be considered. When only these metrics would be of interest, certainly the location with the most available resources should be the one chosen.

Regarding the network at the client and server endpoints (also known as the memory-network bandwidth), its use can be potentially reduced by compressing in the client (assuming compression of the data is possible and successful). Therefore, high usage of network endpoints at both the clients and the servers could strongly justify client compression regardless of the other metrics. This is especially relevant for the server network endpoints since the shared-resource position of the servers makes them susceptible to changes due to other applications.

On the contrary, storage metrics do not have direct implications on the compression location. The data will go through the storage in the same form regardless of the location

| Metric | Considered | Justification |
|---|---|---|
| Client CPU | Yes | To weight client/server resource use |
| Servers CPU | Yes | To weight client/server resource use |
| Client Memory | Yes | To weight client/server resource use |
| Servers Memory | Yes | To weight client/server resource use |
| Servers Network | Yes | Client compression reduces use |
| Client Network | Yes | Client compression reduces use |
| Storage | No | No direct impact on location decision |
| Application data | Yes | Extremely (in)compressible data has impact |
| IO patterns | No | No direct impact on location decision |
| Cluster metrics | Yes | Compressing in clients helps with bottlenecks |

Table 4.1.: Relevance of metrics' groups regarding compression location for the work in this thesis.

where the data is compressed. And although faster disks would free CPU resources and change the balance between client and server resources, the impact is not direct and hard to study in an exploratory thesis like this one. Therefore, at this point, I do not consider them relevant for the location decision.

Application data can have some influence, especially when data is in very extreme circumstances. For the cases of very compressible or incompressible data, the benefits of compressing in the clients proportionally grow or degrade, respectively. In most intermediate cases, the data might not play a big role, but it is probably still relevant to consider it for the location.

Regarding IO patterns, the data will eventually get through storage regardless of which pattern is used. Still, the IO patterns of different applications running in different clients can influence the ability of servers to execute compression. However, similar to storage metrics, the impact is not direct and complex interactions with CPU, memory, and network metrics are expected. For this reason, and to simplify the discussion, I do not consider the IO patterns relevant for the location decision.

Finally, cluster metrics certainly impact the compression location. These metrics are strongly affected by the trade-off between exclusive and shared-use resources that the location decision has to deal with. Therefore, depending on the whole cluster status, compressing in the client or the servers might be relevant. Additionally, the cluster configuration has an impact on the resources available for the servers, since the more servers available per client, the more beneficial might it be to compress at the servers, where the load gets divided across multiple machines.

A summary of these ideas is available in Table 4.1. However, simply considering the groups of metrics that are relevant is not enough. For the discussion in this thesis to be complete, a more detailed study is needed.

**CPU** Collecting and analyzing CPU metrics have the goal of extracting the effective CPU power available in clients and servers to provide an accurate comparison. Overall, for the location decision, what is most relevant is not the available CPU in either the client or the server, but their *comparison*. The one with the greater amount of resources should be preferred for taking care of compression when looking only at CPU metrics. Ideally, such a comparison would include both the low-level and high-level metrics described in Section 3.1. Unfortunately, the comparison of low-level metrics involves analyzing in detail the used hardware and potential performance differences between different machines, which is out of the scope of this experimental thesis due to its complexity. Therefore, only the high-level metrics are considered.

For the location study, both the static and dynamic high-level metrics shall be considered. The static high-level metrics are a simpler alternative than low-level metrics to measure the overall performance capabilities of different hardware. By taking the number of cores, the standard clock frequency and fetching average number of instructions per clock cycle for the specific architecture, an *approximation* of the number of instructions that a certain hardware could run per second can be obtained. These numbers should help making a fair comparison between the CPU numbers in clients and servers. Out of the dynamic CPU metrics, only the system times considered, as they provide similar information, but are more accurate than the CPU load. Out of the system times, the most relevant one is the *idle time*, since it provides the percentage of the time that the CPUs of a system are available. The final calculation for the available CPU can be found in Equation 4.1, where the frequency, the instructions per cycle and the idle time of each of the CPU cores are taken into account.

$$TotalCPU = freq \times IPC \times \sum_{c}^{cores} \text{idle time}(c) \tag{4.1}$$

Once these numbers are available both for the clients and the servers, they can be used to compare them. The result of the comparison can help decide whether *overall* there are more CPU assets available for a client or its respective servers. However, such comparison requires additional information from cluster metrics that will be discussed below.

**Memory** The memory metrics play a similar role to the CPU metrics described above. They are available both at the clients and in the servers. For the location decision, the interesting point is the *comparison*. In general, memory metrics are of less relevance than CPU metrics, since most algorithms do a less intensive use of this resource than of CPU (see algorithm characterization in Table 4.2). Although that is not a fit-all-algorithms decision, it is certainly appropriate for most workloads. Giving the same relevance to CPU and memory metrics could, on the other hand, bias the decision towards the unfavorable location for most algorithms. To solve this problem completely, the whole decision process would have to be revisited and its complexity increased. Since that is unfeasible for this thesis, it is deferred to future work in Chapter 6.

Since the Memory metrics also have a static and a dynamic part, their use is similar to that of the CPU metrics. However, the static `dmidecode` metrics presented in Section 3.2 are in this case more similar to the CPU low-level metrics than to the static ones. Although some very useful information could be extracted, like the latency and speed of the memory installed, it is certainly not straightforward to translate those to real-world performance [15, 51]. Therefore, since the comparison of the inherent memory performance is simply unfeasible, I will discard it and leave it for future work. Still, dynamic metrics provide very useful information about the memory status of a system. The *free* and *available* values from `/proc/meminfo` [43] are those of greater interest. These two values give some very important information: the amount of memory completely free to use, and the amount of memory that could be potentially used, at the cost of freeing other kinds of memory (mostly different kinds of cache). Ideally, the free memory should be able to fit all the memory requirements from compression algorithms. When that is not the case, but available memory can match the requirements, compression would be possible at the cost of evicting some caches (which the correspondent performance penalties). In the case where only either the clients or the servers have the amount of required *free* memory, those should be the ones chosen for compression according to the memory metrics. Finally, if all the clients and the servers are under strong memory pressure, the location decision is likely irrelevant, since compression will likely not be attempted.

**Network** The collection of network metrics has a different goal to those of CPU and memory metrics. Contrary to those, the relevance of the network for the location decision does not have to be with the comparison between client and server use. Instead, both client and server network metrics have the same impact on the location decision: the busier the network is, the more relevant client compression becomes. For this reason, the metrics of interest are those that allow the identification of the network load. However, special attention must be paid to the situation where the server network load is close to being filled. A very high server load can cause the processing latency to greatly increase, and in case the available load is overwhelmed, some network packages could be dropped. Given the shared nature of the server resources, this could interfere with other applications already running, or with new applications about to start. In consequence, whenever the server network load is close to its maximum, client compression should be a priority.

To identify the effective usage of the network, it is necessary to aggregate metrics from different sources. First, the kernel interfaces through which the data is transferred. Once those are identified, the internal interfaces within the kernel for `ethtool` or `ibstat` (depending on the technology in use) need to be used to obtain the *speed* metric. Such metric is a static metric as discussed in Section 3.3. In case multiple interfaces exist, the total speed can be aggregated to obtain the maximum bandwidth that each machine supports. During runtime, the kernel `rtnl_link_stats64` structure can be fetched for each of the interfaces to identify their use. The relevant fields from such structure are then `tx_bytes`, `rx_bytes`, `tx_dropped`, and `rx_dropped`. The change of the first two (sending and receiving bytes, respectively), together with the information about the speed, provides enough knowledge to assess the effective load of each interface. The last two, instead, are relevant to identify situations where a machine can no longer sustain its network load and is forced to drop packages. Although nothing can be done to fix the situation where some packages were dropped, it is a strong indication that the network is overwhelmed and compression should be shifted to the client regardless of other metrics.

**Application Data** The relevance of the application data regarding the compression location depends mostly on how compressible the data is. Although it is not possible to predict the exact compression ratio for a certain piece of data, some calculation of its entropy is possible, obtaining at least an upper bound on the compressibility of the data. Unfortunately, even with fast and modern entropy functions [19], such entropy calculation takes a non-trivial amount of time. The benefits become relevant when either the data is greatly compressible or incompressible. In the first case, the data should be compressed in the clients, boosting the benefits of reduced bandwidth and processing of the data at later steps in the data flow. In the second case, client compression provides very little improvement, while increasing the CPU and memory use in the clients[1], which could be better used for other client tasks. In this case, compressing in the server might still make sense, depending on how often might the data be used.

However, since executing the entropy calculation to every piece of data, regardless of how and where it will be compressed is too expensive, some solution has to be explored. In Section 4.4, user-provided information will be discussed and could be integrated to improve the efficiency of the decision on when to execute entropy calculations. However, such efficiency is also important regardless of whether user information is available. One possibility would be to integrate some of the possibilities discussed in related work in Section 2.5. Another alternative would be to use a sampling approach, where for every certain number of writes, the entropy of the written data is calculated and the location decision adjusted if relevant. This approach is based on the belief that the application's data does not differ drastically for subsequent writes, which should have to be verified if this approach was considered. In any case, such integration is considered part of future work.

**Cluster metrics** Cluster metrics are relevant in the location decision for two tasks: comparison between clients and servers and getting an overview of the cluster stage.

For the first case, we have already discussed that for the CPU and memory metrics their relevance lies not in their absolute values, but the comparison between clients and servers. Since there is usually not a single server per client, and clients often write to more than one server at a time, the client-server ratio and number of routable OSSs per client are of relevance. For example, the greater the client-server ratio is, the least amount of CPU and memory resources a certain client should be allowed to use from the server. Otherwise, any client could easily attempt to use most of the server resources, disregarding the idea of considering the server as a shared resource. Since these two metrics are necessary to propose any kind of fair comparison between the resources available in the client and the servers, they will be further discussed in the calculation proposal below.

For the second case, the overall network status is of relevance. Although complex and out-of-scope in this thesis, the process of obtaining reliable information would first need

[1]That is not necessarily the case for the modern algorithms (LZ4, zstd) presented in this thesis, which will abort or adjust their processing if their internal entropy coders detect incompressible data. However, from the point of a general discussion where other algorithms could be potentially used, this is certainly true.

to identify potential bottlenecks in the network architecture of the cluster. Then, the free bandwidth at those points (like Lustre routers) shall be sent to the clients, which could make a more informed decision depending on the status (in general, if those points are in high use, client compression becomes more important).

### Calculation proposal

Having the relevant metrics, the goal is to map a set of continuous input values with a variable bias to a binary decision. I propose to simplify such transformation by doing an intermediate transformation to a unidimensional continuous space. There, the bias can be used to select a cut-off point to split the binary decision. Such bias serves as a numeric representation of all the different factors that influence the decision towards the clients or the servers but that are not expressed in the metrics (such as client metrics being more accurate). This concept is expressed mathematically in Equation 4.2, where $n$ is the number of aggregated metrics used for the calculation. Ideally, both transformations would be linear. However, extracting the specific coefficients for an efficient decision-making process is out of the scope of this thesis.

$$\mathbb{R}^n \rightarrow \mathbb{R} \rightarrow \{0,1\} \tag{4.2}$$

It is also appropriate to discuss here some proposals for the calculation of accurate and fair comparisons between the client and server resources. For CPU and memory, the comparison constitutes more than simply comparing the resources in the client with the resources in a specific server, due to the cluster metrics and cluster configuration.

For the CPU, and as a first tentative approach, I consider it a cumulative metric for the data (in the sense that to compress the double amount of data, in general, double the amount of CPU resources are needed). Although that is not always the case (compression performance and resources needed do not vary linearly with the data, for example when performance degrades due to very small chunks being compressed), it is a simplification that can be appropriate for a first experimental implementation. With this CPU consideration, and since server resources are shared, each routable server could, in principle, contribute a percentage of the client-server ratio. A proposal calculation for the comparison is presented in Equation 4.3, where the relationship between the server and the client CPUs is presented as the sum of the CPU resources for each routable server, divided first by the client-server ratio, and then by the client CPU available. A result greater than one for this Equation would favor server compression while a lower than one result would favor client compression.

$$\text{Server-Client CPU} = \frac{\sum_{s}^{routableServers} \textbf{Server CPU (s)} \div \text{client-server ratio}}{\textbf{Client CPU}} \tag{4.3}$$

For compression and decompression, the memory needed has to be with the amount of data being compressed or decompressed in parallel and the algorithms in use. The part corresponding to the algorithms is unknown at this stage since the algorithm has not yet been chosen. However, the part corresponding to the data is. Since the data to be compressed or decompressed would be split among different servers, each of them would, ideally, get a proportional part of the data, and require such memory proportion. Only when all of them have the required amount of memory available is compression feasible in the servers. Such relationships are expressed mathematically in Equations 4.4 and 4.5. In the first one, the free memory from every server must be greater than the proportional requested memory for every client. By multiplying the data size with the client-server ratio, the equation considers that every client should only have access to a fraction of the server memory. In the second one, the equivalent to the CPU Equation 4.3 is presented. The only difference is that the minimum memory for all the clients is considered in the Equation, to make sure servers with extra memory do not compensate for servers with little memory.

$$ServerPossible = \textbf{FreeMemory (s)} > \frac{\text{data size} * \text{client-server ratio}}{\text{routableServers}}, \forall s \in \text{Servers} \quad (4.4)$$

$$\text{Server-Client Memory} = \frac{\min(\text{Server Free Memory}) * \text{routableServers} \div \text{client-server ratio}}{\textbf{Client free memory}} \quad (4.5)$$

### 4.3.2. Algorithm configuration decision

The algorithm configuration decision is much more complex than the location decision. While deciding on a location is a binary problem (client-server), the algorithm configuration has two dimensions: algorithm (in this thesis: LZ4 and its variation LZ4-HC, zstd and its variation zstd-fast, or no compression, but possibly more in the future) and level. Additionally, the relationship between metrics and their impact on algorithms is more complex and less simple to reason than that of the location binary decision. Therefore, before studying the metrics in detail, it is needed to characterize the algorithms in detail. This helps the reasoning and the theoretical discussion over the impact of the metrics in the algorithms.

#### Algorithm characterization

As it has been already discussed in Section 2.2, compression algorithms are usually only characterized by two properties: throughput and compression ratio. That is not only the case in computer engineering but also in science. Literature that discusses CPU and memory requirements for algorithms is scarce [59] or even non-existent. In consequence, I am forced to carry a small performance characterization of the algorithms of interest for the discussion to have a scientific basis. This is also done in the hope that it will be useful for further reference.

For the characterization, I will use lzbench [27] as the benchmarking tool. lzbench is a well-regarded tool (results presented by LZ4 and zstd authors use it [4, 50]) that allows to benchmark a multitude of lossless compression algorithms and supports all of those of interest in this thesis. lzbench execution follows this series of steps:

1. First, it reads the input data into memory and allocates two buffers of memory. The buffers are big enough to hold the whole data so that they can be used to store both compressed and decompressed data.

2. Regardless of the compression algorithm chosen, lzbench does a fast memory benchmarking by copying the original data to the allocated buffers. This process verifies the maximum possible bandwidth when not using compression.

3. Then, the data is compressed to one of the buffers with the first chosen configuration. The measurements taken by lzbench are: the size of the compressed data, the compression ratio, and the total compression throughput.

4. The same process is repeated for decompression using the spare buffer.

5. Finally, the buffers are erased, and, for each algorithm configuration requested, the process is repeated starting with step 3.

For the small characterization in this thesis, I will use the silesia corpus [24]. It is a widely known corpus for testing compression and also the one used in lzbench examples [27]. It contains a mix of data from different sources including books, pdfs, executables, or binary data. However, just running lzbench on the silesia corpus is not enough. lzbench only provides information about throughput and compression ratio, leaving aside the CPU and memory measurements needed for this thesis. Therefore, to measure CPU and memory requirements, the following approach is developed:

- For the memory, I will monitor the process memory usage during the benchmarking of different algorithms. This memory usage compromises the memory needed for the input data, the buffers, and the allocation from the algorithm itself. However, for a fair comparison, the input data and compression buffers shall be discounted from the total memory use. While compressing data of any kind, those buffers are unavoidable and intrinsic to the compression. Their size varies depending on the amount of data that is compressed and handed to the algorithm at a single time but is not part of the algorithms' properties. In consequence, for the memory characterization, I will consider these buffers plus the additional specific memory requirements for each algorithm.

- For the CPU, I consider an algorithmic approach. As it has been discussed in Section 2.2, the throughput considers both the compression time and the time needed to transfer the compressed data to its final storage. During my experiments, lzbench was always able to make the compression algorithms use the full allocated CPU for compression. Therefore, the percentage of CPU use can simply be disregarded, and comparisons of CPU time for a single copy should be enough for a fair comparison. The CPU time is defined as the total time that an algorithm takes to compress the data. Since lzbench provides the size of the compressed data and the throughput for both the compression algorithms and simple memory copies, it is possible to calculate the split of time needed to transfer the data. The total time can also be calculated by knowing the original data size and taking the compression throughput provided by lzbench. The difference between the whole copy time and the time to transfer the data is, in consequence, the CPU time. This process is mathematically represented in Equations 4.6.

$$\begin{aligned} CPUtime &= \text{total time} - \text{transfer time} \\ \text{total time} &= \text{data size} \div \text{throughput} \\ \text{transfer time} &= \text{compressed data size} \div \text{memory throughput} \end{aligned} \tag{4.6}$$

Following the above procedure, lzbench was run on the silesia corpus in an Intel Xeon E31275 CPU. The benchmarking was run on a single CPU and executed for at least 16 seconds to avoid variations, especially for very fast algorithms. Repeated runs of the experiments to capture the memory metrics yielded mostly identical results. The results are presented in a quantitative form in Table 4.2. Specific numeric values are not especially relevant, as they are tied to the data and hardware used for testing. The importance lies in the *relative* comparison between the algorithms. We can observe in Table 4.2 that in general LZ4 algorithm requires less CPU usage with higher throughput, although also achieves lower compression ratios. It is also remarkable that only zstd in high-level configurations (consistently with claims by the authors [21], which warn of high memory use for levels greater than 20) allocates some significant amount of memory, which adds to the memory buffers required to store the compressed data.

Another important conclusion from this run is that, given the much higher throughput for the case where no compression is executed, the throughput and CPU usage are mostly proportional. This is expected, since the lzbench is an in-memory benchmark, but will not be the case in a Lustre cluster. This is useful to characterize the CPU, but when considering the throughput in a Lustre system, it is expected that other delays happen, as will be further discussed below.

### Metric characterization

Once the algorithms have been characterized, the different metric groups have to be evaluated against them. Since the algorithms have been split into multiple properties, the evaluation will be against each of the properties.

However, as we have observed in Table 4.2 and its discussion, the CPU and the throughput are mostly proportional in the benchmark. Therefore, at first glance, it would not make sense to study them independently. In reality, for real-world deployments (opposite to an in-memory benchmark), the effective algorithm throughput between the client and storage in the servers does not only depend on the algorithm itself. Instead, the throughput and bandwidth of the intermediate components play a more important role[2]. If at some point, it were the case that the CPU use is dominating the throughput, it must be because the interconnection and shared resources would be in little use (or otherwise the cluster is likely wrongly dimensioned).

[2]Here I am considering the assumption that CPU processing is much faster than all the other components, and that the CPU-memory connection is much faster than network and storage processing. Since this is one of the motivations for this thesis, it should be a safe assumption.

| Algo | CPU time (s) | Memory (MB) | Comp. Ratio (%) | Throughput (MB/s) |
|---|---|---|---|---|
| none | 0 | 0 | 100 | 8064 |
| LZ4 | 0.39 | buffers + 0 | 47.6 | 509 |
| LZ4 10 | 0.24 | buffers + 0 | 57.4 | 782 |
| LZ4 100 | 0.08 | buffers + 0 | 80.13 | 2120 |
| LZ4 HC 1 | 2.03 | buffers + 0 | 39.54 | 99 |
| LZ4 HC 12 | 22.13 | buffers + 0 | 36.45 | 9.13 |
| zstd 3 | 1.10 | buffers + 0 | 31.46 | 183 |
| zstd 9 | 4.92 | buffers + 10 | 28.28 | 41 |
| zstd 22 | 99.57 | buffers + 640 | 24.86 | 2.03 |
| zstd-fast 1 | 0.60 | buffers + 0 | 41.09 | 329 |
| zstd-fast 10 | 0.39 | buffers + 0 | 50.49 | 496 |
| zstd-fast 100 | 0.15 | buffers + 0 | 75.36 | 1187 |

Table 4.2.: Results from lzbench for compression with silesia corpus on a single core, Intel(R) Xeon(R) CPU E31275 @ 3.40GHz. Memory results are floored to the closest integer.

A smart decision-making process should favor the use of those shared resources instead of the exclusive-use resources from the clients. The hypothesis is that when the lack of CPU power hinders the throughput, that must be due to other resources being free. When that is the case, I expect the smart decision-making process developed in this thesis to shift to greater use of those resources, limiting the impact of the CPU use in the throughput. In consequence, for this discussion, I will not consider CPU effects over the throughput when that is the only possible impact on the throughput for a metrics group. That also means that the discussion on throughput impacts the whole system and not just the algorithm properties.

In the following, for each of the metric groups, I will consider whether there is a correlation between the resource that the metrics measure and each of the compression algorithms' characteristics, or the throughput of the system. This simplifies the discussion points and helps identify the consequences and impacts of each group of metrics:

- **CPU** Regarding the CPU metrics, CPU status certainly impacts the CPU available for the algorithm. Therefore, collecting CPU metrics is a requirement. The CPU is also involved in copying the data to the memory and the network interfaces. However, such processes usually take negligible time or present complex correlations with other metrics in this study. For that reason, such correlations are dismissed and considered to be included within the regular CPU collection process.

- **Memory** Since compression in general and some algorithms, in particular, need more memory than no compressing, memory metrics must be considered. However, memory metrics should not have any other implication in any other algorithm properties. It could be argued that the memory bandwidth must be taken into consideration for the overall throughput. However, results in Table 4.2 show that compression algorithms are not fast enough to saturate the memory bandwidth. Even the fastest algorithm (LZ4 100) only reaches one-fourth of the memory throughput with a very limited 80% compression ratio. Therefore, the memory bandwidth will not be studied, and I do not consider the memory to have an impact on the algorithm's throughput.

- **Network** Network properties are most strongly related to the throughput and the compression ratio property of the algorithms. However, their impact on compression is only relevant when the compression happens in the client. If that is the case, the more compressed the data, the least network resources are overall needed. Additionally, the higher the algorithm's throughput, the more bandwidth is required from the network. Memory and CPU at the endpoints of study (clients and servers) do, however, not see any remarkable impact due to the network properties.

- **Storage** Storage metrics have a sizable impact on most algorithm's properties. Most clearly, the speed of the storage system can be a bottleneck for the throughput. Moreover, the writing process of data to disks requires a non-negligible amount of CPU resources, be it for the filesystem execution or for waiting on IO operations. The effect on the compression ratio is similar to that of the network: the higher the compression ratio of the data, the least the storage needs to be used. There is also a relationship with the memory which applies when the compression happens at the server. ZFS maintains an in-memory cache pool (ARC) which might interfere with algorithms that use a lot of memory and should be taken into consideration. The goals of that pool are to improve the performance of ZFS. Since the ARC can be compressed, the compression ratio also plays a role there. The more compressed the data is, the more will fit into the ZFS cache.

- **Application Data** Application data metrics have a severe impact on the algorithms since the data is the subject of compression. There is a correlation with the memory, since, as we have seen above, the compression buffers' size varies with the amount of data to compress. The compression ratio is also clearly impacted by the data itself since it is the properties of the data that determine the maximum possible ratio. Additionally, the CPU requirements for each algorithm vary depending on the data, since algorithms consume different amount of CPU resources depending on the data properties.

- **IO patterns** The IO patterns reflect how the application reads and writes. For these metrics, I put my attention on the different impacts for sync and async IO patterns. When IO is synchronous, compression algorithms can more freely exploit CPU and memory, but throughput becomes more relevant to avoid long blocking times. When IO is asynchronous, these conditions are the opposite. In general, the more IO there is, the more important it is that compression decisions are accurate and effective.

- **Cluster metrics** Cluster metrics related to the algorithm decision mostly impact the cluster network. In consequence, the throughput and the compression ratio are of

| Metric | CPU | Memory | compression ratio | Throughput |
|---|---|---|---|---|
| CPU | Yes | No | No | No |
| Memory | No | Yes | No | No |
| Network | No | No | Yes | Yes |
| Storage | Yes | Yes | Yes | Yes |
| Application Data | Yes | Yes | Yes | No |
| IO patterns | Yes | Yes | No | Yes |
| Cluster metrics | No | No | Yes | Yes |

Table 4.3.: Relevance of metrics' groups regarding algorithm characterization properties.

relevance, depending on the bandwidth and general availability of intermediate points. This is too complex to study here, so despite this impact, I will not discuss cluster metrics further for the algorithm configuration decision.

A summary of the relationship between metrics and algorithm properties can be found in Table 4.3. In the following, the discussion will increase its depth and consider the implications of the available metrics in the algorithm configuration decision.

**CPU** The CPU metrics aim at knowing the total power available for the compression action. As we have seen during the characterization of the algorithms in Table 4.2, different algorithms require different amounts of CPU resources. Depending on the resources available, some algorithms might completely block the CPU for a long time, potentially harming the overall performance. Others could make use of a very little portion of the CPU, leaving too many unused resources that could otherwise be used to improve the overall performance, e.g: by increasing the compression ratio with a more powerful algorithm.

Therefore, the overall aim for the CPU metrics is to obtain the CPU power available in the machine that will do the compression, and match it against the CPU requirements for every algorithm for the decision-making. The first part can be achieved with a very similar process as that considered for the CPU in the location decision. By considering the number of processors, their frequency, and the amount of time each of them is idle, the CPU power available for a machine can be obtained. This is expressed mathematically in Equation 4.7.

$$CPUPowerFree = freq \times \sum_{c}^{cores} \text{idle time}(c) \tag{4.7}$$

However, matching this free amount of CPU power against each possible algorithm and accurately predicting the amount of time that each algorithm might take is complex. For the compression algorithms used in this thesis, their performance and resource requirements are highly dependent on the data. In consequence, some estimation of those resources would be needed. Some approaches to solving this problem exist in the scientific literature, either through heuristics [75, 44] or, more recently, with Machine Learning approaches [58]. Those approaches could be considered and compared to decide on the best-fitting one for the use case in this thesis. Then, the chosen approach could be applied as a pre-processing step whenever the resource use varies enough for the algorithm configuration to be worth reconsidering. A more detailed consideration of this issue is discussed under the Application Data metrics.

**Memory** The memory metrics are relevant since compression requires additional memory that otherwise would not be needed. The only relevant memory metrics are the free and available memory values, which are already discussed in Section 3.2 and Subsection 4.3.1. According to the amount of memory needed, the results in Table 4.2 show that there are only 3 possible scenarios for all the different algorithm configurations:

- When no compression is executed, there are no memory requirements and the data can be sent to the storage without any copies.
- For any kind of compression algorithm, buffers with at least the size of the compressed data must be allocated. Their exact size depends on the size of the data compressed at once and could be potentially relevant when compressing big chunks of data.
- For the zstd algorithm in its regular configurations, high levels (above 9 and especially over 20) request growing amounts of memory, additionally to the compression buffers.

The relevance of the memory metrics for the algorithm-configuration decision is mostly for the *enablement* of certain configurations. The available memory resource is a very important asset for any computer system. Contrary, for example, to the CPU use (where a lack of resources will simply slow down execution but not yield any further consequences), a lack of memory can have catastrophic consequences for other programs running on the same machine. From the concept known as *trashing*[3] to invoking the Out-Of-Memory killer which kills programs until enough memory is available. In consequence, when there is not enough memory free for some of the previous scenarios, the corresponding algorithms can simply not be used.

However, there is still the question of which of the two memory metrics (free and available memory) is better fitted for the comparison against the different scenarios. There exists a trade-off between those. When there is not enough free memory, but there is enough available memory, caches, and other in-memory data must be freed to use the available memory. Although hard to measure, it is expected that freeing such information could have some impact on the general system performance (the same happens if the ZFS ARC has to be shrunk, but that discussion belongs to the storage metrics). I propose to first use the free memory for the comparison. When the free memory is not enough, but other parameters (network, storage, CPU use) support the use of some compression algorithm, then the available memory could be used for the comparison. The hypothesis is that, when compression is beneficial, any performance penalty for dropped caches is certainly worth it.

[3]Trashing is the circumstance where a system starts using storage hardware as main memory due to the lack of the last one being available, dramatically slowing down the processing.

**Network** Contrary to the CPU and memory, the impact of network metrics on the algorithm configuration decision-making process is less straightforward. The compression algorithms have an impact on the network in two ways:

- The compression ratio, together with the data size, determines the total amount of data that will be sent. In other words, it has a direct impact on the total amount of network resources that an algorithm will use (regardless of the speed at which such use happens).
- The algorithm throughput determines the maximum speed at which any algorithm might try to write to the network. In practice, it is unlikely that any network can achieve enough bandwidth to support the algorithm throughput. Especially when dealing with fast algorithms like LZ4 or zstd-fast and in the server-endpoints, which are a resource shared by all the clients.

In consequence, the metrics to collect are the network bandwidth and its current use in all the end-points of interest.

To do so, first, the relevant endpoints need to be identified through Lustre's networking configuration. For this thesis, however, I simplify this process and consider all the end-points as equally important and relevant. Once they are identified, the maximum possible bandwidth can be extracted by querying the *speed* property for the corresponding driver in the kernel. This can be done by mimicking the work carried out by tools like `ethtool` or `ibstat`. Once the maximum speed is identified, its current use can be queried through the `rtnl_link_stats64` structure, as done for the location decision. The same relevant fields as in the location decision are of interest: `tx_bytes`, `rx_bytes`, `tx_dropped`, and `rx_dropped`. The first two allow obtaining the use of the interface, by calculating deltas over some period; while the last two are needed to identify the over-saturation of an interface.

Once the required metrics are collected, and the relationship between the algorithms' behavior and the network is known, the question remaining is: How do the metrics impact the algorithm configuration decision-making? Since one of the goals of this project is to improve the resource usage, the first step to the decision-making would be to choose any algorithm able to saturate the network. Out of those able to make the maximum use of resources, the network metrics should favor the one with the highest compression ratio. The rationale for that decision is that the higher the compression ratio, the least overall network resources will be used.

However, there is one important point to clarify when making the comparison between algorithm throughput and network bandwidth. As we have seen in Equation 2.3, the throughput of the algorithms is measured in bytes of *uncompressed* data, while the bandwidth considers the real amount of data traversing the network. Since data being transferred is compressed, a transformation from one to the other must be carried out. The transformation consists simply of multiplying the throughput for the compression ratio to obtain the speed of transfer for compressed data, as can be seen in Equation 4.8.

$$\begin{gathered} NetworkBandwidth = AlgorithmThroughput * CompressionRatio \\ CompressionRatio < 1 \end{gathered} \tag{4.8}$$

| Algo | Comp. Ratio (%) | Throughput (MB/s) | Bandwidth (Mbps) |
|---|---|---|---|
| none | 100 | 8064 | 64512 |
| LZ4 | 47.6 | 509 | 1938 |
| LZ4 100 | 80.13 | 2120 | 13590 |
| LZ4 HC 1 | 39.54 | 99 | 313 |
| zstd 3 | 31.46 | 183 | 461 |
| zstd 22 | 24.86 | 2.03 | 4 |
| zstd-fast 1 | 41.09 | 329 | 1081 |
| zstd-fast 100 | 75.36 | 1187 | 7156 |

Table 4.4.: Results from lzbench for compression with silesia corpus on a single core, Intel(R) Xeon(R) CPU E31275 @ 3.40GHz. Throughput and compression ratio are used to calculate the theoretical network bandwidth which would be required.

Just as an example, a simplified version of Table 4.2, with the addition of a bandwidth column can be seen in Table 4.4. The bandwidth has been calculated by the use of Equation 4.8 and later transformed to bytes-per-second (bps), which is the standard dimension for network hardware. The transformation between the dimensions is carried out by simply multiplying by 8, given $1byte = 8bits$. To provide some perspective to this example, the machine used in the experiment has some network connectivity with a maximum 1Gbps or 1000Mbps. In consequence, even with a single CPU core in use, all the algorithms portrayed but LZ4 HC 1, zstd-3, and zstd-22 would saturate the network. In the more real-world scenario where all the 8 cores are used for compression, and the throughput possibly multiplied by 8, all the algorithms presented but zstd-22 would saturate the network.

Unfortunately, at the decision-making time, the throughput and compression ratio that every algorithm can provide is still unknown. This problem has been already discussed and could be solved through heuristics or other approaches like Machine Learning [58]. This discussion is disregarded here for simplicity.

Finally, it is important to mention that the network metrics in this decision are only relevant when the compression happens in the client. When the location decision analyzed the network metrics and decided to compress in the server, the network metrics stop being relevant for the configuration decision. If the compression happens in the server, the data will always flow uncompressed through the network and the use of different algorithms does not have any impact.

**Storage** The storage metrics have some commonalities with the network metrics. Since the data has to be moved to storage, the previous rationales of the relationship between the required bandwidth and the algorithms' throughput and compression ratio still apply. However, storage metrics have some key differences:

- Writing and reading to storage requires a measurable amount of CPU time, which is even tracked by the CPU metrics. It might be worth using a powerful algorithm that steals a lot of CPU if it has a high ratio and then reduces the IO time, including the CPU costs.

- The actual amount of data transferred (and hence the total bandwidth to the storage) is often greater than the logical amount of data to be transferred. That is a consequence of redundancy often needed to withstand drive failures, which is the standard in ZFS configurations.
- The nature of the storage system is less deterministic than that of the network system, which makes estimates harder. Drive performance (especially rotating Hard Drive Disks) varies depending on hard-to-analyze parameters, like the distance in disk from one read to the following one.
- The storage system will always be traversed, regardless of whether the compression happens in the clients or the servers, and the storage metrics are always relevant for the algorithm configuration decision.

Unfortunately, the characterization of the storage system is also more complex than that of the network. Even a minimal ZFS setup like the one presented in Figure 3.5 contains 8 entities with multiple interactions between them. The complexity of the study of this problem makes it unfeasible to discuss it in-depth in this thesis and is considered out-of-scope. For any further discussion, I will assume one single disk is in use exclusively by ZFS for storage. This also simplifies the problem remarked above of the difference between the logical and real amounts of data transferred.

However, even with that great simplification, the effective calculation of the maximum possible bandwidth to the storage system is quite complex. This metric would be the equivalent to the network speed that was explained in the network metrics above. It would be very useful to obtain a clear picture of the power of the storage system to make informed decisions regarding compression. The reality is, that contrary to the network speed, which has some very clear specifications that can be consumed, the storage speed is non-deterministic. Characterization of storage hardware, even by manufacturers [65, 66] includes metrics that are dependent on the workload (sequential or random access [66]) or approximations or means (Maximum Sustained Transfer Rate [65]). The reason for these problems is related to the specifics of the storage technologies, or how data is read and written. A discussion of this problem is tangential to this thesis and considered out-of-scope. But its consequences are not. Given that even manufacturers rely on benchmarks to support claims about the performance of their devices, such is probably the most appropriate solution. Luckily, the information about the ZFS configuration speed is a static metric as long as the configuration is not modified. Therefore, the benchmarking would not have to run often. For the benchmarking purpose there exist a variety of tools, e.g: fio [9] or bonnie++ [16] any of which could be leveraged for this purpose.

Once the process to obtain the maximum storage bandwidth has been determined, its current use is also needed. For that purpose, the main source of reliable data from the kernel comes from the statistics in `/proc/diskstats`, which can also be obtained directly from inside the kernel. The most valuable statistics are:

- The number of sectors read and written. With its delta through some period, they allow calculating the average bandwidth for read and write operations. Together with the expected bandwidth obtained from the benchmarks, it would allow calculating the current use of the storage system.

- The number of IOs in progress and the milliseconds spent doing IOs. It provides information on the pressure of the storage system. If the number of IOs is very high, or steadily or rapidly growing, the IO storage is under pressure, which can lead to a slow performance, that could be improved with greater compression.

In principle, the `iowait` statistic from the CPU could directly be used to obtain the time the system spends doing IO to the storage. However, most reads and writes are executed asynchronously in the kernel. Therefore, while a processor is waiting for the IO to complete, it is allowed to context-switch and execute a different task. Consequently, the IO wait time is only relevant when the CPU use is mostly in an idle state. It is certain that under a higher load, part of that IO wait time could be used for processing instead, with the IO wait time reduced without any other relevant differences.

Overall, the decision-making process gets affected by the status of storage metrics related to CPU, compression ratio, and throughput in the following way: For a certain workload, the algorithms able to saturate the bandwidth should be preferred over those too slow to make maximum use of the storage. From the ones that can saturate it, those with the better compression ratio, to reduce the use of the system, are preferred. This is, in general, the same rationale as for the network metrics. However, for storage metrics, when compression happens in the server, the IO wait times in the CPUs have to be considered. In principle, most of that IO wait time could be consumed asynchronously by executing tasks while the processors are waiting. That should allow leveraging at least a part of the waiting time to increase the CPU power available for compression.

In addition to all the above, the memory also has an impact on the storage metrics. ZFS will perform better the more cache it has available. In any circumstance, at least one part of it will be allocated in-memory. When the compression happens in the server, any shrink of such cache might reduce the storage performance. Weirdly enough, the memory used by ZFS does not count within the used memory in Linux (hence, it is considered *free* memory). This has to be taken into account when considering compressing in the server, as an additional use of the memory. However, similar to the free and available metrics for the memory, there are also different metrics in the ARC that should be considered differently. In ZFS, the ARC contains both filesystem metadata and caching of regular pages. Although the eviction of regular pages could have some impact, the eviction of the filesystem metadata is of much more concern. Therefore, if some algorithm that uses a lot of memory is requested at the cost of reducing the ARC size, it must make sure that the metadata (which is the last part of the ARC which will usually be evicted) size of the ARC is always respected.

**Application Data** The data metrics are of great impact on the general behavior of the algorithm. Most remarkably, the data, together with some hardware static metrics (such as the type of CPU or speed of the memory) is what determines the CPU resources a certain algorithm will need, and the compression ratio it can achieve. Therefore, a correct analysis of the data could help solve the issues outlined in the other metrics' groups relative to the prediction of resource use. In this sense, application metrics are different (at least in what concerns CPU and compression ratio; memory, as discussed below, is the same as other groups) from the metrics discussed above, in that they do not directly impact the decision-making. Instead, they are *enablers* to improve the prediction for other resources' use, potentially increasing the accuracy of the decision-making.

The goal is to use the information about the application data (mainly, the entropy metric) to predict both the compression ratio under a certain algorithm and the processing resources that the algorithm might need. To integrate such a solution into the system of study in this thesis, I propose two possibilities:

- Use a stream-based real-time method that can analyze the data without a measurable overhead. Although there are a few stream-based solutions [70, 75], they are not widely adopted and integration and correlation of their results with the chosen algorithms for this thesis can be challenging.

- Use non-real-time methods through sampling to avoid introducing excessive overhead. This would provide a lower resolution than real-time methods, but at the same time, an easier correlation. On the hypothesis that the kind of data written by a common scientific application does not vary wildly through its processing, such a sampling-based approach would remain accurate. Some research on these solutions provides two alternative approaches:
  - Use fast and modern entropy calculation methods [19], or even directly attempt partial compression of the data with some of the available algorithms.
  - Use modern Machine Learning (ML) approaches [76, 46, 58] that could provide information on the expected outcome for different algorithms. This modern research usually aims to directly recommend an algorithm to use. Such a result is not appropriate for the use case in this thesis, since the decision must incorporate knowledge from other metrics. Instead, the ML approaches should be modified to provide expectations for the runtime and compression ratio of the algorithms. This makes this implementation more challenging.

These results could be then leveraged for a more accurate comparison of the decision-making related to other metrics.

Regarding the relationship between data metrics and the memory, the amount of allocated data depends on the size of the data and the parallelism being used (normally, we could consider this the number of CPUs). Therefore, when considering the amount of memory needed for any algorithm, such comes from the amount of data compressed at once. For those algorithms that require additional memory, there is unfortunately not a published characterization of their relationship with data metrics. Therefore, the requirements for the data size are unknown and would need to be part of future work.

**IO patterns** The IO pattern metrics do not have a direct impact on the algorithm but only affect the decision-making process. Within the IO patterns, two situations depending on the synchronicity of the IO can be identified:

- When mostly asynchronous operations are being executed (due to a lot of `MPIIO_NB` counts and a lack of `SYNC` calls), the use of resources must be careful. It is common that with asynchronous IO, applications use a greater amount of resources since they can compute and do IO at the same time. However, it could be that at a certain point in time, the application is not able to fully utilize CPU and memory resources. In that situation, smart decision-making should avoid utilizing all the free available resources, even if that is its original goal. Instead, it should leave some buffer of used resources. It would be possible that the application increases its resource use at a later point in time. If that happened, but the CPU and memory resources were already completely in use, the application could stall or run the node out-of-memory, which must be avoided.
- When mostly synchronous operations are being executed (identified to a lot of `SYNC` calls), the IO operations will block any further processing. In these circumstances, it is expected that the CPU and memory resources are fully available for compression. In consequence, the decision-making can make more aggressive decisions and consider algorithms with greater requirements, but also higher compression ratios. However, it must also be considered, that the longer the application is waiting for compression, the longer it will take to continue with its work. For that reason, the best algorithm combination would be that which can saturate the network throughput, at the greatest compression ratio possible.

Both of these situations and their importance in the pipeline is determined by the amount of IO taking place. By using all the `READS` and `WRITES` calls for the different methods, the amount of the IO can be computed. The greater those values are, the more resources shall be left as a buffer in the asynchronous case. Equivalently, the greater the IO, the more important it is to choose a fast algorithm in the synchronous case. Otherwise, the possibility of stalling the application increases.

Another relevant IO metric to consider is the amount of `RW_SWITCHES`. It is not something uncommon that due to the huge amount of data generated by scientific applications, some or most of it must be temporarily committed to storage. When that is the case, the data written will be subsequently read. If this workflow can be identified, the decision-making process must consider fast algorithms both for compression and decompression, avoiding long waiting periods to write high compression ratio data that will also take a long decompression time.

#### Calculation proposal

Contrary to the location decision, the mapping between metrics and algorithm configuration is not as simple as a transformation to a linear space and a cut-off point. First, algorithms have a multi-dimensional characterization (as described in Subsubsection 4.3.2) that cannot be represented in a linear space. Second, the final decision is not a simple binary decision that can be decided with a cut-off point (or even a simple classification in a higher-dimensional space). Instead, it compromises the decision over multiple algorithms (not just the ones in this work, but new algorithms that could potentially be added) with different configurations.

In consequence, a similar but more complex approach than the location decision could be considered. Such an pproach would present an intermediate high-dimensional space and a final discrete space which represents each of the possible configurations. The mathematical representation can be seen in Equation 4.9. In this Equation, $n$ is the number of metrics and $m$ the size of the algorithm properties, which is much lower than the number of metrics. Finally, $\mathbb{X}$ represents the discrete space of all the possible configurations.

$$\begin{gathered} \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{X} \\ n \gg m \end{gathered} \tag{4.9}$$

The problem for such an approach is to find the appropriate transformations, especially between the $\mathbb{R}^m$ and the $\mathbb{X}$ spaces. That transformation is also influenced by some metrics, like the application data or the IO patterns metrics. This is certainly a very complex problem that does not have a single solution.

An alternative approach could be to train an algorithm on real-world data to execute the classification task. It would have the benefit of being robust against different kinds of inputs but would require an extensive amount of data which is not available at this moment.

In consequence, this problem cannot be solved within the scope of this thesis and it would be the responsibility of future work to solve this problem.

## 4.4. User-guided decisions

Now that the core of decision-making is in place, the information that users or user programs could share with Lustre can be included in the process. This can potentially improve the decision-making process and allow for the decision-making to be more tailored to the user's needs. At the same time, it keeps the flexibility to adapt the decisions to the overall cluster environment. This section contains an initial discussion on which information to be provided could be useful and why. For some of the most useful hints, some greater discussion will take place. It should be noted that the basics for users and programs to provide this information to Lustre are already in place through the `ladvise` interface. This interface already exists in Lustre for certain purposes and has been showcased to be extensible enough to also support this use-case in Subsection 2.1.2.

The integration of external knowledge into the smart decision-making is important to provide information that is otherwise lacking. From all the previous discussions, the basis for data collection and smart use has been set. However, data collection is limited to analyzing the present and the past. Information about the future is completely unavailable and can only be guessed. Allowing users to guide the smart decision-making is one way to provide such missing future information. Additionally, users might want a higher degree of control over the behavior of the machines they are using, or simply provide extra information for the decision-making process. The additional information can be beneficial in many situations, especially when the user is trusted.

For the actual integration of this information, and given the previous discussions in Subsections 4.3.1 and 4.3.2, the information from the user hints can simply be a bias in the final decision. Such bias might be of different importance depending on how much a specific user is trusted, which could be an administrator configuration.

Still, it is important to mention that all the discussion in this section is mainly based on a thought experiment. There is no scientific base or previous work on which to base the discussion. It is mainly a collection of thoughts and ideas for different kinds of bias that could be applied to the metrics and the decision-making process. For this same reason, it is not and neither aims to be exhaustive.

However, before going further into the discussion, the hints will be categorized for an easier study. Currently, the `ladvise` interface is designed to be used both by people (through a user-space command-line program [3]) and by programs (through a well-documented library [2])[4]. Both interfaces allow for the same information to be provided to Lustre. However, it is expected that often people and programs have different levels of knowledge about the program execution. In general, the people making use of HPC clusters are scientists with a narrow knowledge of computer science and the computer science details of the programs they run. It is expected that most of these users can only provide or request high-level information, but otherwise lack detailed knowledge. On the contrary, programs that make use of the `ladvise` interface can be assumed to have more in-depth knowledge. Some of those programs already exist [58] and the process described here could simplify its integration into the whole process. However, this distinction is done for the mere purpose of simplifying the discussion. Expert scientists or simple programs can always make use of all the hints since they will all be exposed to the same interfaces. Therefore, in the following, the discussion will be split among high and low-level hints.

### 4.4.1. High-level hints

This is information that non-computer scientists could be expected to provide to a decision-making algorithm. This information has been organized into three groups:

- **Resource use prioritization** Although users might not be very knowledgeable about their applications, they might have some preference on how they want the system to be used. For that reason, I propose three alternatives:
  - Maximize resource usage. This would usually be the algorithm default and its goal would be to maximize the use of resources to extract the greatest performance possible.
  - Efficient resource usage. Would try to avoid extreme uses of resources at a small performance improvement. For example, high zstd levels could be avoided, using LZ4 HC instead. Some compression ratio is traded for less CPU and memory usage.

[4]For simplicity, they will both be, in the following, referred to as *users*.

  - Low-power resource usage. The power usage in HPC has been proven to be mostly a consequence of CPU and IO usage [28]. Fast algorithms which limit the CPU use, but still can compress the data and reduce the IO would be preferred in this case.

- **Data information** When users lack advanced information about their data, they might still be able to provide useful information for compression. In this case, I propose a two-fold classification, taking into account the typical users of HPC systems:

  - Classification based on the type of data. This information is usually much more accessible for users (that might be experts in particle physics but not necessarily in Computer Science), which might at least know if their results are numeric calculations, big text files, huge CSVs, or binary files. This information can be used to improve compression results. For example, text files will, in general, compress much better than binary files, and an aggressive algorithm might not be needed to achieve a high compression ratio.

  - Classification based on the field of study. In the worst-case scenario where users cannot provide the type of data they are using, they might still be able to provide the field of their work. Based on literature [60, 28] and conducting further studies, it might be possible to associate certain fields of study common in HPC computing with the compressibility of the data they produce. For example, it can be observed in the work of Promberger et. al [60] that different datasets of lead-lead and proton-proton collisions produce similar average compression ratios across a wide variety of algorithms.

- **Data usage** The further application of the data being written is of interest to decide on how to compress. If the future usage is known, it helps to make better, more informed decisions. I propose three alternatives:

  - The data is only written for legal or compulsory reasons. That could be the case for some parts of data for some applications. In these situations, the data is not expected to be ever written. Therefore, using the algorithm with the highest compression ratio available, even if it were slow both to compress and decompress could be preferred. In the end, the data is never expected to be read.

  - Data will be analyzed in detail and used a lot. That might be the case for experimental results that must be analyzed. In these cases, the data is expected to be read many more times than it was written. Therefore, the impact on the decompression ratio must be considered and take preference over the possible overhead of compression.

  - Data will rarely be used, for example, when running preliminary experiments or exporting short-living results. In such cases, the data might be used a few times and likely later discarded. An equilibrium between compression and decompression should be the goal in this circumstance.

### 4.4.2. Low-level hints

This is information that expert users or programs designed by them could provide. These could be hard-coded in applications designed to be run with Lustre, or by other applications running in parallel. Previous work on adaptive compression [58] has been designed mostly with this purpose in mind. Having a program with knowledge about both the goals of the scientific application running, and the data it exports, can be of great utility. When that were the case, this solution overcomes most of the issues presented in Subsection 4.3.2 regarding the prediction of algorithm behavior. Therefore, such a program could work as a complementary decision-maker to the decision-making process presented in this thesis. However, not every program able to provide low-level hints might be of that great quality. Therefore, a trust parameter to measure this contribution would still be appropriate. The possible hints proposed in this section are:

- **Algorithm selection** Allow users to directly provide the best fitting algorithm for the application being run. This is certainly of great use when using applications that run in parallel to the decision-making proposed in this thesis. Still, the decision-making process can decide to ignore the recommendation when the overall cluster status suggests a different solution.

- **Location selection** Users might have a preference to use exclusive or shared-use resources and might want to enforce it. Again, this could be overwritten if the overall cluster status suggests the opposite solution.

- **Expected workload** Whether the application is especially heavy in the use of one specific resource (CPU, memory, IO). This would be good information for the decision-making process to better adjust to temporary circumstances. For example, if a program hard on memory suddenly frees a part of it, the smart decision-making might still avoid zstd-22, as it would expect the memory consumption to grow again.

- **Workflow information** Specially regarding IO patterns. It would improve predictions on how might the application behave during IO. If a program can provide the expected behavior during reads and writes (is the application using a measurable amount of resources?), it could improve the decision-making choices.

- **Data information** Similar to the high-level hints, it would be beneficial for the decision-making to have additional information about the data. Since the high-level hints covered basic information that users can provide, here more advanced knowledge is covered:
  - Information about the compression status of the data. It is not uncommon that scientific programs output data with some kind of compression, e.g: zero reduction algorithms. Since such compression might be redundant to the following compression process by the smart algorithm, having the information could save time in the overall process.

– Information about the compressibility of the data. Even if the data is not compressed, the program which generates the data might know the amount of redundancy available and its expected compressibility. When trusting this information, entropy extraction might become unnecessary.

# 5. Results and discussion

After a lengthy but mostly theoretical discussion about metrics and their impact on compression for a Lustre cluster, some experiments are needed to verify and challenge the discussion. Contrary to a typical thesis, in this exploratory thesis, the experimental section happens only after the discussion. This is necessary due to the lack of previous work in the field of compression in HPC and to be able to produce some proper assumptions before the experimentation.

For this same reason, there is not a fully-ready system where end-to-end testing is possible. Such testing would require writing to a Lustre client, such client taking the compression decision, compressing if needed, and sending the data to the server. Then the server would compress the data if requested and hand it to ZFS for storage. Finally, any other client should be able to request the data, decide whether it is to be decompressed in the server or the client, ask the server for it, and verify the data integrity after it has been read. Unfortunately, many of the building blocks still have to be polished, including better integration and some bug fixes that prevent some use-cases, or simply developed (the read path is currently unimplemented). In consequence, for this thesis I will only *emulate* a part of such end-to-end testing:

- Only the compression path will be tested. The compression process concentrates most of the decision-making choices since during decompression the algorithm configuration cannot be selected. It also avoids additional effort in dealing with the read of the data and its integrity.
- Server-side compression is not possible. Doing so would require a further Lustre integration since there is no user-space integration on the server side.
- For the experiments and writing data, I will use partdiff-IO, a CPU-intensive program used to calculate differential equations and modified to generate a sizable amount of IO.

Following this process, some exploratory experiments will be executed. Those will try to challenge some of the basic hypotheses in this work and get answers to basic questions related to them. The hypothesis that will be challenged are:

- **Compression is always beneficial** One assumption at the core of this thesis is that compression is of great utility in a Lustre setup. However, it could also potentially harm the applications. To validate such an assumption, two important questions need to be answered: can compression harm the application? How much can it harm it? To answer these questions compression must be run in unfavorable conditions and the results evaluated against an uncompressed workflow.

- **Compression can improve the application's performance** Regardless of the previous questions, it is expected that compression can, at least in some cases, improve the overall performance. The expectation is that shifting part of the computation costs to the client can leverage a better use of shared resources. Especially for applications that do not do great use of all the resources available. For this hypothesis, compression can be executed in a scenario of limited availability of shared resources (most easily exploitable are servers and the network). It could then be compared against a scenario without compression, which should provide worse results.

## 5.1. Experiment configuration

Before going into the experiment execution, it is important to clearly explain how those experiments are run. This compromises the application used for testing (partdiff-IO), the overall setup and its configuration, and some preparatory steps to adjust the result-gathering process and the application parameters.

### partdiff-IO

The application chosen to simulate scientific applications running in Lustre is partdiff-IO. The original program, partdiff, takes over a matrix and calculates a set of differential equations on it. It was originally designed to showcase the possibilities of parallelizing computations across a cluster of machines. For that, reason, it supports two calculation methods to solve the differential equations: Gaus-Seidel, and Jacobi. The first one modifies in-place the matrix, while the second one allocates two matrices, and writes the results of each iteration to alternative ones. Out of those, only the Jacobi method can be efficiently parallelized and is thus the method used in this thesis.

Other relevant parameters for partdiff are: the *interlines*, which determine the size of the matrix according to Equation 5.1; the *inference function*, which can simply represent the mean or some sinusoidal equation; and the termination process and its value, which can be either a set precision for the results or a fixed number of iterations.

$$MatrixSize = Interlines * 8 + 9 \tag{5.1}$$

Before the development of this thesis, the application was modified to execute IO intensively, renaming it to partdiff-IO. It makes use of OpenMPI [69] to work as a cluster and synchronize the IO. In its current form, each of the machines involved in the calculations calculates a part of the overall matrix. Then, according to the added *save-frequency* parameter, each machine writes its part of the matrix to a common file.

Additionally, for the development of this thesis, several modifications were done to partdiff-IO:

- Before writing, it is now possible to compress the data with either LZ4 or zstd, selecting the desired compression level with the new *level* parameter. Negative levels

| Option | Consequence |
|---|---|
| Interlines | Directly proportional to the matrix size. Controls the computation cost and the data size |
| Iterations/Precision | Can be used to tune the total run-time and the compression ratio by the end of the experiment |
| Save frequency | Frequency to send the matrix to storage. Tunes the IO/computation ratio |
| Save delay | Controls the first iteration at which writing is allowed. Controls the compression ratio at the beginning of the experiment |

Table 5.1.: partdiff-IO tuning

when compressing with LZ4 request the use of LZ4 HC, while negative levels when compressing with zstd request the use of the zstd-fast alternatives.

- It is also now possible to delay the occurrence of the first write a certain number of iterations through the *save delay* parameter. This was necessary to avoid artifacts in the results. In the beginning, the matrix is filled with zeros, and huge compression ratios could be achieved, polluting the results.

In consequence, there are quite some parameters to be modified. However, for the work developed in this thesis, not all of them have the same impact. Those with a direct influence on the result can be seen in Table 5.1.

It is also important to remark that in the experiments, partdiff-IO can be instructed to work in a slightly asynchronous manner. Since for all the experiments the Jacobi method will be used, there are two matrices available. Therefore, it is possible to ask the application to continue the computation for one single iteration, using the matrix which is not being written. Although there should not be a big difference in a single iteration run, the mostly synchronous but slightly asynchronous process can resemble the reality for some HPC programs. Alternatively, partdiff-IO can also completely block from the beginning of the writing. One option or the other will be used when most appropriate.

### The setup

For the experiments in this thesis, I was provided with 10 identical machines to set up a Lustre cluster. Each of them is fitted with an Intel(R) Xeon(R) CPU E31275 with 8 cores at 3.40GHz frequency and 16GiB of RAM. They have one network connection with 1GbE each living in a shared network with greater speed (the exact speed of the network is unknown, but from the experiments able to sustain at least 3Gbps which is the maximum bandwidth tested). There was also an additional network interface for management tasks which does not interfere with the experiments.

The machines came with a CentOS 7 installation. Such an OS setup, although quite old, is the best-tested one for the Lustre fork used in this thesis. The said fork contains the `lmetrics` module designed and implemented in Section 3.8. A Prometheus [61] node exporter is installed in each machine and takes care of forwarding the metrics from the `lmetrics` module to a log collector outside the cluster. Later, Grafana [32] is installed also outside the cluster to be able to visualize the results.

During the tests, I initially configured 4 machines to work as servers and 6 to work as clients. Unfortunately, some bug in the OpenMPI library or the setup prevented more than 3 of the clients from reliably communicating correctly for the process. In consequence, the experiments were always executed with 3 clients and a varying number of servers, depending on the hypotheses being tested.

## Experiment preparation

Before executing the requested experiments, it is needed to verify the usefulness of the application being used. For that reason, some preparatory runs were executed. In those, the application was run with a varying number of parameters to identify appropriate values for the experiments:

- For the matrix size, multiple experiments were run, mostly analyzing the memory in use. Since the computation-to-IO ratio can be tuned with the save-frequency parameter, the CPU use was not considered relevant here. After some analysis, I chose 2048 interlines, which represents a matrix size of 16393x16393 *doubles*. This provided a total matrix size of approximately 2GiB, out of which one-third will be allocated to each of the 3 clients. Since partdiff-IO keeps 2 copies of the matrix in memory, that corresponds to a sizable amount of memory (approximately 10%), while still providing plenty of memory for compression. Additionally, transferring such an amount of data through the network requires a non-negligible amount of time: approximately 16 seconds at 1Gbps (maximum cluster bandwidth with a single server) and 16 at 3Gbps (maximum cluster bandwidth with 3 clients and 3 or more servers).

- For the number of iterations where writing should start and finish, I analyzed the compressibility of the data after a different amount of iterations. Given than the matrix gets filled from the borders to the inner values, the minimum iteration at which the matrix did not show huge differences was 20 000. The results can be seen in Table 5.2 with up to 40 000 iterations, where the compression ratio starts approaching 0.9, meaning data is mostly incompressible. Remarkably, the differences in compression ratio are considerably smaller that those from the characterization seen in Table 4.2 in Chapter 4. The most plausible explanation is that the output data from partdiff-IO consists purely of floating-point numbers, where algorithms see less variance than, for example, in the text data available in the silesia corpus. Different iterations will be used for different experiments.

- For the write frequency, I experimented with different values to obtain an appropriate computation-to-IO ratio and a good visualization. The results can be seen in Figure 5.1, where green lines correspond to client resources and red ones to server resources. There, three consecutive runs of partdiff-IO in its slightly asynchronous version are executed. The runs are done without compression and a configuration of 4 servers

| Iterations | 20 000 | 30 000 | 40 000 |
|---|---|---|---|
| LZ4 | 0.72 | 0.82 | 0.88 |
| LZ4 HC 9 | 0.67 | 0.77 | 0.84 |
| zstd 3 | 0.69 | 0.79 | 0.85 |
| zstd 22 | 0.58 | 0.67 | 0.73 |

Table 5.2.: Compressibility of partdiff-IO matrix at different iterations

and 3 clients. Such a setup provides the maximum possible bandwidth between clients and servers, and therefore the fastest IO. The first run compromises 10 iterations, the second 50, and the last 100. The beginning of each run can be identified by an increase of the CPU use to 100%. The vertical purple line identifies the beginning of the IO process. It is clear, that the 100 iterations run is the first of the experiments where there is an equilibrium between the computation time and the time for the IO section.

- Out of the two inference functions available, the simplest one was always used. The current write frequency warranties a good computation-to-IO ratio, so there is no need to increase the use of the CPU resources.

Before continuing with the experiments, it is worth discussing Figure 5.1 in more detail. The graphs in this Figure are obtained with Grafana and follow the same color-coding and scheme as the further Figures in this Chapter. The Figure shows that approximately 10% of the memory is allocated for each of the partdiff-IO runs, which is consistent with the previous statement. During the computation process, the CPU of all the clients reaches 100%, which is the desired and expected result. However, there is also some client CPU use after the IO starts, which should correspond to the slightly asynchronous processing from partdiff-IO. One result that is interesting and unexpected is that the IO process is compromised of two phases: first, the data is transferred from the clients to the servers, generating a very high network use. Then there is a spike in the server's CPU use, which is presumably a consequence of the server synchronization and post-processing process. This second stage is something unexpected, but still a reasonable result.

After this basic experimentation and the visualization of the first results, the preparation is considered successful.

## 5.2. Degrading performance

The goal of this experiment is to challenge the hypothesis that compression is always beneficial. To do so, the setup and compression will be configured to run the application in the most unfavorable circumstance for compression. For that reason, this experiment is run under the following conditions:

- partdiff-IO will be run in its partly asynchronous mode and results collected in close to 40 000 iterations. On one hand, this warranties that, during compression, there is

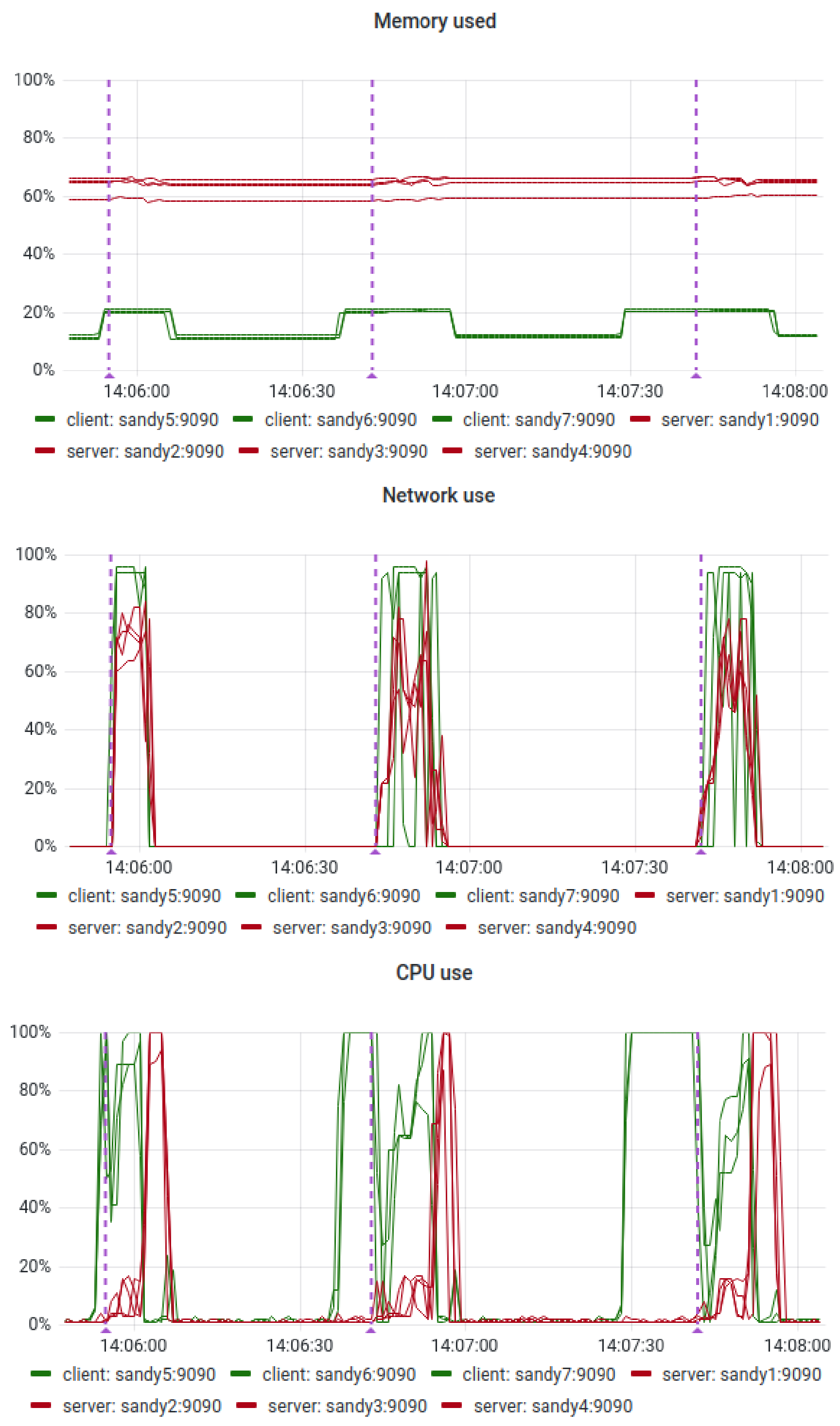


Figure 5.1.: Experiment computation-to-IO ratio

a slight CPU pressure on the clients. On the other hand, the data is compressible by a very little amount. This should warrant that the algorithms still try to compress, although at a very high cost.

- There will be multiple runs of the application, writing data both uncompressed and compressed with LZ4 HC 12 and zstd-22. These are the most demanding algorithm of their group according to Table 4.2 and have the potential of greatly slowing down the IO process.
- The cluster will be configured with 3 clients and 4 servers, across which the data will be evenly split. This warranties that servers do not act as a bottleneck, and clients are free to use as many shared resources as they can. Although having more servers than clients is probably not a real-world scenario, neither is the cluster in this thesis representative of real-world clusters, where client and server machines differ. Therefore, I believe this scenario should still be able to showcase the circumstance where shared resources are mostly free.

The results are presented in Figure 5.2 on page 80. There, the resources used when no compression is executed (top), when LZ4 HC 12 is used (middle), and when zstd-22 is used (bottom) are visualized. Contrary to Figure 5.1, the visualization compromises only the IO cycle. From the point in time where compression for the IO starts (vertical purple line) to the point in time where the IO finishes and new iterations start being computed. In those where compression is executed, the red vertical line represents the point at which the first client finished compression.

From Figure 5.2 I can outline the following remarks:

- The behaviors for the no compression run and the LZ4 HC 12 are fairly similar, and thus presented with the same time frame. It can be observed that the LZ4 HC 12 run takes approximately 4 seconds to compress, and overall, it takes 3 seconds more for all the client machines to go back to 100% CPU. There is certainly some slow-down consequence of the compression, but not critical.
- The behavior for the zstd-22 compressed run is completely different and, unfortunately, must be plotted with a different timeframe. Compared to the less than 20 seconds that it takes no compression and LZ4 HC 12 to execute the complete IO, it takes zstd-22 approximately 1 minute. Regarding the behavior, it is observable how zstd-22 requests a disproportional amount of memory, nearly triplicating the clients' memory use, and the compression process is dramatically slowed-down compared to LZ4 HC 12. It is also visible in those graphs how one of the clients finishes the compression earlier, freeing the memory and executing its corresponding network transfer. It is not only until the other two machines finish compression and free their memory, that the whole data finishes transferring and the servers post-process it.

The results show that it is perfectly possible to strongly degrade the performance of an application by using the wrong compression algorithm in the wrong circumstance. One

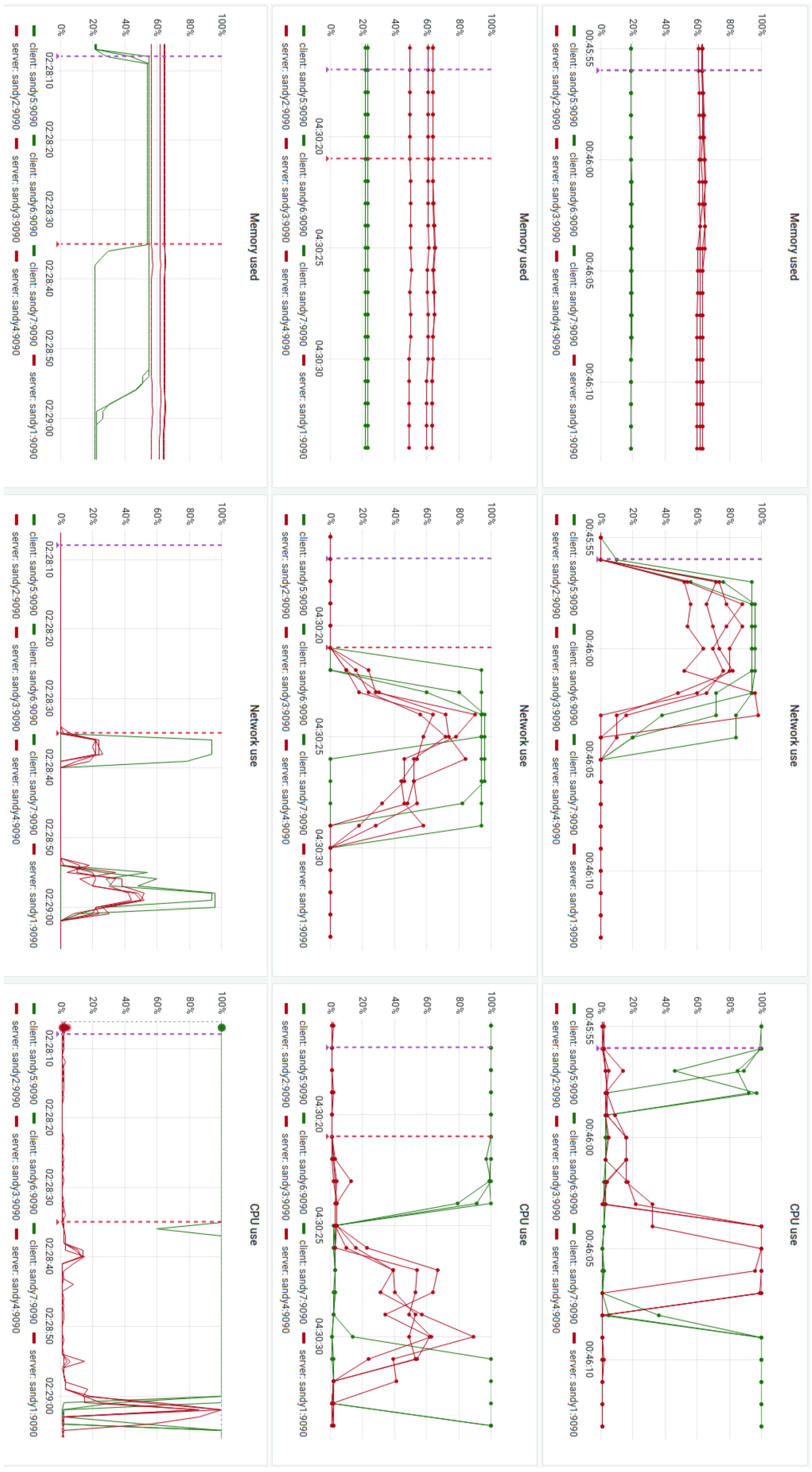


Figure 5.2.: No compression top against zstd-22 compression middle and LZ4-HC 12 bottom for memory, network and CPU use.

additional interesting result from the zstd-22 graph in Figure 5.2, is how the compression requirements can wildly change depending on the data. It is clear, from the figure that one of the clients finishes its processing much earlier, but its CPU was blocked in busy waiting until the other clients finished. The differences in the speed could certainly be related to the compressibility difference for different parts of the matrix which partdiff-IO writes. This should be a warning for any future compression implementations.

## 5.3. Compression improvement

The goal of this experiment is to prove the hypothesis that compression can improve the performance of scientific HPC applications in a real-world scenario. For that reason, this experiment is run under the following conditions:

- partdiff-IO will be run in synchronous mode and results collected close to after 20 000 iterations. That provides data with a reasonable compression ratio according to Table 5.2.
- There will be multiple runs of the application, writing data both uncompressed and compressed with LZ4 1 and zstd-3. These are modern and fast algorithms, with a good equilibrium between their speed and the compression ratio they provide. I expect them to provide a reduction in the network speed at the cost of very little computation time.
- The cluster will be configured with 3 clients and one single server. This warrants the server to be a network and CPU bottleneck for the writing of the data. Even though the overall network could sustain more load, each client will be limited to writing at one-third of its maximum possible bandwidth.

The results can be seen in Figure 5.3 on page 82. There, the resources used when no compression is executed (top), when LZ4 1 is used (middle), and when zstd-3 is used (bottom) are visualized. The purple vertical lines mark the beginning of the compression and the red ones its end. It is expected that for the no-compression use case, both lines overlap since no compression happens. However, the LZ4 lines also overlay, showing a less-than-1-second compression process.

From this figure, I can find the following remarks:

- The hypothesis that compression can improve the performance of some applications has been proven. It is clear in the figure how both LZ4 and zstd runs come back to 100% client CPU faster. That means the IO finishes earlier and the application can come back faster to its processing task.
- It is clear how both the network transfer time and server CPU post-processing time get reduced when less data is transferred. The reduction in the network transfer time is approximately from 18 to 12 seconds, corresponding to a 0.67 compression ratio. This is in line with compression data for 20 000 iterations for both algorithms in Table 5.2.

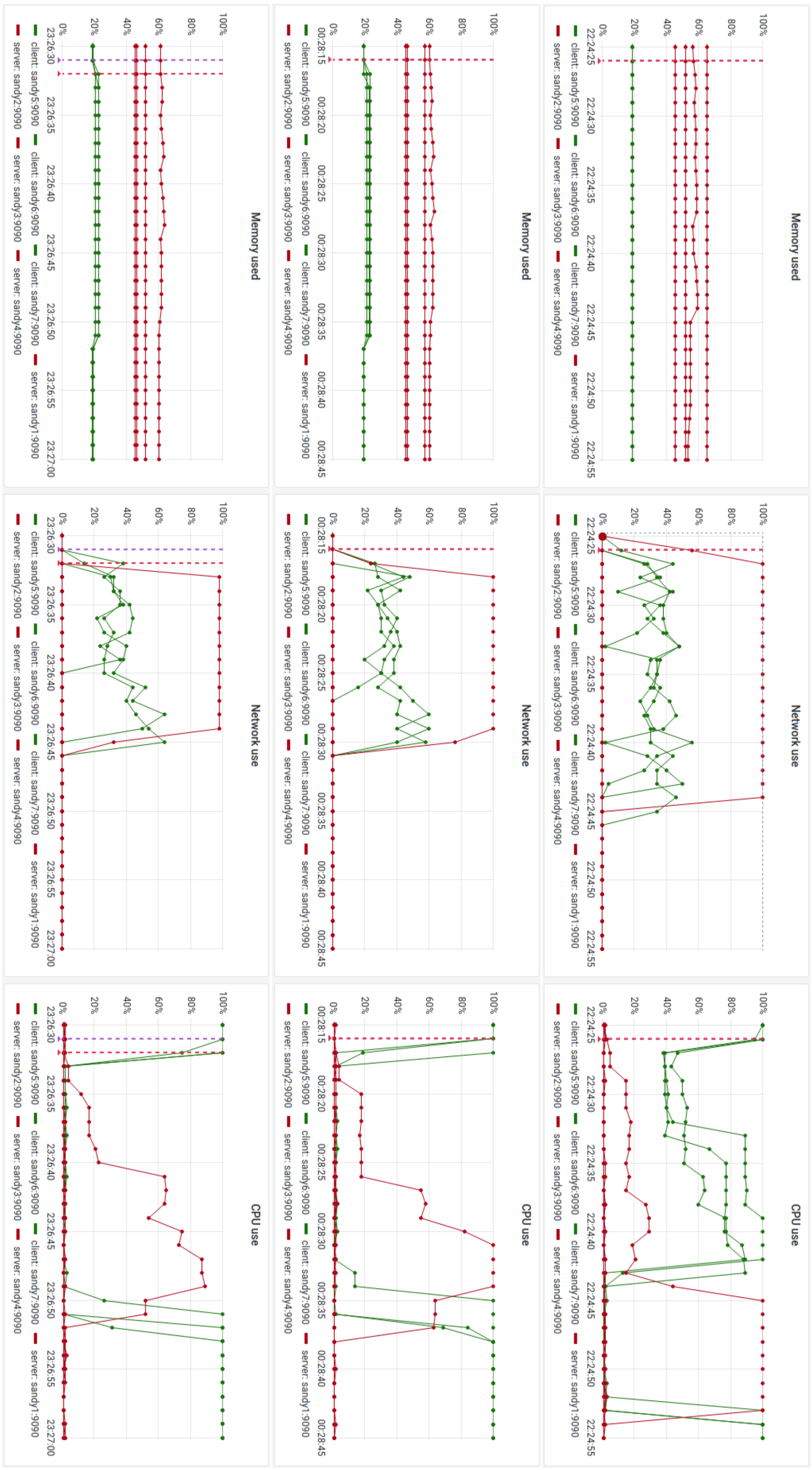


Figure 5.3.: No compression top against LZ4 1 compression middle and zstd-3 compression bottom for memory, network, and CPU use. Purple vertical lines mark the beginning of compression, and red vertical lines, its conclusion.

- It is easy to identify how, during the compression and transfer process, the runs with compression slightly increase their client memory use. Such increment corresponds to the necessary compression buffers that have been discussed in Subsection 4.3.2. Those buffers are only freed after the data has been fully processed at the servers.

Overall, this is a small and very specific, but promising result.

## 5.4. Discussion

The experiments executed in this Chapter are useful to understand some basic logic and consequences of applying compression to a Lustre cluster. The results are promising and that compression can both be a great advantage and a great problem, depending on the circumstances. That strongly justifies the requirement for smart compression and decision-making. However, the experiments depicted here are very limited. They are based on a small custom application and a very limited cluster. Further issues like that which prevented the execution of the application in more than 3 clients add to this problem. Therefore, although the results shown here are promising, their generalization requires further work. For the experiments executed here to be validated, they should be re-run considering:

- The use of different applications which resemble other HPC workflows. partdiff-IO claims to be an example of an application that could run in a Lustre cluster, but outside academic circumstances, it was never its purpose. Ideally, any further tests with another application should consider or prove the resemblance of the application with real HPC workflows.
- The use of a real-world cluster. Even if only resembling the simplified architecture portrayed in Figure 3.1, it would already be a remarkable improvement. The cluster used for these experiments had identical machines connected through GbE, which is far from the reality of any Lustre deployment. The difference could prove crucial to verifying or rejecting the results presented in this thesis.

Additionally, as part of this thesis, it was expected to be able to execute some experiments that could simulate some basic decision-making using user-provided hints. Such work was unfortunately delayed and then canceled due to time constraints. The main reason for this was the additional unexpected amount of time that needed to be committed to getting partdiff-IO in a usable state for this thesis. The codebase was supposed to provide a solid foundation for parallel computation and IO, to which only compression had to be added. Unfortunately, that was not the case, and a relevant part of the work had to be committed to first fix the application. That lead to irreversible delays that made further experiments unfeasible.

### 5.4.1. Data collection impact

For all the experiments presented in this Section and for the generation of the graphs, a similar process was followed, than would be needed for making smart decisions. The `lmetrics` kernel module developed in Section 3.8 was used to log all the necessary information. That information was then picked by a log collector and sent over the network to the machine doing the post-processing. In total, having the data properly post-processed (extracting idle times from CPU usage and interfaces' bandwidth use), a total of 1 `long` value and 3 1-byte numbers were needed as dynamic metrics plus a 1-time `long` value. In total, with a very basic set of metrics and a 1-second collection interval, the maximum communication overhead is approximately 88bps for each machine in the cluster. Even with thousands of machines in the cluster, the bandwidth needed for this communication is trivial compared to the network requirements of such a cluster. However, for the communication to be effective, metadata might also be needed. Since Lustre already implements such communication for a variety of purposes, it is certainly possible to expand some of that communication to include metric exchange, keeping the additional bandwidth to the minimum. In consequence, I can conclude that the data collection has no relevant impact on the network performance of the cluster. Regarding CPU, the post-processing is trivial (fetching data directly from kernel structures, sums, and divisions) and executed asynchronously. Therefore, it is also safe to assume that the CPU impact is minimal, and the data collection shall not have a sizable impact on the cluster performance unless the amount of data shared or its frequency increase dramatically.

# 6. Future work and conclusion

Up to this point, this thesis has covered a basic study of the requirements for smart compression in Lustre and an introductory discussion of the process. This exploratory thesis is, therefore, one of the first steps to making such smart compression real. In consequence, this thesis contains a multitude of simplifications, and many complex topics have been considered out-of-scope or lightly discussed and deferred to future work. Still, it is a good first step in the desired research direction. It is now the place to compile all these blank spots, aggregate them to lead the way for future studies on this topic, and wrap up the results of this thesis.

## 6.1. Future work

This section covers ideas for future work in smart compression. It is based on concepts that have been skipped or not developed enough due to time constraints, as well as incremental improvements to the ideas that were presented. I identified 4 main directions in which improvements can be made. These are the further integration of the read path and decompression metrics and processes; the improvement and implementation of a smart decision-making process; the integration of further metrics and analysis of their importance; and the discussion and study of scalable deployments that consider complex architecture, network and storage configurations.

### 6.1.1. Read path

The read path and decompression logic have been briefly discussed in Section 4.2. However, for the more detailed analysis in Section 4.3, they have been dismissed for space and complexity reasons.

As already outlined in Section 4.2, the read process is different from the write one from several perspectives, including the data usage, the algorithm performance, and the information available about the data. Since they have been already presented at the time, I will not re-iterate them here. However, further study and considerations about the read path might also lead to re-consider some of the assumptions that were made during the discussion of the compression process.

Most remarkably, the client as the decision-maker might not be the best approach for the read path. During reads in Lustre, the communication between clients and servers is executed as a two-step approach. First, the clients ask the servers to locate the data it wants to read since it can be spread among multiple servers and then request the corresponding piece from each server. It was suggested in Section 4.2 that the first step could be used for the servers to also provide the clients with the necessary information for decision-making. However, that process might be limited by the fact that clients cannot make any compression-related decisions until the servers have answered for the first time. Therefore, it might be beneficial to shift some part of the read path decision-making to the servers. There could be different ways to do it, with different possible considerations. Some different possibilities are:

- Servers take a decision considering only local metrics. In this case, the server decompresses only its metrics allow it. Otherwise, it sends to the client the compressed data. This approach can reduce the communication and the decision-making overhead, and make sure shared resources are never overloaded. However, it loses the global picture and might require the client to exceed its resource usage.

- Severs decide in a similar fashion to clients, with the advantage that they have the data directly available. This approach poses the problem of the increased bandwidth and communication required for servers to get the necessary metrics from the clients. Two approaches could be considered. On one hand, servers could periodically request the metrics from the clients. At the expense of lowered accuracy, they would get all the necessary information without a huge bandwidth impact. On the other hand, clients could send the most relevant metrics in the first message where they initiate the communication. The data would be included in every request, potentially increasing the bandwidth under a high IO load. But servers would get accurate information on the status of clients and the decisions could be, potentially, more accurate.

### 6.1.2. Decision-making

The *requirements* for an effective and structured study of the decision-making for smart compression have been greatly studied in this thesis, even though some simplifications were considered in the process. Additionally, the actual implementation of such decision-making has been only theoretically tackled. Therefore, future studies on the decision-making process could go in both directions. They could deepen into the requirements for the smart compression and possible additions to the theoretical process, or consider an actual implementation.

Regarding possible improvements and additions to the decision-making process, the following tasks could be part of future work:

- For simplicity, in Section 4.1 it was decided to only consider the location and the algorithm configuration decisions. These two were deemed the most relevant ones out of an opinionated list of possible decisions to take. In consequence, there are multiple possible paths of improvement in this process. Future studies could improve the status by doing a more detailed and structured consideration of all the decisions to be taken. Certainly, the number of components involved in a Lustre cluster and the possibility to configure most of them, offer further paths of research. Some of them were already mentioned in Section 4.1, like deciding on an appropriate Lustre networking configuration or tunning ZFS parameters for better storage bandwidth depending on the compression in use.

- Although the location and algorithm configuration decisions were considered in detail in Section 4.1, it was decided to tackle them independently. This approach simplified the exploratory study and allowed extracting the relevant metrics for each of those decisions, regardless of their interaction. However, after such analysis, the question arises of whether simultaneous decision-making over both decisions would be more appropriate. In the case of a big imbalance between resources available to clients and servers, the decision-making process might be flawed. This is more obvious presenting the example. When servers had plenty of CPU but little memory available, and clients the exact opposite, taking the location decision ahead might not be the most appropriate. The best-fitting algorithms for this circumstance will certainly change depending on whether compression happens in the client or the servers. Therefore, making the location decision first might discard an algorithm that would be more fitted for the circumstance. Future research work could consider a simultaneous decision-making process and study co-relations between metrics that affect both decisions.

- Further and more complex decision-making approaches could also be considered. In this thesis, only a synchronous fail-safe decision process has been considered. In this situation, decisions are expected to be taken only when a read or write happens and default to no compression in case compression is attempted and failed. Therefore, there is space for improvement both in the synchronicity of the decision and when failures happen. One possibility would be to consider a re-evaluation of past decisions based on acquired knowledge and cluster status. For example, it might be beneficial to try to re-compress data that was compressed with a low compression ratio algorithm if such data is not read for a long time. The same re-evaluation could be considered in case the compression of some data failed or produced undesired results, like the data resulting from compression having a too low compression ratio.

- In this thesis, only LZ4 and zstd algorithms have been considered. However, there exists a multitude of other compression algorithms which could be beneficial depending on the properties of the data and the status of the cluster. Future work could consider the analysis and integration of further algorithms, following the process and characterization in Section 4.3.

However, it would also be possible to consider an actual implementation of the smart decision-making process. This part has seen little attention apart from the theoretical study and some hints presented in the calculation proposals from Subsections 4.3.1 and 4.3.2. However, the theoretical study has shown both the places which need further study and different ways in which the solution the smart compression could be reached.

From the tasks which need further study, it is most remarkable that a greater understanding of the resources used by compression algorithms and the impact of different kinds of data is needed. The small characterization in Table 4.2 has been useful for this study but is limited. It presents the characterization of a single dataset on a single machine and the results in Chapter 5 already show great differences in the behavior of algorithms. This is a requirement for any further study of smart compression. Without a correct understanding of how might the algorithms behave in real-world circumstances with real-world datasets, it would never be possible to warranty an efficient decision-making process. The experiments in Chapter 5 show promising results and the importance of having smart compression enabled, but much more experimentation with real-world applications is needed. Related to this, the memory characterization in Table 4.2 presents important information, like which algorithms allocate additional memory. However, a quantitative approach is still missing, since the extent of the memory use for zstd as also observed in Figure 5.2 is certainly unknown.

Regarding the actual implementation of the smart compression, related work and the study in this thesis have shown that there are two ways to tackle the problem: through a heuristic approach and by using black-box Machine Learning algorithms.

**Heuristic solutions** Of this kind are the calculation proposals in Subsections 4.3.1 and 4.3.2. This approach considers a mathematical study of the problem, a learning process, and a final solution that should be statistically proven to approximate the best-case scenario. The advantage of this approach is that the reasons for a certain decision can be explained, and it is possible to learn from mistakes and iterate to improve the performance. Its main drawback is that it requires producing a mathematical model, which is hard to achieve and requires a lot of expert knowledge which is hard to obtain. When looking at the calculation proposals in Subsections 4.3.1 and 4.3.2, it becomes obvious that extracting the coefficients for the proposed model is extremely challenging. They would require to run a multitude of experiments with a great number of algorithms, manually decide on the best approach, and with multiple iterations improve the results.

**Machine Learning approaches** These solutions have started to become popular in recent research, with different examples of adaptive compression being published [58, 76]. These solutions use some labeled input data known as *training* data to refine their internal state to correctly identify a designated output. Once it is considered correct enough, new data can be fed to the algorithms, which will provide the desired assessment. Their main advantage is that their use does not require extensive experience and knowledge about the possible complex co-relations between the inputs and the outputs. Instead, such complexity is hidden in the internal state. Their main drawbacks are the requirement for extensive amounts of training data, and the complexity of their training process, which often requires expert knowledge. The collection data process would require a similar approach as the heuristic solutions based on experimentation, although the algorithm characterization process could be avoided.

### 6.1.3. Metrics

One important discussion point throughout this thesis has been the collection and analysis of system metrics. They have been initially collected and discussed in Chapter 3, and then their usage was analyzed in Section 4.3. Finally, a small amount has been collected and used in the experiments in Chapter 5. However, the length of the discussion and the complexity of the analysis for some of the metrics required considering some of them, their analysis, or their integration in the decision-making process out-of-scope. Future work could attempt to include the missing metrics and extend the discussion.

The metrics that have been discussed but not considered for the decision-making process are:

- CPU low-level metrics: these were originally discarded for the complexity of translating them to real-world performance. Instead, it was suggested to use average instructions per cycle as a way to summarize their impact on the decision-making. However, for an accurate comparison between the power available in clients and servers, such low-level metrics should be taken into account. In the common situation where the hardware in clients and servers differ, it would be important to have a faithful comparison. These low-level metrics could be used, together with benchmarks, to better approximate the difference in power between different machines.
- Memory static metrics: the memory static metrics have not been considered for decision-making. The reason is that their impact on the performance of algorithms is unknown or hard to measure, with research being seldom published. Still, the memory speed, and its latency could play a role in the compression process, since the data needs to be read and will be written back to memory.
- CPU-memory bandwidth: the CPU-memory bandwidth has been discarded from the decision-making. It was considered much bigger than the network bandwidth, and therefore never a bottleneck. However, the network setup in real-world HPC configurations, including the existence of multiple interfaces or Infiniband technology might challenge that assumption. If that were ever the case, it would be important to understand the limitation that the memory bandwidth poses in the general execution of the algorithms. In Table 4.2, it can be seen that for very fast algorithms (like LZ4 100), the memory bandwidth already plays some role. With an overall throughput of one-fourth of the bandwidth, already approximately 25% of the time will be spent just on the transfer of the data to memory.
- Storage metrics and IO patterns have not been integrated with the location decision. They have both not been considered relevant enough for this exploratory study. However, there are circumstances where their importance grows. For example, in the case of a read-modify-write workflow, where the clients read and change small pieces of data, the process would be more efficient to execute in the servers. For that reason, the clients would need to identify the specific pattern (for example, through a great number of `RW_CHANGES` in darshan) and take the decision accordingly.

Additionally, some other metrics have not been discussed or deliberately left unconsidered. These are metrics that are believed to have little relevance or impact on the decision-making. However, such belief has not been proven and could be of interest to future work to challenge the assumption. These metrics are:

- ZFS metrics outside the ARC: ZFS is a greatly configurable system that can provide metrics and information about many of the tasks it executes. For example, the whole

IO pipeline is monitored, and ZFS can provide metrics about the number of pending operations, the transactions executed, or their size. These metrics have not been used in this thesis due to the great complexities that the storage metrics pose. However, they might be of interest in certain circumstances where the storage system performance is well characterized and the implications of ZFS metrics known.

- CPU use consequence of network use: Similar to the CPU required for IO operations, network operations require a certain amount of CPU. However, that CPU use is, in general, much lower. That assumption lead to dismissing the introduction of the said metric. Still, that CPU use exists and could be considered. Any compression that reduces the amount of data to be transferred will also, indirectly, reduce the CPU that network operations require. This could be a slight advantage for compression, which helps reduce the impact of compression algorithms on the system CPU.

Finally, the data collection process and its implications have only been briefly covered in Subsection 5.4.1. There, a short discussion of the bandwidth requirement for the experiments executed in Chapter 5 takes place. However, further requirements like the sampling frequency for different groups of metrics or the latency for the extraction and processing of the values have not been discussed. Future work aiming at improving the metric collection process could do a more detailed analysis of the timing and frequency needed for an effective metric collection.

### 6.1.4. Scalable deployments

During this thesis, there have been discussions on the possibilities of highly-scalable Lustre deployments like the example architecture presented in Figure 2.1. However, the in-depth study has considered such deployments out of scope. The main reason is that directly tackling a highly complex system in an exploratory study is challenging and could lead to a too shallow study. Instead, it was considered more appropriate to study a simple system and leave it for future work to build up to a more complex study. In this thesis, the simplifications regarding the scale have been considered at the architecture, network, and storage levels. Future work could consider them.

**Complex architecture** In this thesis, a simple client-server configuration has been considered. Although some cluster metrics like the client-server ratio and the number of routable servers per client have been studied, their implications have been briefly considered. Considerations like the scalability of the communication or the decision-making have been kept out of scope. For example, it is common that scientific applications to span multiple nodes in the cluster. In such circumstances, the decision-making approach could be challenged, by considering a more centralized decision-making process. Additionally, client-to-client communication has not been considered in this thesis. However, some clients might still benefit from knowing the status of their peers, for example, to make better predictions on the use of shared resources. Future work could deep into these topics to close the gap between this exploratory study and real-world scenarios.

**Complex network** The interconnection between clients and servers has been considered, in this thesis, as a direct connection between machines with single network interfaces. Real-world scenarios are much more complex. They often involve servers with multiple interfaces for data transfer, fail-over machines, and Lustre routers that connect different parts of the network. This complexity poses a problem both for metric collection and for the decision-making process. Regarding the metric collection, it has to be adapted to consider multiple interfaces, varying distances and latency for different components, and the overall correct assessment of the network circumstance for both the whole cluster and individual components. The decision-making process should identify the network paths relevant for each client and act upon them. For example, if a server with two interfaces only routes to a certain client through one of them, the decision-making process in the said client should be made aware of this.

**Complex storage** Similar to the network study, storage metrics in this thesis have only considered a very simplified storage scenario with a single disk that can be potentially benchmarked. However, scaled-up Lustre clusters can have a lot of different types of storage, as depicted in Figure 2.1. The storage can have different kinds of redundancy, consist of heterogeneous groups of drives, or be shared by multiple OSSs, among others. These scenarios have different consequences in the metric collection and decision-making processes, with complex characterizations and possible interactions among each other. Future work could aim to understand the issues of these circumstances and look for solutions that can integrate complex storage into the system.

## 6.2. Conclusion

This thesis is, to the knowledge of the author, the first in considering a full-scale study of smart compression for Lustre. As such, and given the existence of few previous work in the field [29], made this thesis an exploratory study. As such, its goals have been to study a simplified scenario and extract both theoretical and practical conclusions that could be leveraged by further studies.

The theoretical discussion in Chapter 3 provides useful insights into the systems involved in compression for a Lustre setup and the metrics that can potentially characterize them.

Afterwards, Chapter 4 contains the core of the discussion about smart compression. It does not only put the metrics from Chapter 3 into context but discusses the smart compression process in depth. It centers its attention on the two metrics that are considered more relevant: the location and the compression algorithm configuration to use. These two, although with some simplifications, are analyzed in great detail, which allows understanding their relevance and the process that can be followed when applying them. As part of that process, there is also an initial characterization of compression algorithms relative to the needs of smart compression. Although very specific, it lies the base for the requirements of future more complex characterizations for smart compression.

Once the theoretical foundations for smart compression were laid, Chapter 5 provided some experimentation to prove the feasibility of some of the previous discussions. Results show that compression can be beneficial but also a problem depending on the configuration. This supports the idea that an adaptive and smart compression process can be a great benefit for a Lustre cluster.

Finally, this chapter has also covered possible future paths. Further research could challenge some of the results presented in this thesis and deepen into the implementation and development of the actual smart decision-making process.

# A. Decompression resource usage

Although briefly covered in the main body of this thesis, compression algorithms are expected to use less resources during decompression than during compression. The experiments with lzbench executed in Subsection 4.3.2 and presented in Table 4.2 gave good insights about the compression characterization. However, the lzbench experiments executed both compression and decompression. This allows to also present some decompression results, which can be viewed in Table A.1. There, the CPU time and throughput for both compression and decompression for the same algorithms as in Table 4.2 is presented. The results show that although the memory bandwidth is the same in both cases, **all** the algorithms performed faster and required fewer CPU time during decompression. This is most remarkable for the slow high-compression algorithms like LZ4 HC 12 and zstd-22, whose decompression is approximately 350 times faster than their compression. These results could be a good input for future work in decompression.

| | Compression | | Decompression | |
|---|---|---|---|---|
| Algo | CPU time (s) | Throughput (MB/s) | CPU time (s) | Throughput (MB/s) |
| none | 0 | 8064 | 0 | 8041 |
| LZ4 | 0.39 | 509 | 0.05 | 3138 |
| LZ4 10 | 0.24 | 782 | 0.04 | 3363 |
| LZ4 100 | 0.08 | 2120 | 0.02 | 5132 |
| LZ4 HC 1 | 2.03 | 99 | 0.06 | 2879 |
| LZ4 HC 12 | 22.13 | 9.13 | 0.06 | 3130 |
| zstd-3 | 1.10 | 183 | 0.25 | 775 |
| zstd-9 | 4.92 | 41 | 0.23 | 853 |
| zstd-22 | 99.57 | 2.03 | 0.27 | 731 |
| zstd-fast-1 | 0.60 | 329 | 0.16 | 1195 |
| zstd-fast-10 | 0.39 | 496 | 0.12 | 1497 |
| zstd-fast-100 | 0.15 | 1187 | 0.04 | 3024 |

Table A.1.: Results from lzbench for compression and decompression with silesia corpus on a single core, Intel(R) Xeon(R) CPU E31275 @ 3.40GHz.

Additionally, it can be seen that for those algorithms whose authors claim a constant decompression speed (LZ4 HC and zstd) are the ones with the least variability. Although values vary slightly, their differences are a lot less dramatic than the ones of other algorithms, proving at least partial success in their authors intentions.

Regarding the memory used, it has not been studied here. Unfortunately, the memory buffers are still a requirement when doing decompression and those can never be avoided. However, it would still need to be studied if the memory allocation for high zstd levels also happens during decompression.

# Bibliography

[1] Jean Delvare Alan Cox and Anton Arapov. dmidecode. https://www.nongnu.org/dmidecode. Last visited: 27th May 2022.

[2] Lustre authors. llapi ladvise. https://doc.lustre.org/lustre_manual.xhtml#idm139974522654416. Last visited: 19th July 2022.

[3] Lustre authors. Server-side advice and hinting. https://doc.lustre.org/lustre_manual.xhtml#idm139974532022048. Last visited: 19th July 2022.

[4] LZ4 authors. Lz4 benchmarks. http://lz4.github.io/lz4/#benchmarks. Last visited: 22nd March 2022.

[5] OpenZFS authors. Native and user-defined properties of ZFS datasets. https://openzfs.github.io/openzfs-docs/man/7/zfsprops.7.html. Last visited: 10th June 2022.

[6] OpenZFS authors. Openzfs. https://github.com/openzfs/zfs. Last visited: 27th May 2022.

[7] OpenZFS authors. Openzfs history. https://openzfs.org/wiki/History. Last visited: 27th May 2022.

[8] Various authors. Thermodynamic entropy - an overview. https://www.sciencedirect.com/topics/computer-science/thermodynamic-entropy. Last visited: 27th May 2022.

[9] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio. Last visited: 27th July 2022.

[10] btrfs authors. Btrfs status. https://btrfs.wiki.kernel.org/index.php/Status. Last visited: 3rd August 2022.

[11] Scott Callaghan, Philip Maechling, Patrick Small, Kevin Milner, Gideon Juve, Thomas H Jordan, Ewa Deelman, Gaurang Mehta, Karan Vahi, Dan Gunter, Keith Beattie, and Christopher Brooks. Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. *The International Journal of High Performance Computing Applications*, 25(3):274–285, 2011.

[12] Giacomo Capizzi, Salvatore Coco, Grazia Lo Sciuto, Christian Napoli, and Waldemar Hołubowski. An entropy evaluation algorithm to improve transmission efficiency of

compressed data in pervasive healthcare mobile sensor networks. *IEEE Access*, 8:4668–4678, 2020.

[13] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.

[14] Konstantinos Chasapis, Manuel Dolz, Michael Kuhn, and Thomas Ludwig. Evaluating Power-Performace Benefits of Data Compression in HPC Storage Servers. In Steffen Fries and Petre Dini, editors, *ENERGY 2014: The Fourth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 29–34. IARIA XPS Press, 04 2014.

[15] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*, pages 213–224, 2015.

[16] Russell Coker. Bonnie++. `https://doc.coker.com.au/projects/bonnie/`. Last visited: 27th July 2022.

[17] Yann Collet. Lz4. `http://fastcompression.blogspot.com/p/lz4.html`. Last visited: 22nd March 2022.

[18] Yann Collet. Lz4 documentation. `https://github.com/lz4/lz4/tree/dev/doc`. Last visited: 2nd August 2022.

[19] Yann Collet. New generation entropy codecs: Finite state entropy and huff0. `https://github.com/Cyan4973/FiniteStateEntropy`. Last visited: 23rd July 2022.

[20] Yann Collet. New generation entropy coders. `https://github.com/Cyan4973/FiniteStateEntropy`. Last visited: 1st December 2021.

[21] Yann Collet and Chip Turner. Smaller and faster data compression with zstandard. `https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/`. Last visited: 2nd August 2022.

[22] darshan authors. darshan - HPC I/O characterization tool. `https://www.mcs.anl.gov/research/projects/darshan/`. Last visited: 25th of May 2022.

[23] darshan authors. darshan I/O characterization fields. `https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html#_i_o_characterization_fields`. Last visited: 5th of May 2022.

[24] Sebastian Deorowicz. The Silesia corpus. `https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia`. Last visited: 8th July 2022.

[25] Telecommunication Networks Group Politecnico di Torino. tstat - TCP STatistic and analysis tool. `http://tstat.polito.it`. Last visited: 27th May 2022.

[26] Jeff Diamond, Martin Burtscher, John D. McCalpin, Byoung-Do Kim, Stephen W. Keckler, and James C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 32–43, 2011.

[27] P. Skibinski et al. lzbench. `https://github.com/inikep/lzbench`. Last visited: 8th July 2022.

[28] Rafael Ferreira da Silva, Henri Casanova, Anne-Cécile Orgerie, Ryan Tanaka, Ewa Deelman, and Frédéric Suter. Characterizing, modeling, and accurately simulating power and energy consumption of i/o-intensive scientific workflows. *Journal of Computational Science*, 44, 6 2020.

[29] Anna Fuchs. Client-side data compression. `https://jira.whamcloud.com/browse/LU-10026`. Last visited: 22nd July 2022.

[30] Anna Fuchs, Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Enhanced Adaptive Compression in Lustre, 06 2017. `https://wr.informatik.uni-hamburg.de/_media/research/publications/2017/poster_isc17.pdf`. Last visited: 27th May 2022.

[31] Pawel Gepner, Victor Gamayunov, and David Fraser. Early performance evaluation of avx for hpc. *Procedia CS*, 4:452–460, 12 2011.

[32] Grafana Labs. Grafana: The open observability platform. `https://grafana.com/`. Last visited: 29th July 2022.

[33] Danny Harnik, Ronen Kat, Dmitry Sotnikov, Avishay Traeger, and Oded Margalit. To zip or not to zip: Effective resource usage for Real-Time compression. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 229–241, San Jose, CA, February 2013. USENIX Association.

[34] Scott Huck. Measuring memory bandwidth on the intel xeon processor 7500 series platform. `https://www.intel.com/content/dam/doc/white-paper/resources-xeon-7500-measuring-memory-bandwidth-paper.pdf`. Last visited: 5th March 2022.

[35] Intel. Intel® server board and system products update on intel® turbo boost technology support with low power intel® xeon® processor 3400/5500/5600 series. `https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/white_paper_turbo_boost_on_low_power_processor.pdf`. Last visited: 5th March 2022.

[36] Andrew B. Kahng. Scaling: More than moore's law. *IEEE Design & Test of Computers*, 27(3):86–87, 2010.

[37] The kernel development community. Cpu load. https://www.kernel.org/doc/html/latest/admin-guide/cpu-load.html. Last visited: 27th May 2022.

[38] The kernel development community. ethtool - utility for controlling network drivers and hardware. https://mirrors.edge.kernel.org/pub/software/network/ethtool. Last visited: 27th May 2022.

[39] The kernel development community. Linux Core API documentation - CPU hotplug in the kernel. https://docs.kernel.org/core-api/cpu_hotplug.html. Last visited: 27th May 2022.

[40] The kernel development community. The Linux kernel user's and administrator's guide, i/o statistics fields. https://docs.kernel.org/admin-guide/iostats.html. Last visited: 30th May 2022.

[41] The kernel development community. Linux networking documentation, interface statistics. https://www.kernel.org/doc/html/latest/networking/statistics.html#c.rtnl_link_stats64. Last visited: 27th May 2022.

[42] The kernel development community. The /proc filesystem, kernel data. https://www.kernel.org/doc/html/latest/filesystems/proc.html#kernel-data. Last visited: 27th May 2022.

[43] The kernel development community. The /proc filesystem, meminfo. https://docs.kernel.org/filesystems/proc.html#meminfo. Last visited: 27th May 2022.

[44] C. Krintz and S. Sucu. Adaptive on-the-fly compression. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):15–24, 2006.

[45] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Data Compression for Climate Data. *Supercomputing Frontiers and Innovations*, pages 75–94, 06 2016.

[46] Michael Kuhn, Julius Plehn, Yevhen Alforov, and Thomas Ludwig. Improving Energy Efficiency of Scientific Data Compression with Decision Trees. In *ENERGY 2020: The Tenth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 17–23. IARIA XPS Press, 09 2020.

[47] Chris A. Mack. Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, 2011.

[48] Chris Mason. zstd support. https://lkml.iu.edu/hypermail/linux/kernel/1709.1/01998.html. Last visited: 12th July 2022.

[49] John McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25, 12 1995.

[50] Inc. Meta Platforms. Zstandard benchmarks. https://facebook.github.io/zstd/#benchmarks. Last visited: 7th August 2022.

[51] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 35–43, 2007.

[52] OpenZFS. OpenZFS documentation, performance and tuning. `https://openzfs.github.io/openzfs-docs/Performance%20and%20Tuning/index.html`. Last visited: 27th May 2022.

[53] OpenZFS. OpenZFS documentation, performance and tuning. `https://openzfs.github.io/openzfs-docs/Performance%20and%20Tuning/Workload%20Tuning.html#dataset-recordsize`. Last visited: 30th May 2022.

[54] OpenZFS. Openzfs manual pages, zpool-iostat. `https://openzfs.github.io/openzfs-docs/man/8/zpool-iostat.8.html`. Last visited: 27th May 2022.

[55] Intel Corporation Oracle and/or its affiliates. Lustre software release 2.x - operations manual. `https://www.lustre.org/documentation/`. Last visited: 22nd March 2022.

[56] D.S. Ornstein and B. Weiss. Entropy and data compression schemes. *IEEE Transactions on Information Theory*, 39(1):78–83, 1993.

[57] Uli Plechschmidt. It's lonely at the top: Lustre continues to dominate top 100 fastest supercomputers. `https://community.hpe.com/t5/Advantage-EX/It-s-lonely-at-the-top-Lustre-continues-to-dominate-top-100/ba-p/7109668`. Last visited: 22nd July 2022.

[58] Julius Plehn, Anna Fuchs, Michael Kuhn, Jakob Lüttgau, and Thomas Ludwig. Data-aware compression for hpc using machine learning. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '22, page 8–15, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Laura Promberger, Rainer Schwemmer, and Holger Fröning. Assessing the overhead of offloading compression tasks. In *49th International Conference on Parallel Processing (ICPP): Workshops*, ICPP Workshops 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Laura Promberger, Rainer Schwemmer, and Holger Fröning. Characterization of data compression across cpu platforms and accelerators. *Concurrency and Computation: Practice and Experience*, n/a(n/a):e6465, 2021.

[61] Prometheus Authors. Prometheus - Monitoring system and time series database. `https://prometheus.io/`. Last visited: 29th July 2022.

[62] The rdma development community. RDMA core userspace libraries and daemons. `https://github.com/linux-rdma/rdma-core`. Last visited: 27th May 2022.

[63] Richard Stallman. Interpreting, enforcing and changing the gnu gpl, as applied to combining linux and zfs. https://www.fsf.org/licensing/zfs-and-linux. Last visited: 10th August 2022.

[64] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), aug 2013.

[65] Seagate. Ironwolf pro datasheet. https://www.seagate.com/files/www-content/datasheets/pdfs/ironwolf-pro-20tb-DS1914-20-2204US-en_US.pdf. Last visited: 27th July 2022.

[66] Samsung semiconductors. Internal ssd 980. https://semiconductor.samsung.com/consumer-storage/internal-ssd/980/#specs. Last visited: 27th July 2022.

[67] Zabbix SIA. Zabbix manual. https://www.zabbix.com/documentation/current/en/manual/config/items/itemtypes/zabbix_agent. Last visited: 30th May 2022.

[68] Nick Terrell. Update to zstd-1.4.10. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c8c109546a19613d323a319d0c921cb1f317e629. Last visited: 3rd August 2022.

[69] The Open MPI Project. Open Source High Performance Computing: A High Performance Message Passing Library. https://www.open-mpi.org/. Last visited: 29th July 2022.

[70] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, 34(4):825–845, oct 1987.

[71] Vincent M Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, volume 13, page 5, 2013.

[72] Wikipedia. Data compression. https://en.wikipedia.org/wiki/Data_compression. Last visited: 1st July 2022.

[73] Wikipedia. Entropy. https://en.wikipedia.org/wiki/Entropy. Last visited: 27th May 2022.

[74] Bing Xie, Zilong Tan, Philip Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan S. Vazhkudai, and Feiyi Wang. Interpreting write performance of supercomputer i/o systems with regression models. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 557–566, 2021.

[75] Shinichi Yamagiwa and Suzukaze Kuwabara. Autonomous parameter adjustment method for lossless data compression on adaptive stream-based entropy coding. *IEEE Access*, 8:186890–186903, 2020.

[76] Shinichi Yamagiwa, Wenjia Yang, and Koichi Wada. Adaptive lossless image data compression method inferring data entropy by applying deep neural network. *Electronics*, 11(4), 2022.

[77] Alireza Yazdanpanah and Mahmoud Reza Hashemi. A new compression ratio prediction algorithm for hardware implementations of lzw data compression. In *2010 15th CSI International Symposium on Computer Architecture and Digital Systems*, pages 155–156, 2010.

# List of Figures

# List of Tables

## Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Intelligent Adaptive Systems selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Pablo Correa Gómez

Hamburg, den 21.08.2022 Vorname Nachname

[illegible]

## Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Pablo Correa Gómez

Hamburg, den 21.08.2022 Vorname Nachname

[illegible]