Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Dynamic decision-making for efficient compression in parallel distributed file systems

vorgelegt von

Janosch Hirsch

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

Studiengang:	Informatik
Matrikelnummer:	6208820
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn
Betreuer:	Dr. Michael Kuhn, Anna Fuchs

Hamburg, 2017-08-12

Abstract

The technology gap between computational speed, storage capacity and storage speed poses big problems especially for the HPC field. A promising technique to bridge this gap is data reduction through compression. Compression algorithms like LZ4 can reach compression speeds high enough to be applicable for the HPC field. Consequently efforts to integrate compression into the Luste file system are in progress. Client side compression also brings the potential to increase the network throughput. But to be able to fully exploit the compression potential the compression configuration has to be adapted to its environment. The more adaptations to the data structure and machines condition the better the compression effectiveness will be.

This objective of this thesis is to design a decision logic that dynamically adapts the compression configuration to maximize a desired trade-off between application speed and compression. Different compression algorithms and the conditions for compression on the client side of a distributed file systems are examined to identify possibilities to apply compression. Finally an implemented prototype of the decision and adaption logic is evaluated with different network speeds and starting points to further improve the concept are given.

Contents

1. Introduction

Motivation

Storing the huge data amounts produced by numerical simulations in the High Performance Computing (HPC) field is an increasing problem, due to the ever increasing gap between computation and storage technology[5]. The development rates of computation speed, storage capacity and storage speed are shown in Figure 1.1. Since 30 years com-

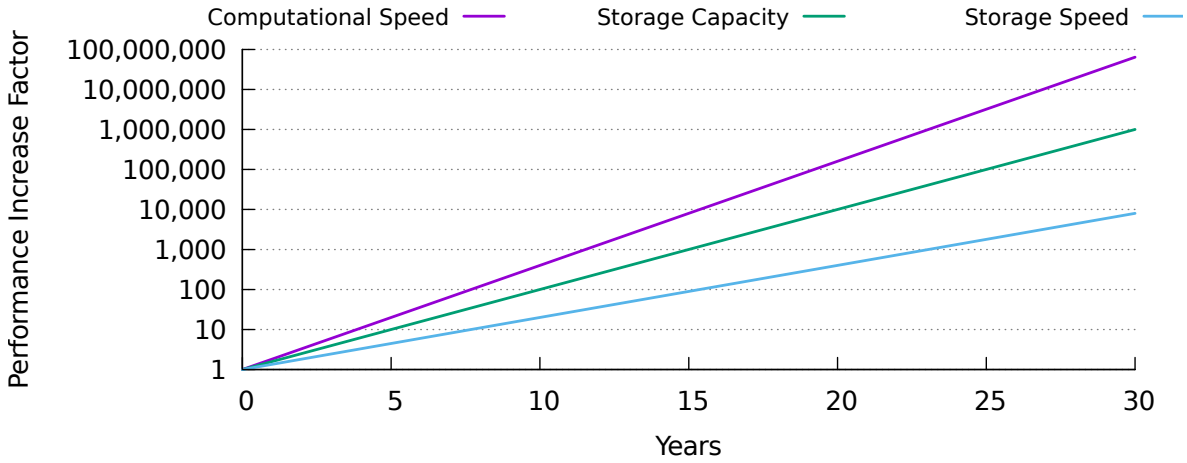

Figure 1.1.: Development of computational speed, storage speed and storage capacity[5].

putation speed is growing faster than storage capacity and speed, resulting in data being produced way faster than it can be stored and storage space running out quickly. Data reduction can help bypassing both problems, thus techniques like compression get a more and more prominent role in the HPC field. File systems like Zettabyte File System (ZFS) and B-tree File System (Btrfs) already provide built-in transparent compression[1][2]. In the HPC field though, where the storage throughput introduces serious bottlenecks and the storage costs accumulate to huge expenses, the state-of-the-art distributed file systems do not come with native compression support yet. The widely established distributed file system Lustre can be set up on top of ZFS to facilitate Lustre transparent server side compression at least[3]. However Intel and the Scientific Computing group at Universität Hamburg are currently working on integrating server side and client side data compression directly into Lustre[4]. Client side compression will be performed on the file system's clients (typically compute nodes). This relives the storage servers of compression load and can potentially induce network throughput enhancements.

High performance distributed file systems like Lustre are usually deployed site wide and used by various applications having different Input/Output (I/O) write patterns and data structures. Thus a static compression configuration approach like used in ZFS would be quite inefficient for client side compression as the configuration would have to be very conservative for the compression speed to always meet at least the network speed and not introduce a new bottleneck. This would waste a lot of compression potential in many cases whereas a dynamic compression approach that dynamically adapts to an individual application could potentially achieve higher compression rates and throughput enhancements.

Goals

The goal of this thesis is to design and implement a decision-making logic that dynamically adapts the compression configuration according to the systems conditions to enhance the resource utilization and compression effectiveness. The decision-making will be based on static and dynamic parameters regarding infrastructure and system information. Additionally the application developer can provide explicit information about file properties by suppling file hints via an Application Programming Interface (API). Compression strategies will enable to choose and weigh different trade-offs between maximizing the compression factor and maximizing the network throughput. The decision-making logic will be implemented as a prototype in the user space. In future work the logic is intended to be integrated into the Lustre file system to facilitate dynamic compression on client and server side.

In the course of this thesis the compression algorithms LZ4, Zstd and XZ will be evaluated extensively on their eligibility for dynamic compression and compression strategies for achieving different objectives will be elaborated.

Structure

This thesis is structured in five parts. In Chapter 2 background information about the Lustre file system, compression algorithms and related work is provided. Chapter 3 elaborates on the design principles of the decision-making logic and requirements for effective compression. Chapter 4 provides information about the implementation details. In Chapter 5 the compression algorithms, compression strategies and the dynamic decision logic are evaluated. And finally Chapter 6 concludes the work and proposes future work.

2. Background

2.1. Lustre

Lustre is a parallel distributed file system for Linux licensed under the GNU General Public License Version 2 (GPLv2). It is designed for high performance and high scalability, thus commonly used in large computer clusters and supercomputers having tens of thousands of clients and hundreds of storage servers. The first release of Lustre was in December 2003. Today Lustre is the most common file system in the HPC area and widely present at the top of the TOP500 list of supercomputers[6]. Lustre's high scalability allows for up to hundreds of Petabytes of total storage space and multiple Terabytes per second of aggregated throughput[3].

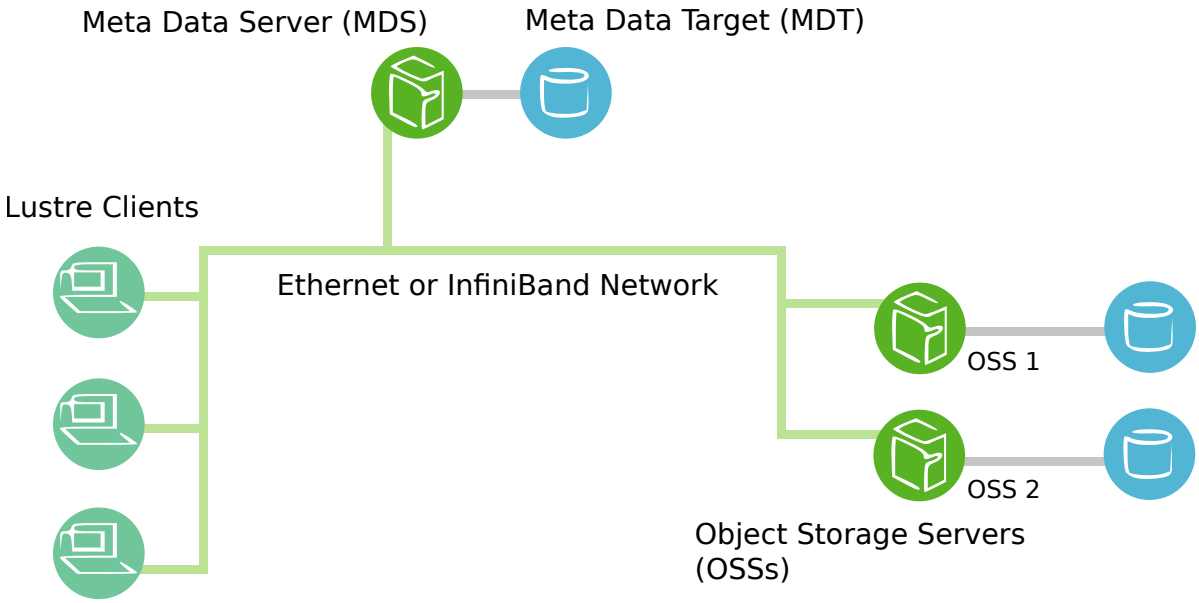

Figure 2.1.: Lustre file system components in a basic cluster[3].

Figure 2.1 displays the structure and components of the Lustre file system. The core components include Meta Data Servers (MDSs), Object Storage Servers (OSSs) and clients. Each MDS manages one or more Meta Data Targets (MDTs) to store the meta data on and each OSS has one or more Object Storage Targets (OSTs) to store the data on. MDSs provide clients with the meta data of all files stored in the Lustre file system under a single namespace. OSSs handle I/O requests form clients an store data in a single objects on the OSTs. OSTs work either on ZFS or ldiskfs, which is a fourth extended file system (ext4) fork. Clients are computation, visualization or desktop nodes.

If a client wants to write or read a file, it asks the MDS for the OST objects involved in the I/O operation. The MDS tells the client the sequence of object identifiers and the stripe size. The client then transfers the file subsequently striped over the OSSs managing the OST handling the object corresponding to the object identifier. Analogous to read a file the client requests and receives the data from the OSSs.

Striping allows to aggregate the single I/O performance and storage capacity of OSSs, hence facilitates the scaling potential of Lustre. Because each OST performs its own locking, the locking performance scales with file system grow too. Lustre also provides high availability and recovery features like fail-over mechanisms to cope with the increasing probability of a failure in large systems. Furthermore Lustre supports Hierarchical Storage Management (HSM) to transparently move data to other storage tiers while maintaining the meta data on the MDSs.

2.2. Compression Algorithms

Compression algorithms reduce the amount of bits required to hold information. A compression algorithm performs lossy or lossless data compression. Lossy algorithms are commonly used for media types like photo, audio and video data. Redundancy and irrelevancy reduction are the core principles of lossy compression. Raw media information has several starting points for lossy compression which cleverly exploits the limitations of the human senses and reduces the amount of information without affecting the fidelity too noticeably[7]. Lossless compression algorithms are looking for statistical redundancy in the bit representation of the information. In contrast to lossy compression, lossless compression restores the exact initial bit sequences on decompression. Although lossy compression can be utilized to compress the floating point data of HPC simulation results[26], this work only focuses on lossless compression algorithms. Furthermore only general purpose compression algorithms are utilized. Alternatively there are many lossless compression algorithms specializing on certain data types like English text, images or audio[8].

Following are two definitions related to compression that are frequently used throughout this work:

- **Compression factor**
  The compression factor is defined by dividing the uncompressed size by the compressed size.
  $$\text{compression factor} = \frac{\text{uncompressed size}}{\text{compressed size}}$$
- **Compression savings**
  The compression savings describe what percentage of the original size is saved due to compression. This is useful because it directly indicates the percentage of any savings induced though compression. For example space or cost savings. Throughout this work the compression savings are stated as percentage in parentheses

following compression factors. The compression savings are defined as follows:

$$\text{compression savings} = 1 - \frac{1}{\text{compression factor}}$$

2.2.1. LZ4

LZ4 is a lossless compression algorithm focusing on very high compression and decompression speeds[9]. LZ4 is developed by Yann Collet and based on the LZ77 (Lempel-Ziv) compression algorithm[10]. LZ4 is provided as open source software and was initially released in April 2011[11].

The compression speed of LZ4 can be increased by selecting an acceleration factor ranging from 1 to 1000. Higher acceleration factors provide faster compression speeds at the cost of reduced compression factors. In addition to the regular version there is a high compression version (LZ4-HC) that achieves higher compression factors but has way slower compression speeds[12]. LZ4-HC has 12 compression levels, with each successive level trading more computation time for an increased compression factor. An advantage of LZ4-HC is that it uses the same format as LZ4, thus can also be decompressed with LZ4's extremely fast decompresser. On the official website the single thread compression speeds of LZ4 are stated to start around 625 MB/s at acceleration factor 1, whereas LZ4-HC at level 9 achieves 34 MB/s. The single thread decompression speed of both is about 3200 MB/s[13].

An important feature of both LZ4 versions is the "early abort" mechanism, which stops the compression process and just copies the data, if the achieved compression factor is to small[1]. This prevents wasting computation time for compression that will not even achieve a decent compression factor.

2.2.2. Zstandard

Zstandard (Zstd) is a lossless compression algorithm open sourced by Facebook in January 2015[14]. The algorithm is developed by Yann Collet and provides both high compression factors and high compression speeds. Zstd is based on LZ77 and combines recent innovations like Finite State Entropy (FSE) in a hardware optimized design to improve the trade-off between compression and performance[8]. This allows Zstd to clearly outperform zlib, the reigning compression algorithm since 1995, in both compression speed and factor[15]. The single thread compression speed of Zstd starts at around 200 MB/s and the decompression speed is about 500 MB/s, almost triple the speed of zlib[16]. Zstd has 22 compression levels, which allow to further increase the compression factor by trading compression speed. This enables a fine granular adjustment to fit the requirements of most compression use cases.

2.2.3. XZ

XZ is an open source compression software and file format using the lossless compression algorithm Lempel–Ziv–Markov chain algorithm (LZMA)[17]. The XZ file format was

released in January 2009. XZ achieves very high compression factors but in turn very slow compression speeds. XZ provides 9 compression levels having different compression speed and factor trade-offs. The compression speed starts at around 15 MB/s whereas the decompression speed is about 4 times higher.

2.2.4. Effective Data Rate

Performing compression prior to sending data through a data rate limited medium like a network can have a significant speedup potential depending on the compressibility of the data. A positive compression factor implies the amount of bytes required to fully represent the data is reduced by that factor. What again implies the amount of time required to get all bytes through a data rate limited medium is reduced by that factor as well. The mediums data rate stays constant but a transformation of the original data is effectively sent with a higher data rate because the information transfer completes faster. The enabling factor is the possibility of performing compression and decompression with a higher data rate than the mediums data rate. By doing so the medium's bottleneck data rate in the compression-medium-decompression sequence is effectively raised by the compression factor. This is referred to as the mediums effective or virtual data rate. The effective data rate is solely determined by the compression factor and calculated as follows:

$$f : \text{Factor}$$
$$s : \text{Speed}$$

$$s_{\text{effective}} = s_{\text{medium}} \cdot f_{\text{compression}} \tag{2.1}$$

The compress and decompress data rates have to be at least as high as the effective data rate to not limit it.

$$s_{\text{transfer}} = \min(s_{\text{effective}}, s_{\text{compression}}, s_{\text{decompression}}) \tag{2.2}$$

A compression factor of 2 for example would halve the data's space requirements and consequently half the transfer time with the same data rate thus the effective data rate would be double the mediums data rate. Even a smaller compression factor of 1.2 would increase the data rate respectively by 20%.

2.3. Related Work

There is some work done in designing adaptive compression frameworks and evaluating the feasibility of adaptive on-the fly compression to improve data transfer throughput over different networks. Generally the focus is on slower networks connections like cellular networks, Wireless Local Area Networks (WLAN) or consumer Internet connections. Also the data amounts are typically much smaller than what can be expected in the HPC area.

A basic approach on adaptive compression is pursued with the Adaptive Compression Environment (ACE)[18]. ACE is implemented into a Java Virtual Machine (JVM) to be able to intercept Transmission Control Protocol (TCP) socket communication of Java programs and to enhance the network throughput with adaptive compression. The adaptation process decides on basis of a 32 KB block size whether to perform compression or not. As compression algorithm solely zlib with compression level 1 is utilized. The compression decision considers the network speed and Central Processing Unit (CPU) load, obtained from the Network Weather Service (NWS)[19], and the achieved compression factor of the previous data block. Compression is only performed if the predicted time for the compression, transfer and decompression sequence is shorter than the transfer time without compression.

ACE pursues a very simple approach as the adaptation process only does a binary decision to perform compression or not. A more elaborated adaptation process using zlib's compression levels or additional compression algorithms would allow to adapt the compression performance to the networks and machines circumstances as well, thus increase the compression efficiency.

The authors of [20] designed an adaptive compression system named Datacomp. Datacomp is intended to increase the network throughput of general purpose computing devices in different network environments. The decision parameters are based the CPU load and frequency, network speed and a compressibility estimation of the data. The compressibility of a data block is estimated by counting unique bytes. As compression algorithms Lempel-Ziv-Oberhumer (LZO), zlib, bzip2 and XZ are utilized. To chose the most suited compression algorithm a lookup table with quantized combinations of the decision parameters is used. The compression algorithms are weighed by using each algorithm and quantized level at least twice to measure and save the achieved network throughput for every possible parameter combination in the lookup table. If the current parameter combination has already enough samples in the lookup table the compression algorithm, which in the past achieved the highest network throughput, is chosen. No compression is considered as well. The adaptation process happens in a interval of a not fixed block size between 32 and 512 KB.

The adaptation approach of Datacomp might work for slower network speeds, but with increasing network speeds the compressibility estimation would likely induce to much overhead. Also the concept of a lookup table might be adequate for use cases where very different data types and network conditions can occur, but not for HPC environments, where the network is stable and the data type is mainly floating point data[21].

In contrast to the presented designs this work focuses on the HPC field. The highly potent hardware in HPC compute nodes having many processor cores is able to provide much more resources for compression to keep up with the likewise much more potent network infrastructure. Secondly this work does not only consider increasing the network throughput but also achieving high data compression to save storage space and costs.

3. Design

Originally the idea of a decision-making logic was more about whether to perform compression at all or not and if compression is performed, what algorithm, level and how many threads to use. The compression decision and configuration should have been derived from different factors like the system load, network speed, user given file hints and pre-configured compression and storage costs. This concept was changed due to insights from evaluating compression algorithms, investigating application I/O write calls and prototypical tests over the course of this work.

Actively sensing the CPU load turned out to be inefficient and ineffective to base a compression configuration decision on. The process of retrieving the system's CPU utilization in a reasonable accuracy took too long and naturally only indicates the past utilization. The new approach described in this Chapter still takes CPU load into account but in a passive way.

Furthermore the idea to dynamically spawn compression threads proved to be inferior compared to dynamically adapting the compression algorithm and level on a constant number of threads instead. Also dynamic cost considerations and file hints are not really decisive because dynamic adaptation of the compression configuration being beneficial in almost all cases.

3.1. Client Side I/O Write Calls

An I/O write call is issued by an application and can be synchronous or asynchronous. I/O write calls are usually synchronous implying the application waits for the call to return and is blocked in the meantime. While waiting the machine's resources are idling and not being utilized by the application. Synchronous write calls are used if no effort in asynchronizing the write calls has been put into the application or it is not even possible. This is mainly because the whole memory is being written out as a checkpoint and can not be modified meanwhile to maintain a consistent copy on the storage. Without memory to compute on the application can only wait for it.

Asynchronous write calls are used when the application is still able to compute something meaningful while sections of the memory are written to storage. These computations can read from but not write to the memory being written. So there has to be memory not involved in the write call to write to. If asynchronous write calls can be utilized profitably is mostly determined by the type of application and application domain. Figure 3.1 pictures a synchronous write call and an asynchronous one with its asynchronous and synchronous portions.

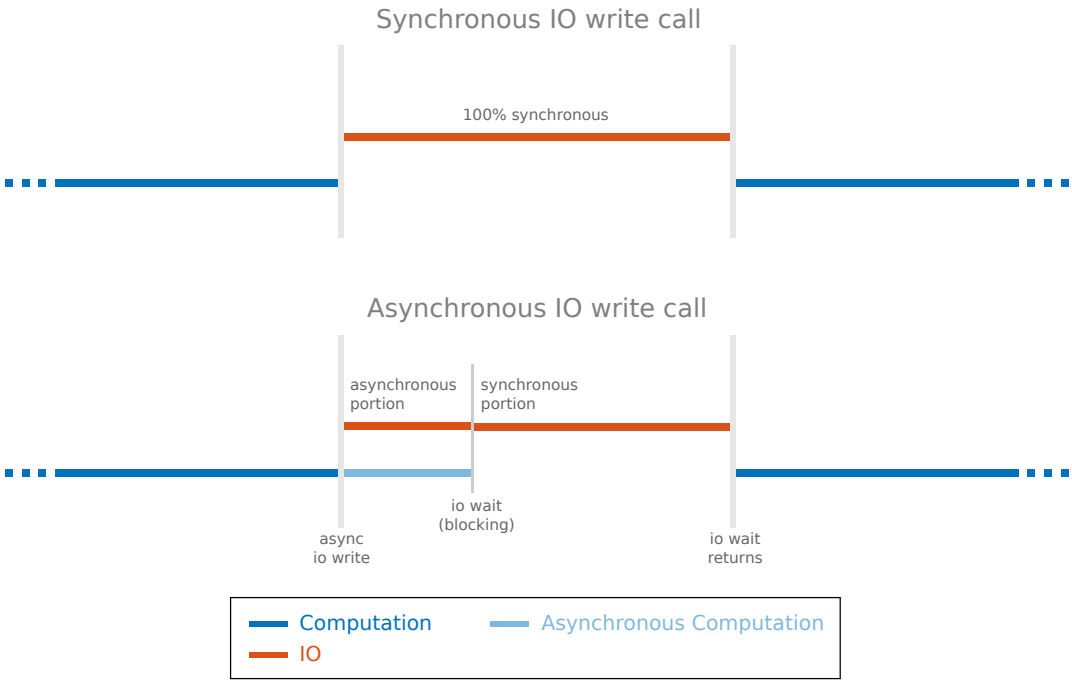

Figure 3.1.: A synchronous and an asynchronous I/O write call.

The asynchronism is usually limited to a certain degree. Due to the gap between computation and storage speeds the computation can usually not go on along the whole write call. Eventually the application might have to wait to write to the memory section still involved in the write call. At this point an asynchronous write call reaches its wait constraint and behaves exactly like a synchronous write call until it returns. In the case this would not happen, a 100% asynchronous write call would have been achieved.

However in our own experiments at best an asynchronous portion of up to 10-30% is realistic. The other 70+% will be a synchronous portion of an asynchronous write.

3.1.1. Compression during I/O Write Calls

As described earlier during a synchronous write call the system's resources are idling for the whole call duration. The same applies to the synchronous portion of an asynchronous write call. This period of unused resources is an ideal spot to utilize them for compression as the application runtime will not be negatively impacted at all.

The duration of an I/O write call is determined by the hard disk speed or with a distributed file system by the network or storage speed. Usually the network speed is the bottleneck. Consequently for a synchronous write call the compression of all passed data has to be at least as fast as the network speed to not prolong the write call and thus application runtime. This can be done with pipelined compression and dynamic compression speed adaptation utilizing different algorithms and levels to achieve the required speed.

For asynchronous write calls the same procedure will be applied. Commencing compression only as the synchronous portion begins and the machine's load drops is not effective though. This would imply only data written after the synchronous portion started will be compressed. However it is way more effective to compress all data with a lower compression factor, because of required speed adaptations due to CPU contention with the asynchronous application computation happing in parallel, than to only compress a portion of the data with a higher compression factor. The compression factor increase from uncompressed data to slightly compressed data is generally way higher than the compression factor increase resulting from increased compression levels or better compression algorithms.

Now naturally the asynchronous application computation will be slowed down due to CPU contention and thus prolonging the asynchronous portion and shortening the synchronous portion of the asynchronous write call. The asynchronous computation time roughly doubles if compression is done in parallel. But unless the asynchronous portion does not extend to the point where it would take more than 100% of the write calls time this still has no negative impact on the application runtime as the original write call duration is not prolonged. The extension of the asynchronous application computation is illustrated in Figure 3.2.

So for compressing all write call data there is exactly that amount of CPU time available that the synchronous portion of the write call would last without compression being performed. By choosing a suited compression algorithm and level the required compression speed can always be achieved. For synchronous write calls the available CPU time can be calculated by dividing the call data size by the network speed. The adaptive algorithm (described in detail in Section 3.2.1) does a similar thing to maximize the compression efficiency. However for asynchronous write calls there is no way of knowing or estimating how long the asynchronous portion will last. Hence how much CPU time will be consumed for that and has to be subtracted from the compression CPU time. This could possibly lead to spending too much CPU time for compression resulting

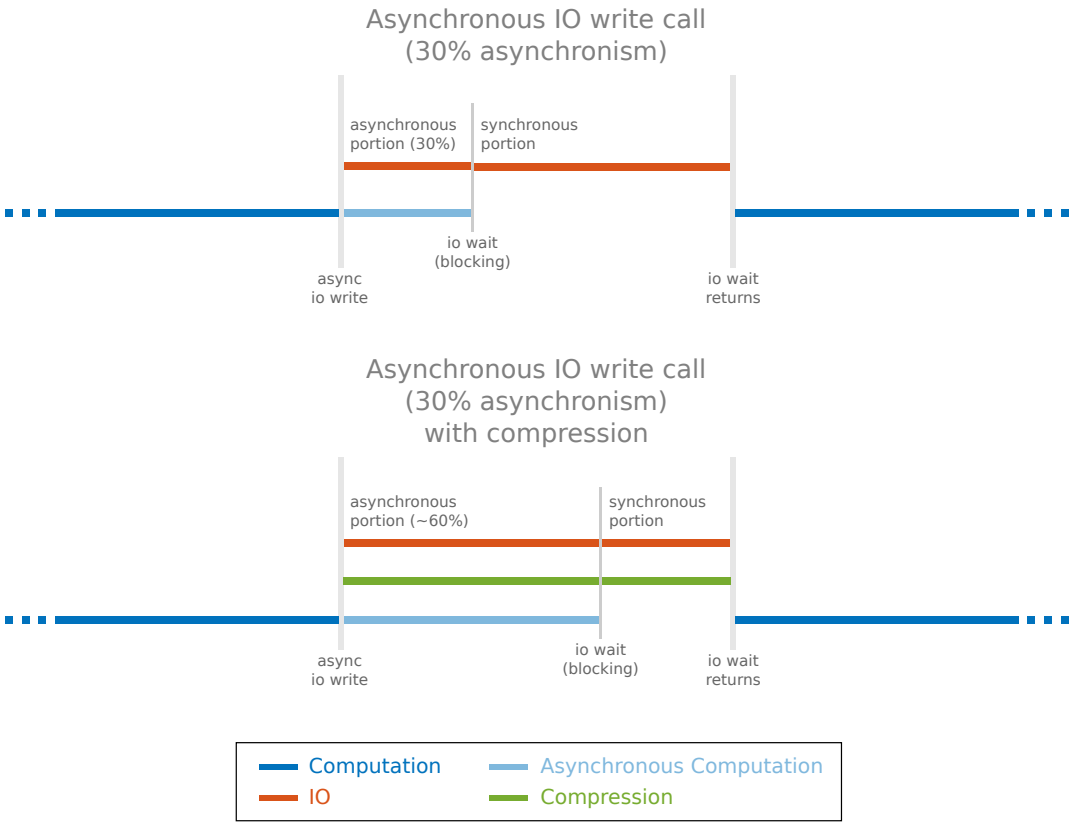

Figure 3.2.: Implications of performing adaptive compression during an asynchronous write call having less than 50% asynchronism.

in a prolonged write call and prolonged application runtime respectively. Figure 3.3 illustrates this case with an asynchronous write call having an asynchronous portion of 80%. Normally the now delayed computation (yellow line) would have been completed within the write call if no compression had been performed in parallel. But because the compression took away too much CPU time the remaining computation has to be completed afterwards thus prolonging the application runtime by the amount of computation time required to finish it.

As assumed earlier the asynchronous portion should be way under 50%. So supposing the asynchronous portion is exactly 50% and compression and asynchronous computation would share the CPU time in a fair way, this would prolong the asynchronous portion at maximum to 100% if the compression would even take the same amount of CPU time as the asynchronous computation. Which is perfectly fine because no prolonging of the write call happens.

There may be some context switching overhead by performing compression and computation in parallel. This overhead has to be subtracted from the compression time as well but is basically negligible. At least the same overhead would occur in somehow hard transitioning from asynchronous computation to compression. Also different computation units of a core may be utilized more efficient by running in

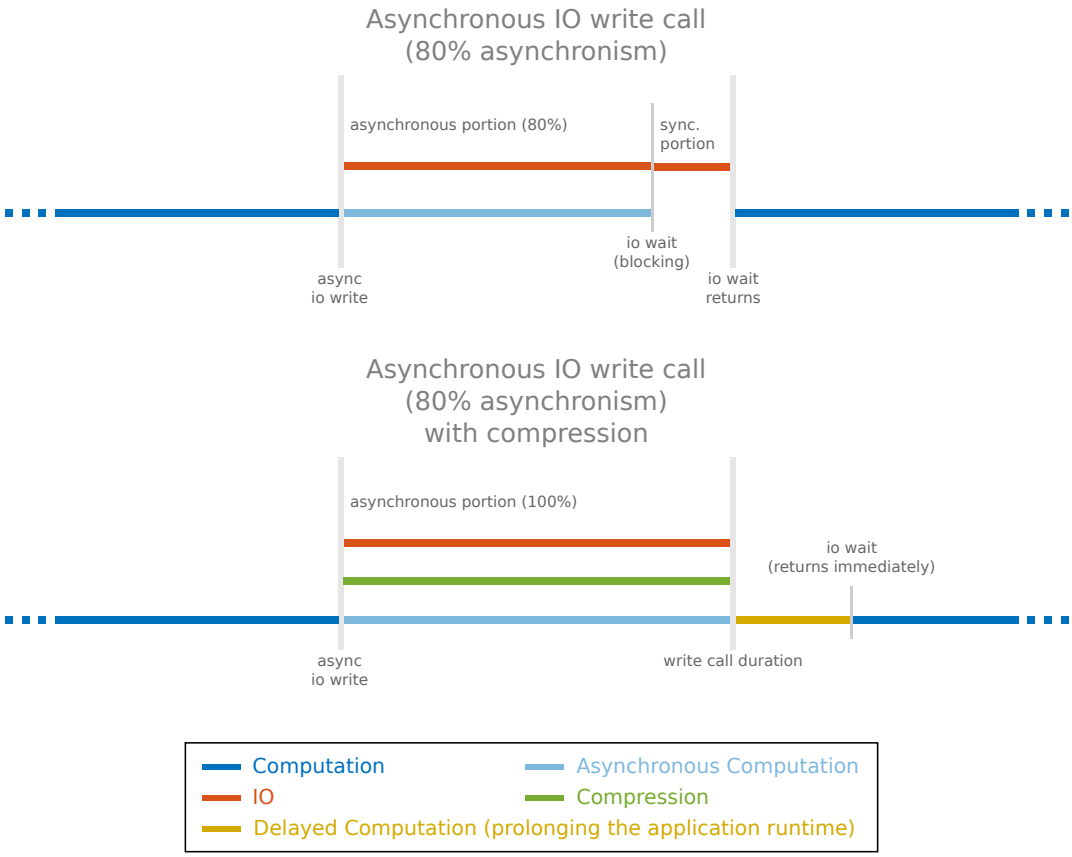

Figure 3.3.: Implications of performing adaptive compression during an asynchronous write call having over 50% asynchronism.

parallel with compression performing integer operations and asynchronous application computation normally performing floating point operations.

However running compression in parallel with asynchronous computation right as an asynchronous write call starts is crucial. This ensures the compression of all data and achieving the best overall compression factor with the given CPU time.

3.2. Pipelined Compression

To efficiently incorporate compression into an I/O write call it has to be done as a stream just in front of the write-out stream to the storage. To not prolong the write call the write-out stream has to stay saturated the whole time. Normally it is the bottleneck in the I/O stack and should not be slowed down further. To facilitate that compression and write-out are pipelined. Figure 3.4 illustrates the advantage of pipelined compression over sequential compression. Like that only a minimal write-out delay of the time needed to compress the first pipeline unit is introduced. However as data is getting compressed it is written in memory in its compressed form before getting transferred over the network. Implying the write call can return as soon as all data is compressed because all remaining

data not yet transferred can be sent asynchronously into the network. This cancels out the negative effect of starting the pipeline and even speeds up the write call as the compression buffer should be at least double the amount of data as the network can transfer in one pipeline cycle.

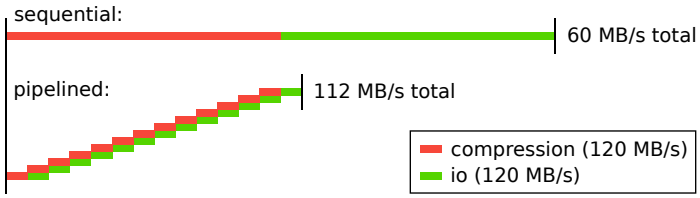

Figure 3.4.: Comparison between sequential and pipelined compression.

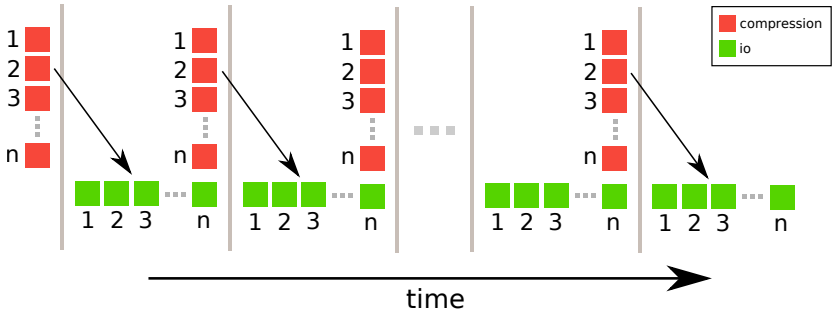

Figure 3.5.: Compression and I/O pipeline with n compression threads.

In Figure 3.5 the compression and I/O pipeline is visualized utilizing n compression threads. Compression is always executed equally distributed over all cores. So every pipeline cycle as many compression units (red squares) are processed in parallel as the machine has cores. The I/O write-out of compressed units (green squares) always happens in the successive pipeline cycle and is naturally done in series, as single network links are a serial medium. For systems with multiple network links the I/O units are in addition distributed in parallel.

The pipeline speed is always limited by the slowest pipeline stage. So for the most efficient operation the pipeline stages being compression and write-out have to be equally fast. Implying the parallel compression of n units should be as fast as their write-out in series. Slower compression would introduce periodic pauses in the write-out stream, thus not saturating the network link and prolonging the write call as the network was already the bottleneck. The effective network speed may compensate for the inefficiency tough (see Section 2.2.4). The pipeline speed represents the maximum achievable transfer speed and is defined as follows:

s : throughput

f : factor

$$s_{\text{transfer}} = \min(s_{\text{compression}}, s_{\text{effective network}}) \tag{3.1}$$

$$s_{\text{transfer}} = \min(s_{\text{compression}}, s_{\text{network}} \cdot f_{\text{compression}}) \tag{3.2}$$

3.2.1. Dynamic Compression Adaptation

The dynamic compression adaptation logic has two main objectives. Firstly to dynamically adapt the compression configuration to always provide a desired compression speed. And secondly to maximize the utilization of resources for compression. To achieve that compression algorithms having different levels like LZ4, Zstd and XZ are used. Naturally higher compression levels provide higher compression factors and require more time for the compression. More compression time in turn implies a slower compression speed and vice versa. Compression algorithms having different levels therefore allow to adapt the compression speed by adjusting the compression level. Simultaneously the compression factor is increased or decreased by varying the compression speed.

The compression speed has a special role in the compression pipeline as it is the only adaptable variable and is passively influencing the effective network speed via the related compression factor (see Equation (2.1)). Also the network transfer speed is directly dependent on the compression speed and effective network speed (see Equation (3.1)). So by intelligently adapting the compression speed different compression and pipeline characteristics can be achieved.

3.2.2. Compression Speed Implications

Figure 3.6 shows three sections (1, 3, 5) and two states (2, 4) in the relations between compression speed, physical network and effective network speed, having different implications for the write call duration and compression effectiveness. The compression algorithm, throughput and levels are just exemplary as the implications of the compression speed always apply.

In section 1 the compression speed (blue line) is faster than the network and effective network speed. If the compression speed is faster than the effective network speed the pipeline is limited by the effective network speed. Data can not be transferred faster than the effective network speed. Having faster compression is inefficient. To optimize the pipeline efficiency and ultimately the transfer speed, the compression level should be increased. Raising the compression level generally implies slowing down the compression speed but in turn getting a higher compression factor and thus a higher effective network speed. Both outcomes bring the pipeline closer towards an optimal situation being for example state 2.

At state 2 the compression speed is as fast as the effective network speed hence the pipeline is working in the most efficient way. The maximum data transfer speed is reached at state 2. It can not be increased any further as raising the compression level would put the compression speed behind the effective network speed and lowering the compression

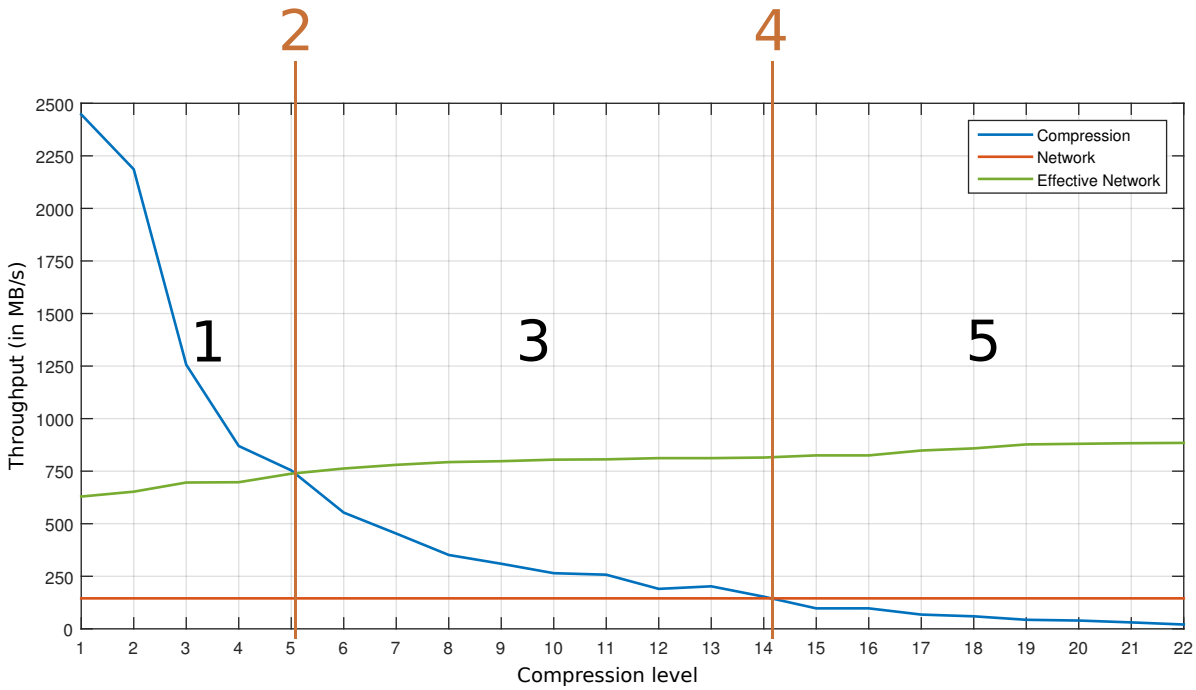

Figure 3.6.: Compression speed sections.

level would lead back to section 1. State 2 has to be aimed for if the maximum transfer speed is desired.

In section 3 the compression speed is faster than the network speed but slower than the effective network speed. This section is inefficient again as the compression speed being slower than the effective network speed is limiting the maximum transfer speed. On the other hand this section is trading transfer speed for a slightly increased compression factor later on storage. If the desired goal is to achieve the highest compression factor possible without affecting the application runtime negatively, state 4 would be a more consistent way though.

At state 4 the compression speed is as fast as the network speed. Implying the compression is done just in time to still saturate the network thus the data transfer is as fast as it would be without compression. This state allows for the maximum compression factor without prolonging the write call and consequently the application runtime.

In section 5 the compression speed is slower than the network speed. This slows down the write call and consequently the application runtime.

3.2.3. Compression Strategies

The above described compression speed implications show that different compression strategies can be pursued. The effective network speed in combination with a high compression speed can raise the network transfer speed considerably. A higher network transfer speed implies shorter write calls as less time is required to transfer all data over the network. This speeds up the application especially if multiple write calls are done in

between computation phases. So focusing on the write call speedup is a very reasonable strategy.

Besides speed, the focus can be towards the compression ratio. A higher compression ratio can save more storage space and costs. But even by focusing on the compression ratio, different trade-offs between compression ratio and speed have to be considered. Prioritizing only the compression ratio would slow down the application to an unfeasible level. So after speeding up the write call in the first strategy, a reasonable second strategy is to maintain the original write call duration and to achieve a slightly higher compression factor. The compression would be user transparent as the application would run as fast as if no compression would have been performed.

Alongside shortening or maintaining the write call duration another imaginable compression strategy is to deliberately prolong a write call by a certain percentage to achieve even higher compression factors than the two strategies before. This will of course prolong the application runtime but at least in a controlled way. Prolonging the write call by a defined fixed factor would still allow predicable write call durations and reasonable application run-times.

3.2.4. Adaptation Advantage

To be able to pursue any of the above defined strategies, compression speed adaptation is absolutely necessary. A static compression configuration is not capable of maximizing the transfer speed, maintain the write call duration or prolong it by a defined factor. To achieve that fine grained adaptations of the compression speed are required. Compression speed is very dependent on the data structure and naturally on asynchronous CPU load. The data structures are varying in every write call and every application. So it is impossible to set up an efficient static configuration for multiple write calls or even applications. Also static configurations require tests for feedback to fine-tune it. Because of that it is more reasonable to set up a conservative static configuration with very inefficient but always fast enough compression.

Compression adaptation has a second advantage. Because of the fine grained adaptations the compression is better adapted to the file structure and is more efficient if the structure has radical changes in a single write call. Static compression has to find a single configuration for the entirety of the data structure whereas adaptive compression can handle different data sections with adapted configurations.

3.2.5. Adaptation Basics

The dynamic compression adaption algorithm is adapting the compression configuration in between every data block to accomplish the objectives of a chosen strategy. To facilitate that, the achieved compression speed and compression factor of every data block is measured. The compression factor in combination with the physical network speed allows to calculate the effective network speed (see Equation (2.1)). The system's physical network speed is specified as a configuration parameter. The compression strategy is pre-configured as well or dynamically chosen through a file hint. The adaptation then

works on adapting the compression configuration based on the measured feedback. The algorithm and level of the first compression blocks should be preassigned to reasonable values matching with the physical network speed.

Maximizing the Network Transfer Speed

The network transfer speed can be higher than the physical network speed if the data is somewhat compressible and the compression can be performed fast enough. A higher network transfer speed implies a shorter write call as less time is required to transfer all data over the network. This speeds up the application, especially if multiple write calls are done in between computation phases.

To maximize the network transfer speed, the compression speed has to be at all times as close as possible but never lower than the effective network speed. Higher compression speeds are suboptimal because slower compression generally implies a higher compression factor what in turn facilitates a higher effective network speed (see Equation (3.1)). The dynamic compression adaption algorithm does this by comparing the measured compression speed with the calculated effective network speed. The compression speed can be higher or lower than the effective network speed. If the compression speed is higher, the compression level is increased. Increasing the compression level will decrease the compression speed and increase the compression factor. Bringing the compression speed and effective network speed together from both sides. Analogous if the compression speed is lower than the effective network speed, the compression level will be decreased. Decreasing the compression level will increase the compression speed and decrease the compression factor, again bringing the compression speed and effective network speed closer towards each other. If the minimum or maximum compression level of the used algorithm is reached the algorithm is changed to a respectively fast or slower one. This procedure will oscillate the compression speed around the effective network speed because the compression speed is affected much stronger by compression level changes. Figure 3.7 illustrates the adaptation process.

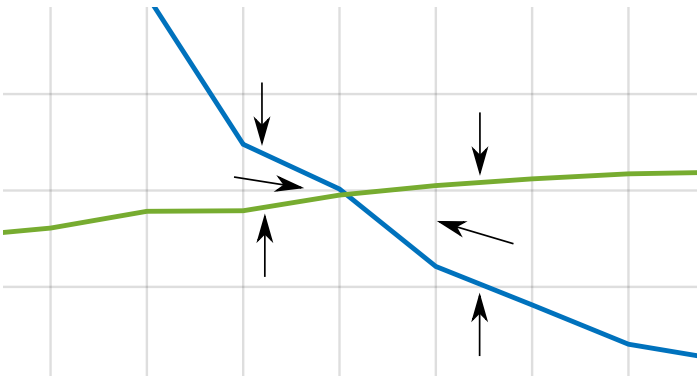

Figure 3.7.: Compression speed adaptation to the effective network speed.

Furthermore drastic compression speed changes due to data structure changes or asynchronous background load are implicitly included in the regulation process. Oscillation is

not optimal because every undercutting will decrease the overall network transfer speed. So the compression speed should only be decreased if it exceeds a certain threshold above the effective network speed to avoid undercutting it. This simple algorithm continuously strives to achieve the highest network transfer speed and consequently the most sped up write call possible. Furthermore the compression factor attainable with this compression strategy is very respectable.

Maximizing the Compression Factor

Without compression the write call duration is determined by the the physical network speed, supposing it is the bottleneck in the storage stack. So to maximize the compression factor under the premise not to prolong the write call duration, the compression speed has to be as fast as the physical network speed. This ensures to invest the maximal amount of time for compression while still keeping the original write call duration. In addition the network load is reduced by the factor of the compression factor. The effective network speed potential is of course dismissed in doing so.

To achieve this behavior the dynamic compression adaptation algorithm has to do the exact same adaptation process like before in maximizing the network transfer speed, but this time the compression speed has to be adapted to the physical network speed. The physical network speed is a static value unlike the effective network speed, which is also influenced by compression level changes, but that does not alter the adaptation process.

Prolonging Write Calls

To prolong the write call in a predicable way the compression speed should be as fast as a certain percentage of the physical network speed. Adapting the compression speed to half the physical network speed would double the write call duration. Consequently prolonging the write call by for example 25% will be achieved by adapting the compression speed to 80% of the physical network speed. The relation is calculated as follows:

$$\frac{1}{\text{write call duration factor}} = \text{network speed adaption factor}$$

Using the mentioned values as example:

$$\frac{1}{1.25} = 0.8$$

This leads to compression being performed slower than the network speed thus behaving as an artificial bottleneck and dedicating 25% more time to compression. Doing so will of course affect the application runtime negatively depending on the number of write calls done. Again the same dynamic compression adaptation algorithm is suited to achieve this strategy. Here the compression speed has to be adapted to the percentage of the physical network speed.

3.3. File Hints

File hints are file specific hints for the Lustre file system specified by the application developer via the Lustre `llapi_ladvise` glsAPI. So far there only exist two hints regarding the file caching behavior on the server side. However the API has been designed to be greatly extensible. File hints are bound to file sections which are specified by the position of the first and last byte. The API facilitates up to 2^{16} different hint types and provides 14 bytes of information to be attached to a single file hint.

It is desirable to have the same possibility of handling file hints to the decision making logic. So the `llapi_ladvise` API has been prototypically implemented with additional file hints which enable the application programmer to provide explicit information about file properties and signal desired behavior to the decision making logic.

The application programmer should have the absolute knowledge about the file structure, properties etc. and may want to support the decision making logic in its execution by pointing out information that may be difficult for the decision making logic to recognize automatically.

The newly added file hints are described in the following.

no compression Already compressed data can not be efficiently compressed further. So in the case of an application doing compression on its own the file hint "no compression" should be set for the sections of a file which will already be compressed. The same applies for sections of random data.

Compressed or random data could be automatically detected by evaluating the compression factor after compressing a data block. It will be 1 or less. This check can not be done effectively though, because the data blocks have to be tested periodically to find the end of a non-compressible data section. This either leads to the skipping of some compressible data using big intervals or wastes resources with worthless test compression on small intervals. By setting this file hint the decision logic will not attempt to compress the specified file sections thus saves resources.

write once Files that are written once and only read prospectively are indicated by this file hint. Higher compression factors and algorithms having faster decompression could be used considering the original idea of the decision-making logic. For the evolved dynamic compression adaptation this file hint does not make much sense. It could indicate to use the strategy of maintaining or prolonging the write call duration for higher compression though.

hot data Data that is read and written often. This file hint could indicate to use the write call speedup strategy. Also thinkable would be to use no compression at all as no decompression has to be performed but that would dismiss any potential write call speedup as well.

3.4. Cost Considerations

To figure out if compressing data before storing it is economically viable some cost considerations have to be done. Naturally the costs of compressing data have to be lower than the savings of requiring less storage space for it. Calculating these costs is not that straight forward because they are dependent on multiple factors like the compression factor, storage duration and system costs which itself are difficult to calculate. In the following an approach to approximate the compression and storage costs is done based on exemplary data from the Deutsches Klimarechenzentrum (DKRZ).

3.4.1. Compression Costs per GB

The compression costs are composed of energy, system investment and maintenance costs. While performing compression the machine's consumed energy rises above idle. The compression efficiency can not generally be assumed because it is heavily dependent on the structure of the data. So for this exemplary cost calculation a very conservative approximation of taking 5 seconds and fully saturating the machine's CPU cores for the compression of 1 GB of data will be supposed. Furthermore a dual socket Intel server with an investment cost of 7000€, a system lifetime (time until the system gets replaced by a newer one) of 3 years and a power consumption of 300 Watt will be supposed as exemplary machine.

The investment costs portion of compressing 1 GB of data (5 seconds of system time of 3 years of expected system lifetime) are

$$7000€ \cdot \frac{5\text{s}}{3 \cdot 365 \cdot 24 \cdot 3600\text{s}} \approx 0.00037€.$$

The maintenance costs are supposed to be about 15% of the investment costs per year. Scaled to 5 seconds system time the maintenance cost portion is

$$7000€ \cdot 15\% \cdot \frac{5\text{s}}{365 \cdot 24 \cdot 3600\text{s}} \approx 0.00017€.$$

For cooling 25% are added to the energy consumption. As energy costs 0.14€ per kilowatt hour are supposed. So the energy costs of 5 seconds system time are

$$0.375\text{kWh} \cdot 0.14\frac{€}{\text{kWh}} \cdot \frac{5\text{s}}{3600\text{s}} \approx 0.000073€.$$

In total compressing 1 GB of data would cost about

$$0.00037€ + 0.00017€ + 0.000073€ \approx 0.00061€.$$

In the context of the decision logic running on the client this cost can in most cases be relaxed. Compression is only done during write calls of a running application. These write calls are often synchronous. Even if they are asynchronous unless they are 100%

asynchronous which is very unlikely there is a synchronous portion in it. Normally the absolute system time required for the compression of the whole write call data completely fits within the synchronous portion of the write call. This means no additional system time is required for the compression due to the nature of the application having to wait for the synchronous portion (I/O) of the call anyway. So the idle energy, investment and maintenance costs can be excluded from the compression costs calculation as they are already accounted (or "payed") by the application. Only the energy costs above idle level required for the compression have to be considered as total compression costs. Assuming the system is idling at $\frac{1}{4}$ of the maximum power consumption the costs for compressing 1 GB would drop by an order of magnitude to only

$$0.000073€ \cdot \tfrac{3}{4} \approx 0.000055€.$$

3.4.2. Storage Costs per GB

Like the compression costs the storage costs are composed of the investment, maintenance and energy costs of the storage system. Supposed is a 1 PB storage system with an investment cost of 100,000€ a system lifetime of 3 years and a power consumption of 3000 Watt. Again for maintenance 15% of the investment costs per year are assumed. So investment and maintenance costs combined for storing 1 GB per year are

$$100,000€ \cdot (\tfrac{1a}{3a} + \tfrac{15}{100}) \cdot \frac{1\text{GB}}{1,000,000\text{GB}} \approx 0.048€.$$

The energy costs for 1 GB per year with a kilowatt hour costing 0.14€ are

$$365 \cdot 24 \cdot 3.75\text{kWh} \cdot 0.14\tfrac{€}{\text{kWh}} \cdot \frac{1\text{GB}}{1,000,000\text{GB}} \approx 0.0046€.$$

Totaling the costs for storing 1 GB per year are

$$0.048€ + 0.0046€ \approx 0.053€.$$

This approximation does not factor in access and write times. But that would only benefit compression more because of shorter times.

3.4.3. Impact of the Compression Factor

The only goal of compression is to achieve some sort of positive compression factor. The factor alone determines the efficiency and worthwhileness of the compression. The compression factor on the other hand is determined by the structure of the data.

The above cost approximations can only be compared in combination with a compression factor. The compression factor determines how much storage space is saved thus how much money is saved over the file lifetime. The relation between compression factor and days of storage until the investment of compression redeems itself is shown in Figure 3.8.

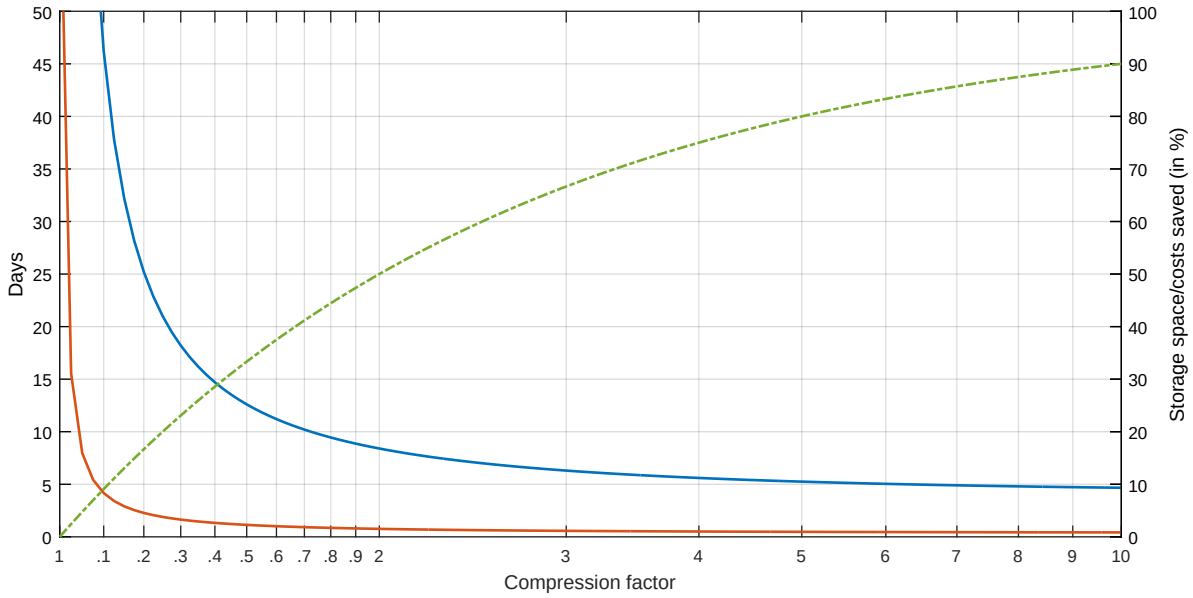

Figure 3.8.: Days until compression costs are redeemed.

Compression factor	1.05	1.1	1.2	1.3	1.4	1.5	2
Cost savings	25€	48€	88€	122€	151€	177€	265€

Table 3.1.: Cost savings of storing 40 TB of data for 3 months compressed.

The blue line represents the compression costs including investment, maintenance and full energy costs. Whereas the red line indicates the special case where only the power over idle is considered as compression costs (see Section 3.4.1). The graphs reveal that even a small compression factor of 1.1 would be worthwhile after only 46 days of storage despite having full compression costs. The red line would even start profiting with a factor of 1.05 after one week of storage.

Given that scientific data is normally stored for many years from a costs perspective compression is absolutely worthwhile doing. Even more when performed on the client during synchronous or not completely asynchronous application write calls as this drops the effective compression costs substantially. The only requirement is a positive compression factor what can generally be achieved. So the potential to save storage space and costs is very high.

Supposing an application calculates 40 TB of data to be stored over a 3 months time period before it gets migrated to a tape archive. Without compression the storage costs would be about 530€. The cost savings for different compression factors are shown in Table 3.1 while the green line in Figure 3.8 indicates the savings in storage space and respectively costs in percentage relative to the compression factor.

Compression naturally brings the costs of decompression, that of course has to be factored in as well. Decompression is normally multiple times faster than compression thus more energy efficient. So its very unlikely that the costs of decompression will ever

outweigh the storage saving profits. But the necessity to decompress limits to a certain small compression factor higher than 1 as the decompression costs may not be once per file like the compression costs.

Considering this Section the decision logic does not even need to factor in cost considerations because compression will always be worthwhile doing by the fact that computation time and hardware is generally cheaper than storage.

3.5. Compression Prerequisites to be worthwhile

Performing compression on the client can be driven by two intentions. On the one hand compression can be used to increase the network speed to speed up the applications and save computation costs. On the other hand the clients resources can be utilized for compression to relieve the compression task from the servers and relax the network contention. Both cases have prerequisites and limitations.

3.5.1. Application Speedup

Speedup Factor

Equation 3.3 expresses the dependencies of the speedup factor. The speedup factor f_{speedup} describes the factor by which the write call is sped up.

$$f : \text{Factor}$$
$$s : \text{Speed}$$

$$f_{\text{speedup}} = \min\left(f_{\text{compression}}, \frac{s_{\text{compression}}}{s_{\text{network}}}, \frac{1}{2 \cdot f_{\text{asynchronism}}}\right) \tag{3.3}$$

The compression factor $f_{\text{compression}}$ is responsible to achieve an effective network speed higher than the physical network speed, thus naturally limiting the possible speedup factor. Secondly the compression speed has to be higher than the physical network speed to be enable the utilization of the effective network speed. Thus the relation between the compression speed and network speed $\frac{s_{\text{compression}}}{s_{\text{network}}}$ limits the speedup factor (see Section 2.2.4). Double the compression speed than network speed for example limits the speedup to a factor of 2. The third limiting factor only applies to asynchronous write calls. The asynchronous application computation will be slowed down through CPU contention with the compression. This roughly doubles the time needed for the asynchronous computation thus might limit the speedup, because the write call is not completed till all data is transferred and the asynchronous computation is finished. If the asynchronous write call would have 50% asynchronism without compression being done, performing compression would roughly double the asynchronous computation to last 100% of the original write call, thus preventing a speedup. In practice the prolonging factor of the asynchronous computation is a little less than 2, as the compression does not use exactly 50% of the CPU time during the asynchronous computation, for example because of short waits for the network.

Compression Costs

Equation 3.4 expresses the CPU time used for compression and equation 3.5 expresses the compression costs.

$$
\begin{aligned}
T &: \text{Compression CPU time} \\
C &: \text{Compression Costs} \\
E &: \text{Energy Costs} \\
D &: \text{Write Call Data} \\
s &: \text{Speed} \\
f &: \text{Factor} \\
p &: \text{Power}
\end{aligned}
$$

$$
T = \frac{D}{s_{\text{network}} \cdot f_{\text{speedup}}} \cdot (1 - f_{\text{asynchronism}} \cdot f_{\text{speedup}}) \tag{3.4}
$$

$$
C = T \cdot (p_{\text{compression}} - p_{\text{idle}}) \cdot E \tag{3.5}
$$

The CPU time required for compression is determined by the write call duration and asynchronism factor. The possibly sped up write call duration is expressed by $\frac{D}{s_{\text{network}} \cdot f_{\text{speedup}}}$. The write call data divided by the possibly increased network speed gives the time required for the transfer, thus write call duration and CPU time available for compression. For asynchronous write calls the asynchronous computation consumes CPU time too, hence has to be subtracted from the compression CPU time. The asynchronous application computation is not affected by the network transfer speedup and expressed as $\frac{D}{s_{\text{network}}} \cdot f_{\text{asynchronism}}$.

The compression costs are calculated by multiplying the time used for compression with the power required for compression multiplied with the energy costs. Like mentioned in Section 3.4 as power consumption only the power over idle is considered as compression costs, as the idle power consumption is already incorporated into the application costs.

When to compress

The costs of compression intended to speed up the application are the lowest compression costs possible as the machine is idling anyway (see Section 3.4). Furthermore the byproduct of performing compression with the intention to speedup the application are computation time and storage savings, which will quickly redeem the compression costs. So costs do not influence the compression decision at all.

Supposing the compression speed is faster than the network speed and the compression factor is over 1, compression for a synchronous write call is always worthwhile. The most natural limitation is of course the compression factor, which is exclusively dependent on the data structure and not foreseeable. However in most cases the application data should be at least somewhat compressible, if not very compressible because of redundant information of for example adjacent areas in the data structure of simulations. Also

incompressible data could be automatically detected or signaled with file hints by the developer to prevent compression. Even if incompressible data gets compressed that usually does not mean the compression speed would be too slow. So the negative effect would be rather the inefficient compression and energy waste instead of a slow down of the application.

The other natural limitation is the compression speed, which of course has to be fast enough. The actual compression speed is very influenced by the data structure as well, but important is that the average compression speed is faster than the physical network speed. Compression algorithms like LZ4 that can reach the memory speed limitations of a CPU socket will guarantee that.

Asynchronous write calls have the same prerequisites for compression plus an additional limitation. The asynchronous application computation occurring during a portion of the asynchronous write call and the compression being performed in parallel mutually influence each others performance. Only half of the CPU time will be available for compression during the asynchronous portion of the write call. This roughly halves the compression speed, thus requires the average normal compression speed to be twice as fast as the physical network speed to not slow down the write call and application runtime during the asynchronous portion of the write call.

Like the compression speed is halved, the asynchronous computation speed is halved as well. Hence if the asynchronous computation initially takes more than 50% of the original write call duration, performing compression in parallel could extend the computation to over 100% of the original write call duration, thus prolong the application runtime. If the asynchronous computation lasts exactly 100% of the original write call duration no speedup is achieved but at least compression savings might be achieved.

This are only rough measures to estimate the prerequisites. Because of hyper-threading and scheduling effects the parallel execution of compression and application computation might be performed slightly more efficient. However in general asynchronous write calls have stronger requirements on the compression speed capabilities and the asynchronism should not be higher than 50% to facilitate application speedup.

3.5.2. Compression Savings

If the objectives are compression savings no write call speedup should occur, instead the write call could be prolonged by a defined prolonging factor. It is important that the prolonging factor can be exactly meet to have a predicable write call duration. A prolonging factor of 1 should maintain the original write call duration. A higher prolonging factor should prolong the write call duration by exactly that factor.

Prolonging Factor

Equation 3.6 expresses the dependencies of the prolonging factor.

f : Factor

s : Speed

$$f_{\text{prolonging\_actual}} = \max\left(f_{\text{prolonging}}, \frac{s_{\text{network}}}{s_{\text{compression}}}, \frac{f_{\text{prolonging}}}{2} + f_{\text{asynchronism}}\right) \tag{3.6}$$

The prolonging factor is dependent on the compression speed and the asynchronism if the write call is asynchronous. A write call should be prolonged exactly by the prolonging factor $f_{\text{prolonging}}$ and not longer. Achieving the desired prolonging factor and write call duration would be prevented, if the compression speed in relation to the physical network speed $\frac{s_{\text{network}}}{s_{\text{compression}}}$ would be too slow. This constraint is weaker than the compression speed requirement for a write call speedup and could be alternatively expressed as $s_{\text{compression}} \geq \frac{s_{\text{network}}}{f_{\text{prolonging}}}$.

The asynchronism factor could prolong the write call duration longer than desired as well. If for example the asynchronism factor is 1 corresponding to a 100% asynchronous write call, performing compression with the intention to prolong the write call by a prolonging factor of 1.5, would not be possible as the asynchronous application computation would be extended by a factor of 1.75 through being partially affected by CPU contention with the compression. The asynchronous computation duration is not doubled, because the compression stops after the write call is prolonged by a factor of 1.5 and the remaining asynchronous computation works at normal speed.

Compression Costs

A prolonged write call is composed of the original write call duration and potentially additional time for compression (see Equation (3.9)). The compression costs for the original write call duration are lower, because the application is accountable for most costs regarding that period. The prolonged period is exclusively utilized for compression thus the whole machine costs have to be considered. The compression CPU time of the original write call duration is calculated by dividing the write call data by the network speed and subtracting the CPU time used for the asynchronous application computation (see Equation (3.7)). The additional CPU time for compression is calculated by multiplying the original write call time with the prolonging factor and subtracting the original write call duration (see Equation (3.8)).

D : Write Call Data
C : Compression Costs
E : Energy Costs
M : Machine Costs (Energy, Maintance, Investment)
f : Factor
s : Speed
p : Power Consumption
t : Compression CPU time

$$t_{\text{original}} = \frac{D}{s_{\text{network}}} \cdot (1 - f_{\text{asynchronism}}) \tag{3.7}$$

$$t_{\text{additional}} = \frac{D}{s_{\text{network}}} \cdot (f_{\text{prolonging}} - 1) \tag{3.8}$$

$$C = t_{\text{original}} \cdot (p_{\text{compression}} - p_{\text{idle}}) \cdot E + t_{\text{additional}} \cdot M \tag{3.9}$$

When to compress

A prolonging factor of 1 implies the original write call duration is maintained as if no compression were done. Like speeding up the write call through compression, maintaining the write call duration does not affect the application in a negative way. Maintaining the write call duration is a special case of speeding up the write call having a speedup factor of 1. Therefore the prerequisites for maintaining the write call duration are the same as for speeding it up, but with a slightly weaker compression speed requirement. The compression speed has to be only as fast as the physical network speed instead of faster.

All in all compression with the intention to maintain the write call duration is always worthwhile for synchronous write calls, if the compression factor is over 1 and the compression speed is at least as fast as the physical network speed. For asynchronous write calls the compression speed has to be roughly double the physical network speed to achieve at least the equivalent network speed under CPU contention. Also the asynchronism factor has to be less than 0.5.

Prolonging the write call is different, as the application runtime is negatively affected and the compression costs increase considerably, as not only the above idle energy costs have to be considered but the whole machine costs (see Section 3.4). The worthwhileness of prolonging the write call is evaluated in Section 5.2.

4. Implementation

The dynamic compression adaptation logic was prototypically implemented in C in the userspace. It is intended to be integrated into distributed file systems later. To develop and evaluate the compression logic in the userspace some wrapper functions simulating the file system infrastructure were implemented as well. The logic provides the modified `llapi_ladvise` API (described in Section 3.3), a synchronous and an asynchronous write function. Listing 4.1 lists the public functions provided by the prototypical implementation to applications.

```
1 int llapi_ladvise(int fd, unsigned long long flags, int
    ↪ num_advise, struct llapi_lu_ladvise *ladvise);
2 ssize_t lwrite(int fd, const void *buffer, size_t nbyte,
    ↪ off_t offset);
3 int lwrite_async(int fd, const void *buffer, size_t nbyte,
    ↪ off_t offset);
4 void lwait();
```

Listing 4.1: Public functions of the prototypical implementation

4.1. Control Flow

A HPC application will usually create or open a file with the intention to write its computation results to a data storage. Just after opening a file the application has the possibility to issue multiple file hints to the compression logic via the `llapi_ladvise` API (see Section 3.3). The compression logic will save the hints dedicated to their file descriptors and return the control to the application. Eventually the application will start to actually write to a file by calling either the synchronous or asynchronous write functions provided by the prototype implementation. In the case of an asynchronous write call a thread for handling the decision logic is spawned and the control flow is returned back to the application.

The decision logic thread first checks if file hints regarding sections of the passed data and given file descriptor are present. If no file hints that negate the compression of all passed data are found, resources for the compression process of that specific write call are allocated. A buffer-pool for the compression threads and a I/O-pool for the I/O thread are initialized. Then a pre-configured amount of memory is allocated as compression buffers and enqueued into the buffer-pool. The pools are implemented as asynchronous queues and can hold the pointers to the memory sections containing the compression buffers. A compression buffer is a data structure having a memory

section to compress the data into and two meta-data variables for the data's compressed length and used compression algorithm. The memory section has to have at least the size of the compression block-size. A slightly longer memory section would be required for incompressible blocks, as they can exceed their original size through compression. However incompressible data would be copied over by the original data to transfer them marked as uncompressed. This is more efficient and will prevent problems with the later integration into Lustre.

The compression process is then started by spawning one compression thread for every CPU core the machine has and one or multiple I/O threads for the transfer over the network. Multiple I/O threads might be required for network technologies having multiple links like InfiniBand.

Compression

Each compression thread starts by dequeuing a buffer from the buffer-pool. After acquiring a buffer the thread compresses its current data block into it. The buffer now containing the compressed data is then enqueued into the I/O-pool. The I/O thread continuously dequeues buffers from the I/O-pool and transfers them over the network. Implemented into Lustre this could be the Remote Procedure Calls (RPCs) to the OSSs. Transferred blocks are then immediately enqueued back into the buffer-pool to be utilized by the compression threads again. Figure 4.1 visualizes the data flow and buffer memory cycle between compression threads and I/O thread.

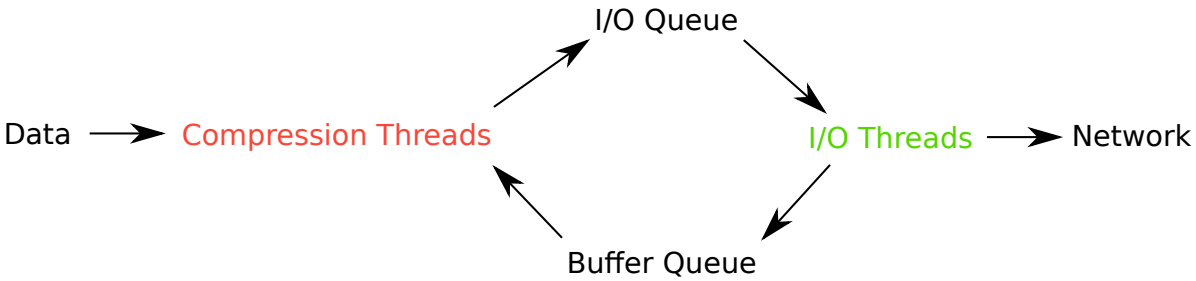

Figure 4.1.: Memory utilization.

The buffer memory interaction between the compression threads and I/O thread is necessary because usually there is not enough memory available on the machine to compress all the passed write call data at once. The whole write call could be cached, compressed and transferred asynchronously in the background, if there would be enough memory available. However in the HPC area the machine's total memory is usually matched with the memory needs of the typical applications frequently running on them. Aside from that there is no need for much compression buffer memory when compression and I/O is pipelined. The memory interaction implements the compression and I/O pipeline described earlier in Section 3.2.

The pipeline cycles are not strictly bound though, because the I/O thread releases the compression buffers successively instead of all at once every pipeline cycle. The

compression threads can therefore be some cycles ahead of the I/O thread, increasing with the configured amount of compression memory. The pipeline requires a minimal amount of compression buffers to run efficiently.

$$t \cdot 2 \leq b \tag{4.1}$$

t : number of compression threads

b : compression buffers

Following (4.1) ensures the working principle of the pipeline, because at least $2 \cdot t$ compression buffers are necessary to let the compression threads compress in the same pipeline cycle as the I/O thread. With this minimum amount of compression buffers the compression threads can get two cycles ahead of the I/O thread.

After Compression

If the application used an asynchronous write call it can continue to compute while the passed write call data is getting compressed and transferred. Normally at some point the application computation has to write to memory sections that where involved in the asynchronous write call. Before doing that the application has to check if the asynchronous write call is already completed. Otherwise an inconsistent version of the memory would be stored. To check if the asynchronous write call has finished the `lwait` function has to be called. This function will immediately return the control flow if the asynchronous write call already completed or block the application until then otherwise.

After all write call data is compressed by the compression threads the control flow immediately returns to the application, while the remaining compressed blocks, buffered in the I/O-pool, are transferred over the network asynchronously. Because the compressed data is buffered in dedicated memory, the compression buffers, no data inconsistencies can occur through the application continuing to write on its memory. This compensates for the initial network transfer delay induced by waiting for the first data blocks to be compressed until they can be transferred. Depending on the configured amount of compression buffers this can speed up the write call by acting as a write call cache. As mentioned earlier the compression threads can get two cycles ahead of the I/O thread with the minimum number of compression buffers.

After the asynchronous transfer of the remaining compressed data in the I/O-pool completed, the allocated resources are freed and all compression logic threads terminate. The file hints still persist though as the application could issue additional write calls over its runtime. The file hints should be dismissed by Lustre when the file descriptor is closed by the application.

4.2. Adaptation Algorithm

The adaptation algorithm is a very important factor for the efficiency of the compression logic. The adaptation concept of the compression strategies is quite simple. However an

efficient and generic implementation requires a lot of evaluation and fine tuning. The core adaptation rules are: Increasing the algorithm's compression level if the compression speed is too fast and decreasing the compression level if the compression speed is too slow. To make this algorithm generic and efficient various additional conditions have to be incorporated.

The adaptation process takes place between every compression block. In theory every adaptation iteration the compression speed is too slow, the application runtime is increased, and every iteration the compression speed is too fast, the total compression savings are decreased.

Level Jumps

The initial configuration, drastic changes in the data structure or sudden CPU load changes can cause the compression configuration to be far off the ideal configuration for the present conditions. Implying the compression speed is significantly faster or slower than the target value. In that case the compression level should not just be incremented or decremented over multiple adaptation iterations until the compression speed finally reaches the target value. Ideal would be to directly determine the right compression level adaption based on the difference between actual value and target value. Due to unpredictable conditions that is generally not possible, but every measure that speeds up reaching the target value will increase the efficiency.

A simple and viable approach is to approximate the compression level change by determining a conservative divisor for the difference between actual and target value. Listing 4.2 shows exemplary code for this. The divisor has to be determined by evaluation. Because this is just intended to speed up the reaching of the target value and prevent small iterative changes it does not need to be very accurate.

```
1 difference = actual_speed - target_speed;
2 level += (int) difference / divisor;
```

Listing 4.2: Level change approximation

Algorithm Interaction

Compression algorithms usually have very different properties. The focus can be on the compression speed, the compression factor or a trade off in between. Having many subsequent adaptation iterations that can have very different conditions like variable background load, extremely compressible or incompressible data and consequently very different effective network speeds, requires more than a single compression algorithm to be able to efficiently adapt to every possible situation. The adaptation algorithm has to switch between compression algorithms whenever the constraints of the current compression algorithm are reached or an overlapping more efficient algorithm begins.

Decision Thresholds

The adaptation algorithm can not provide an exact and steady compression speed. The compression algorithms levels are too coarse grained to adapt the compression speed exactly. Furthermore the compression speed is always affected by unforeseeable data structure changes. With the naive adaptation algorithm this leads to oscillation around the target value, implying the average actual value will be lower than the target value, as higher values can not compensate time losses. If the compression speed is slower than required for the compression objectives, being maximizing the speedup, maintaining the write call duration or prolonging it by a defined amount, they can not be achieved reliably. Therefore it is far more important to never undercut the targeted compression speed than to exceed it. Exceeding the compression speed implies slight compression savings losses, that have to be accepted for a reliable and predictable behavior. Basically meeting the time constraints is valued more than small additional compression savings.

To ensure not to undercut the targeted compression speed the decisions to lower the compression speed by incrementing the compression level has to be done very conservatively whereas the compression level have to be decremented as soon as the compression speed is too slow.

5. Evaluation

5.1. Compression Algorithm Evaluation

In the course of this work the compression algorithm were evaluated extensively for their suitability in compressing write calls. The behavior in multi-threaded compression as well as the achievable compression factors need to be known to properly utilize them in the adaptation logic.

All measurement values presented in this Chapter are the mean values out of 10 measurement runs. The compression block size used in all measurements was 1 MiB.

5.1.1. Test Files

To evaluate the compression algorithms 5 test files having very different data structures were used.

silesia.tar The `silesia.tar` file contains the Silesia compression corpus, which is a data set specifically designed to test lossless compression algorithms[22]. The corpus consists of typical modern data types from various sources. Among other things it is composed out of text documents, medical images, XML files, source code, applications and databases from different domains. The file size is 203 MiB.

linux.tar The `linux.tar` file contains the Linux kernel version 4.9.5. The file size is 664 MiB.

matrix.bin The `matrix.bin` file contains a binary matrix of double values. It is created by the partdiff application (see Section 5.3). The file size is 490 MiB.

file.nc The `file.nc` file is a small part of the output of the ecology system model ECOHAM5. The original file is a NetCDF file. The file is extremely well compressible, because it contains a lot of redundant data values. For the evaluation purpose only a small section of the beginning of the file is used. The file size is 1000 MiB. Compressing the whole 100GB+ file with Zstd provides compression factors from 5.84 (82.9%) to 6.05 (83.5%). XZ achieves factors of 6.04 (83.4%) to 6.32 (84.2%).

movie.mp4 The `movie.mp4` file contains the rendered and compressed open source movie "Big Bunny Buck"[23]. This file was used to evaluate the behavior of the compression algorithms if confronted with an incompressible file. Movie files are typically

compressed by very sophisticated lossy compression algorithms to be able distribute them efficiently thus contain no redundant information. The file size is 794 MiB.

5.1.2. Evaluation Machine

The compression algorithms and logic evaluation were performed on a Non-Uniform Memory Access (NUMA) dual CPU machine. The machine has two Intel Xeon X5650 CPUs each having 6 physical cores running at 2.67GHz. Hyper-threading is enabled totaling to 24 CPU cores. The machine has 12 GB main memory.

5.1.3. Compression Speeds and Factors

Figure 5.1 shows the compression speeds and factors for different files compressed with LZ4, Zstd and XZ in a single thread. The left y-axis indicates the compression speed in a logarithmic scale and the right y-axis the compression factor. Obviously different data

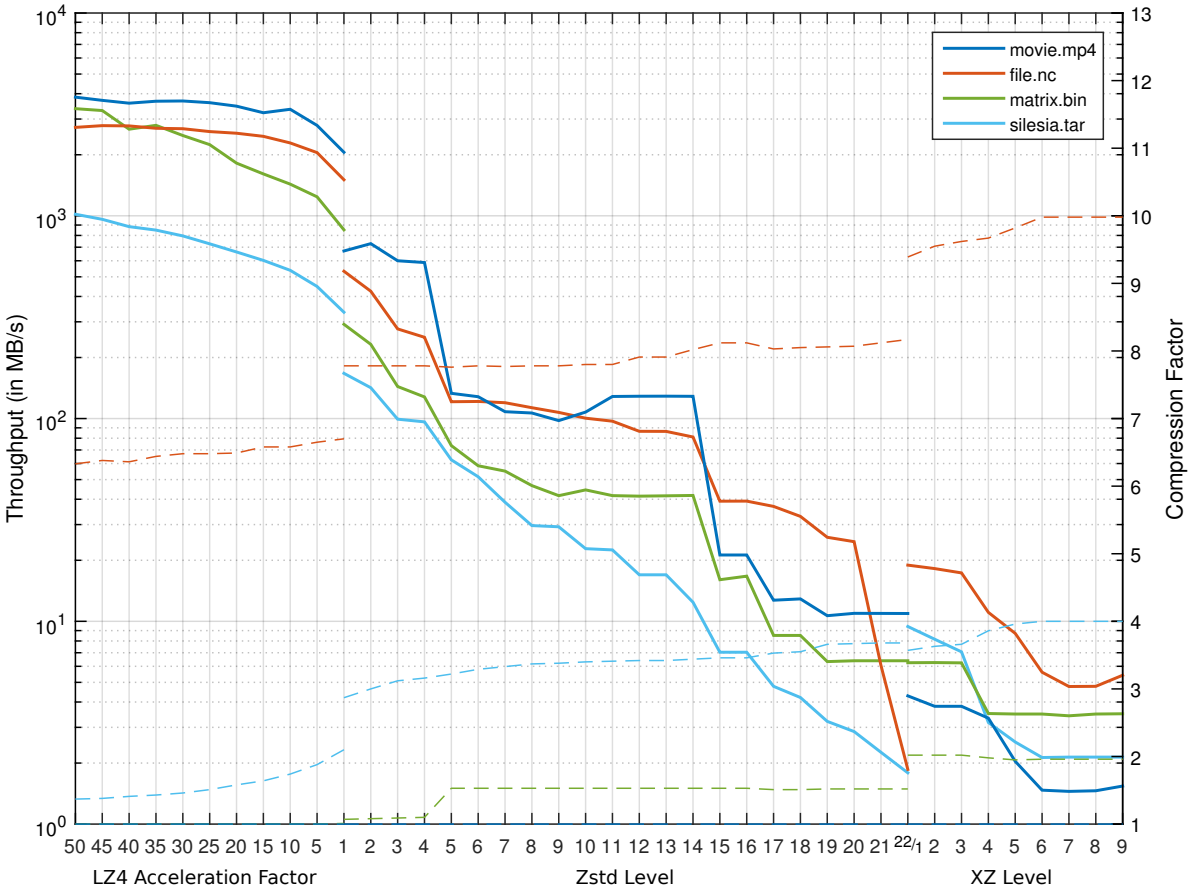

Figure 5.1.: Compression throughput and factors of different file types compressed with LZ4, Zstd and XZ in a single thread.

structures influence the achievable compression speeds and compression factors of each

compression algorithm significantly. Very redundant files like `file.nc` achieve very high compression factors and compression speeds in every compression algorithm. Interestingly absolutely incompressible files like `movie.mp4` achieve even higher compression speeds in some algorithms. This might be due to an early abort mechanism implemented in LZ4 and Zstd. XZ does not seem to have such a mechanism, as `movie.mp4` is the slowest file compressed with XZ. The file `matrix.bin` has a unique behavior across the compression algorithms. LZ4 is not able to compress `matrix.bin`, the file is even enlarged at higher acceleration factors. Zstd achieves compression factors of about 1.1 for the first 4 levels and starting at level 5 the compression factor increases to about 1.53. XZ achieves a factor of about 2 for all levels. The data structure of `silesia.tar` is decently compressible and seems to be more complex for the algorithms to process, as it is clearly the slowest file when compressed with LZ4 or Zstd.

The logarithmic representation of the compression speeds reveals that the algorithms perform in different speed ranges. The slowest compression speed of LZ4 achieves `silesia.tar` at acceleration factor 1 with 335 MB/s increasing up to 1018 MB/s at acceleration factor 50. In Zstd the compression speed of `silesia.tar` ranges from 167 MB/s at level 1 to 1.8 MB/s at level 22 and XZ starts with 9.4 MB/s at level 1 and ends at 2.2 MB/s. The compression speed gap between LZ4 and Zstd does not allow to cover the whole compression speed range, but in general the graduation is acceptable for a smooth adaptation of the compression speed. Oscillation between LZ4 at acceleration factor 1 and Zstd level 1 allows for another step at around 223 MB/s.

5.1.4. Speedup

Performing compression on multiple threads does not speed up the compression speed linearly, although the threads operate on independent data and do not share a sequential section. Especially when increasing the thread count over the number of physical cores. Hyper-Threading (HT) does not nearly provide the same speedup as a physical core. Interestingly compression algorithms can have additional unique speedup characteristics depending on the compression level.

LZ4

Figure 5.2 shows the speedup of all thread counts and acceleration factors between 1 and 50 of LZ4 when compressing `silesia.tar` on the evaluation machine. Without knowing one can see that the machine has HT activated because the speedup per additional core drops drastically after 12 threads. The speedup of two threads is as expected 2. With 6 threads the speedup has already started to be lower than expected, not quite achieving a speedup of 6. 12 threads are far from achieving an expected speedup of 12, instead achieving a speedup of under 10 and additional 12 threads on top totaling to 24 threads do not even achieve a speedup factor of 14. HT naturally does not achieve the same performance as a dedicated core. The degrading speedup per core up to 12 threads on the other hand must have something to do with memory contention, as the memory is a shared resource across the cores. The evaluation machine has a NUMA architecture

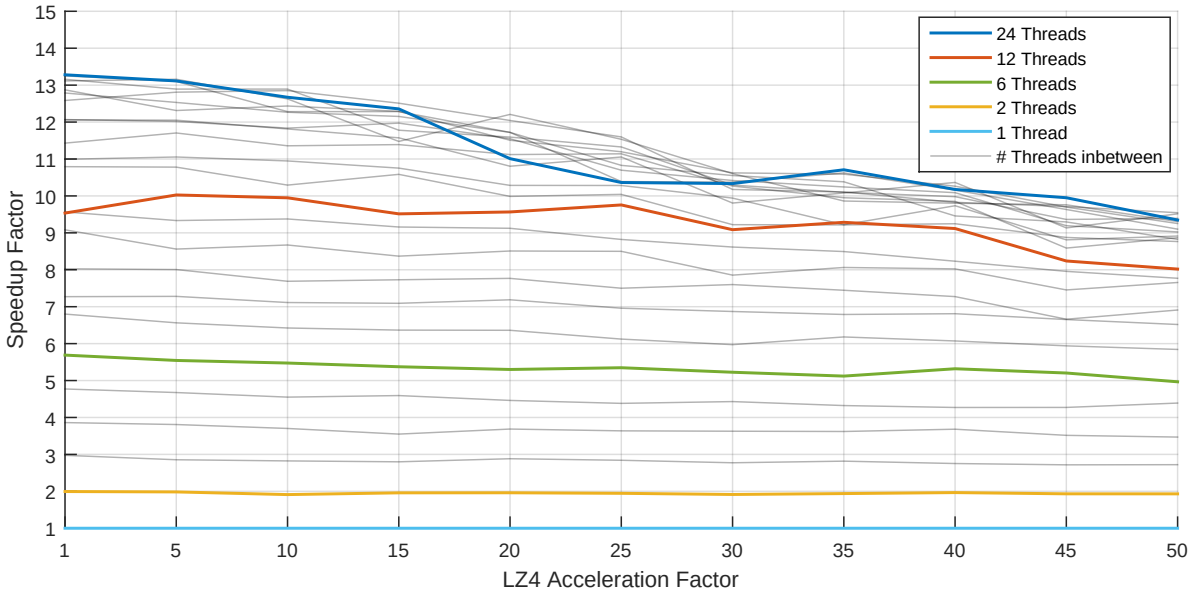

Figure 5.2.: Speedup of LZ4 per thread number and acceleration factor when compressing `silesia.tar`.

what might also influence the speedup if the data is distributed unfavorable across the separated memory banks.

The compression algorithm itself has a quite even speedup characteristic over the acceleration factor range, although the maximal speedup gets worse at higher acceleration factors. This is because at higher acceleration factors LZ4 achieves such high compression speeds per thread, that the memory throughput limitation is easily reached without even utilizing all threads.

Zstd

Figure 5.3 shows the speedup of all thread counts and each compression level of Zstd. The speedup characteristic of Zstd at level 1 is very similar to LZ4's at acceleration factor 1. The maximum speedup of 12 threads and 24 threads is likewise about 10 and 13 for the same reasons. But Zstd introduces another behavior in having unique maximum speedups at certain compression levels. Starting with level 3 up to level 14 the maximum speedup is somehow much lower than the other levels. The compression speed at level 1 is higher than at all other levels, so this should not be an effect of the memory throughput limitation. However because the threads are computing completely independent of each other, except for the shared memory bus resource, some other memory effect of the compression algorithm must be responsible for this effect.

XZ

Figure 5.4 shows the speedup of all thread counts and each compression level of XZ. XZ

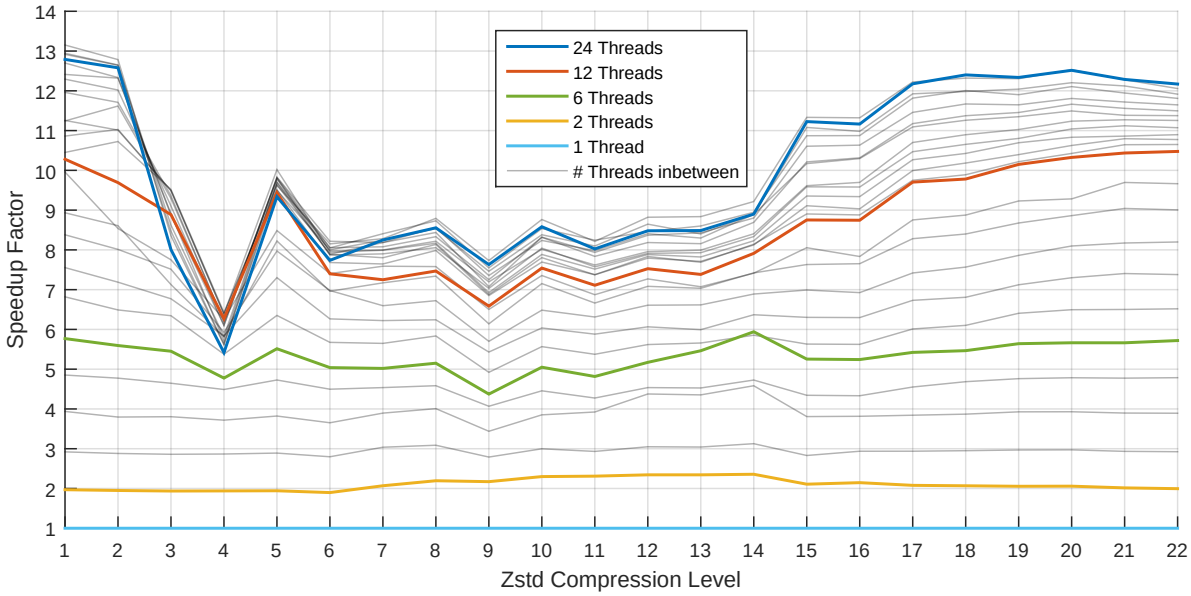

Figure 5.3.: Speedup of Zstd per thread number and compression level when compressing `silesia.tar`.

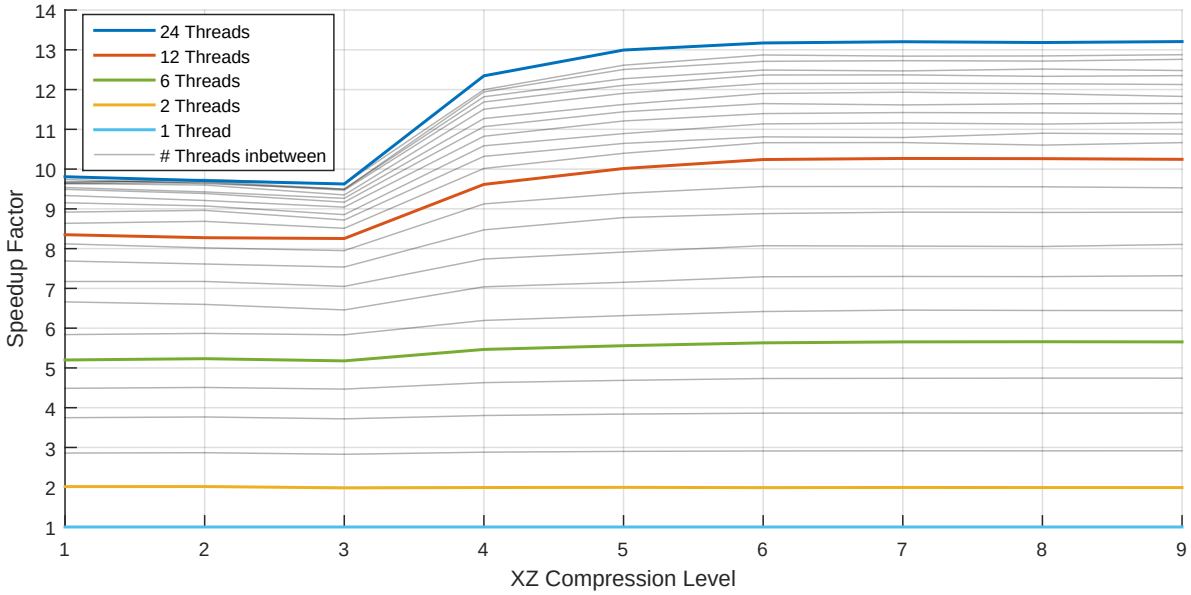

Figure 5.4.: Speedup of XZ per thread number and compression level when compressing `silesia.tar`.

has a lower speedup potential at the beginning up to level 3. Afterwards XZ has again very similar speedup values like LZ4 at acceleration factor 1 and Zstd at level 1.

5.1.5. LZ4

Figure 5.5 shows the compression speeds and factors per acceleration factor of LZ4 when compressing with different thread numbers. The graphs clearly show the impact of the

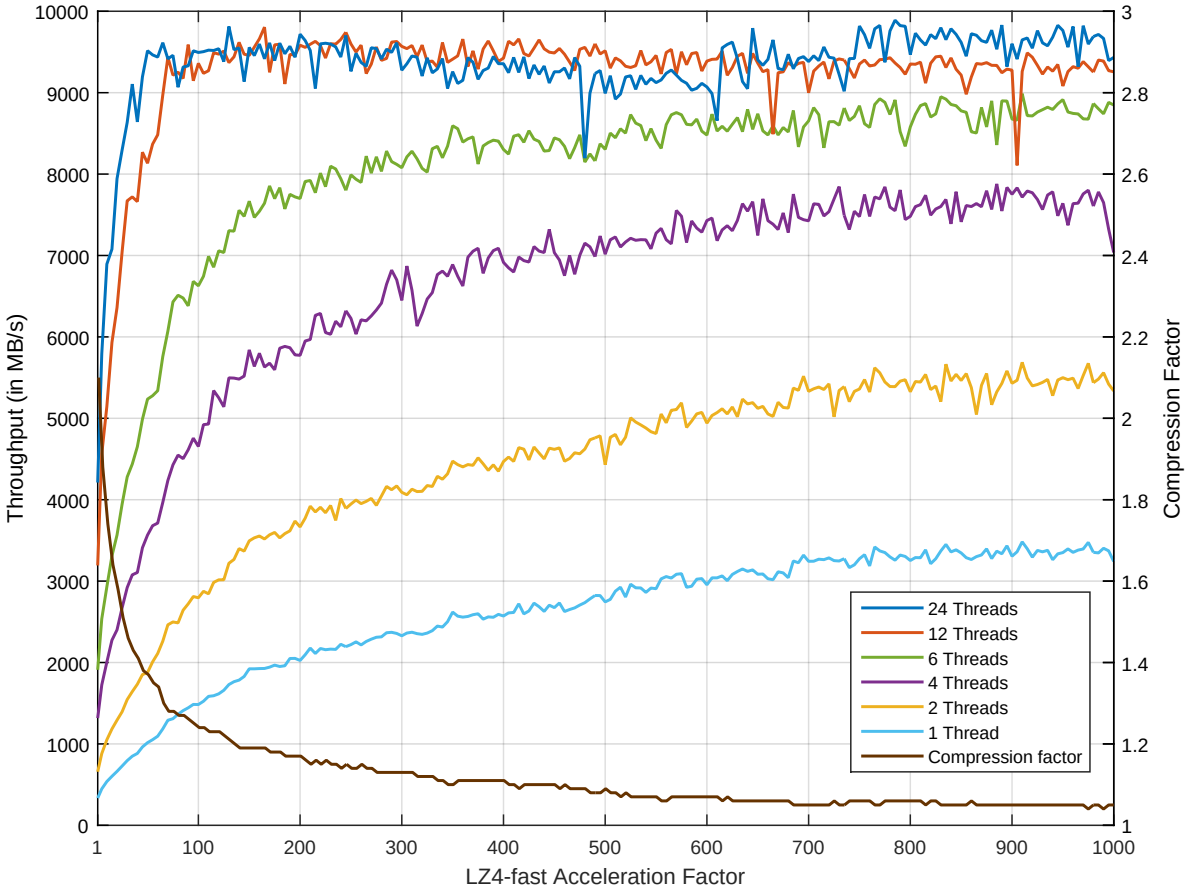

Figure 5.5.: LZ4 compression factor and speed per acceleration factor and different thread numbers for `silesia.tar`.

acceleration factor on the compression speed and compression factor. LZ4 can achieve a throughput of over 3000 MB/s in a single thread. However the compression factor is consequently decreased to about 1.05 (4.8%). So in principle LZ4 can achieve very high compression speeds with only a few threads but the compression savings will be extremely low. The adaptation logic always uses 24 threads and limits the acceleration factor at 50, as the memory limit is already reached at that factor. Otherwise if data is extremely compressible and produces an effective network speed of over 10 GB/s the acceleration factor would be increased in an attempt to further increase the compression speed, although the only effect would be decreased compression savings.

5.1.6. Zstd

Figure 5.6 shows the single thread compression speeds and factors of Zstd, zlib and LZ4HC. Zstd outperforms LZ4HC and zlib by far in compression speed and factor. Therefore LZ4HC and zlib are not used for the adaptation logic.

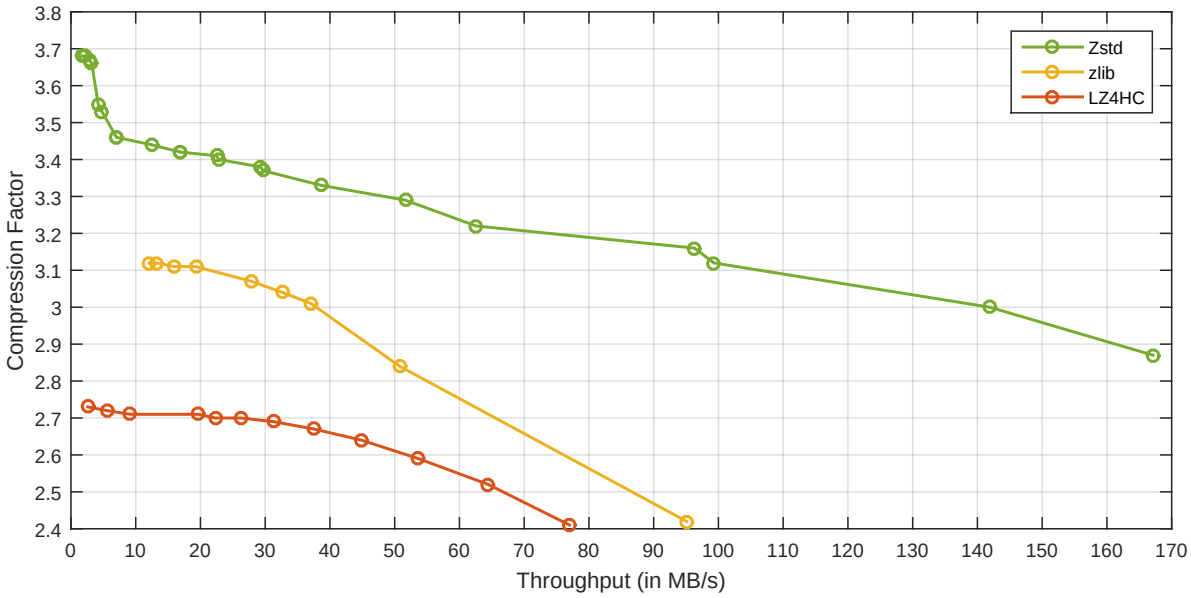

Figure 5.6.: Comparision of compression speeds and factors of Zstd, zlib and LZ4HC compressing `silesia.tar` single threaded.

5.1.7. Decompression

Figure 5.7 shows the decompression speeds of all mentioned compression algorithms. LZ4HC uses the same decompressor as LZ4, therefore LZ4HC decompresses much faster than it can compress. Zstd also outperforms zlib in decompression speed.

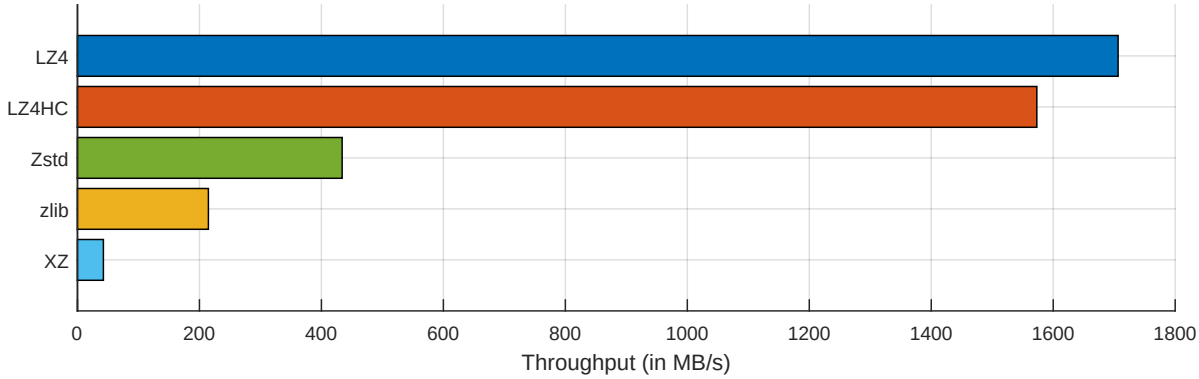

Figure 5.7.: Single thread decompression speeds of `silesia.tar`.

5.2. Compression Strategy Evaluation

So far 3 different strategies for compression speed adaptation have been introduced: Strategy 1 ($S1$) speeding up the write call duration and application runtime if the data is at least somewhat compressible, by continuously adapting the compression speed to the effective network speed. Strategy 2 ($S2$) maintaining the original write call duration and achieving a slightly higher compression factor by adaptation to the physical network speed. And strategy 3 ($S3$) prolonging the write call duration and application runtime by a certain percentage for investing even more time into compression to achieve the highest possible compression factor in a bound and still reasonable time period, by adapting to a smaller percentage of the physical network speed.

In this section the advantages, disadvantages and efficiencies of these strategies are presented. To comprehend the expectable effectiveness of the strategies a closer look at the efficiency of compression algorithms has to be done first.

5.2.1. Compression Algorithm Efficiency

Figure 5.8 shows the achievable compression factor and corresponding compression time for different files using the compression algorithm Zstd. The colored dots represent the 22 compression levels of Zstd applied to different files. The compression time on the left y-axis is expressed relative to the maximum compression time per file.

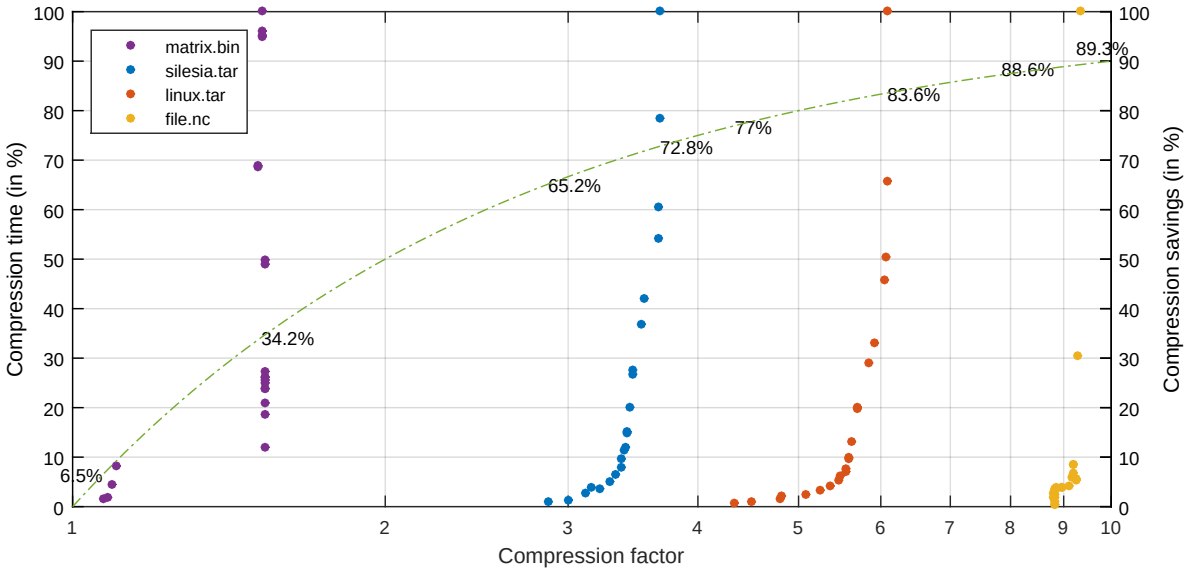

Figure 5.8.: Efficiency of Zstd.

Beginning with the lowest compression level and accordingly fastest compression configuration the compression factor usually starts at a decently high value. The next subsequent higher compression levels grant an increased compression factor for relatively little additional compression time. This behavior changes approaching the last two-thirds compression levels, where the compression factor improvement decreases and

the corresponding compression time increases drastically with every subsequent level. Implying the algorithm efficiency gets very bad after about the first one-third levels.

Furthermore compression factor improvements become decreasingly less relevant the higher the factor starts. A more intuitive measure to compare compression improvements are the compression savings (see Section 2.2). The green line shows the compression savings corresponding to the compression factor and the annotations state the compression savings for the lowest and highest compression factor per file. Their difference indicates the compression savings improvement over the whole algorithm's compression range. `silesia.tar` for example starts at a factor of 2.87 what corresponds to savings of 65.2%. The highest achievable factor with Zstd is 3.68 corresponding to 72.8% savings. So an improvement of only 7.6 percentage points in the compression savings is possible. However the time needed and costs for that are 100x higher than for the lowest savings of already 65.2% at the first compression level. This ratio gets even worse for files having a higher starting compression factor as the compression factor improvements are less relevant. `linux.tar` achieves a 6.6 percentage points improvement for 127x the costs and `file.nc` only 0.7 percentage points for 291x the costs. At lower compression factors `matrix.bin` gets at least 27.7 additional percentage points by increasing the initial factor of 1.07 (6.5%) to 1.53 (34.2%) for 60x the costs. Only 7x the costs by somehow knowing to stop after compression level 5.

The observed characteristics do not only apply to Zstd but to all compression algorithms having multiple compression levels. Overall the time-to-savings efficiency by increasing compression levels past the first one-third levels is very bad especially for initial compression factors past 2. This plays an essential role for the analysis of the different write call compression strategies.

5.2.2. Strategy Effectiveness

Figure 5.9 shows the compression costs, savings and days until the costs redeem itself related to the compression factor of all 3 different write call compression strategies.

The solid lines belong to $S1$, the dashed lines to $S2$ and the dotted to $S3$. The red colored lines represent the unique compression costs and the blue ones the daily storage cost savings per strategy related to the compression factor. The yellow lines indicate the days required until the invested compression costs redeem it-selfs through the daily storage savings. The green line represents additional unique savings of $S1$ detailed below. $S2$ is represented aggregated onto $S1$ and likewise $S3$ is aggregated onto $S2$. As relation between compression and storage costs the values presented in Section 3.4 are used as factor.

Strategy 1 ($S1$)

The compression costs of $S1$ (solid red line) drop related to the compression factor. This is because a higher compression factor implies a higher effective network speed thus a shorter write call duration and consequently less compression time as the compression speed is adapted to the effective network speed. The compression factor is of course

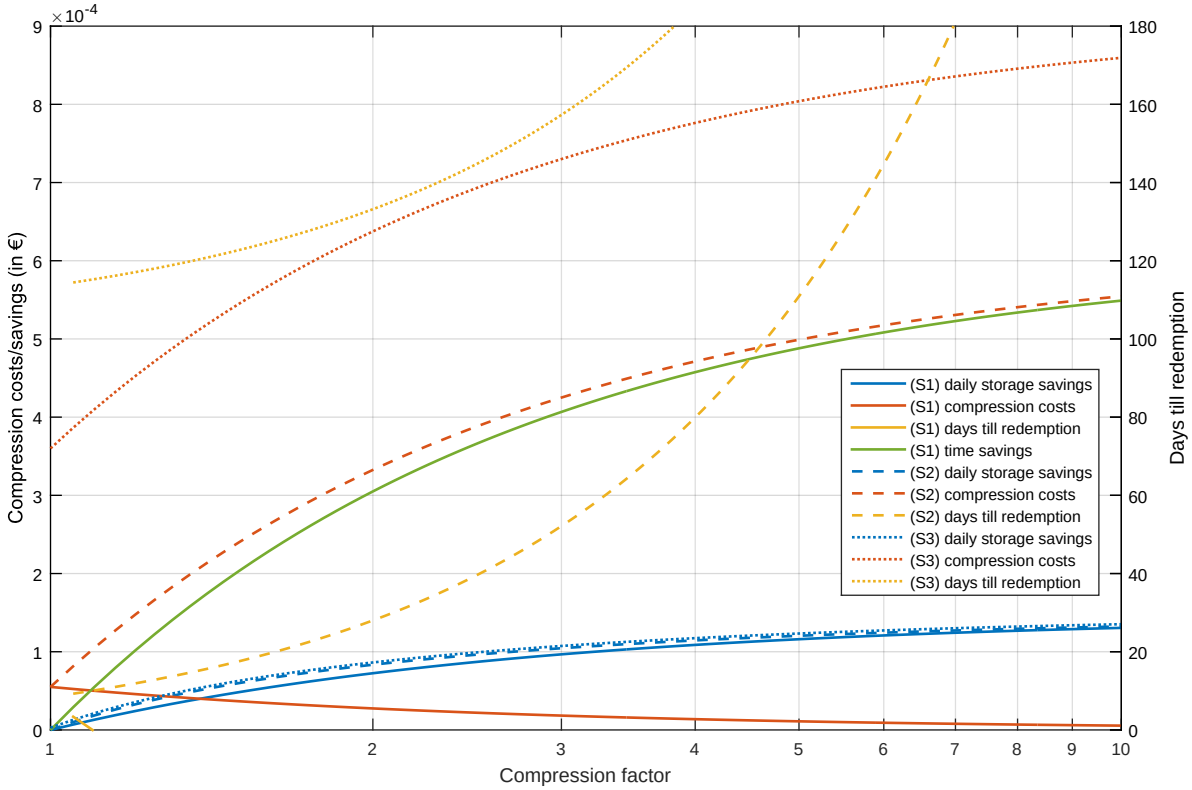

Figure 5.9.: Comparison of the 3 different write call compression strategies.

heavily dependent on the data's compressibility. The savings of $S1$ (solid blue and green line) rise with an increasing compression factor. For one because a higher factor implies less bytes to store, increasing the daily storage savings (blue line). The diminishing returns of higher compression factors become very apparent here though. Then again because of the effect of the effective network speed shortening the write call duration and saving machine time. This is represented by the green line related to the compression factor. $S1$ is the only strategy having this unique instantly present savings as the other strategies do not speed up the write call duration. Almost right away both the unique and daily savings of $S1$ are higher than its compression costs. The days till redemption for $S1$ (solid yellow line) start at 3 days for a factor of 1.05 and drop under 0 thereafter meaning the savings are present immediately for higher factors. This is of course mainly due to the saved machine time and low compression costs as only energy above idle consumption is considered (see Section 3.4).

Strategy 2 ($S2$)

The daily savings of $S2$ are represented by the dashed blue line. The costs and daily savings of $S1$ are always incorporated in $S2$ as $S2$ has to be seen as an extension of $S1$

with the intention to further increase the compression factor by dismissing the effective network speed and performing compression over the whole write call duration. Invested compression time by $S2$ is therefore seen on top of $S1$'s compression time to be able to evaluate the advantages of $S2$ over $S1$. The benefit of $S2$ over $S1$ is hence expressed in the gap between the solid and dashed blue line. Up to a compression factor of 2 the gain rises but soon after diminishes. At a compression factor of 1 $S1$ and $S2$ behave identical as $S1$ did not have a compression factor to generate a higher effective network speed to adapt to. At lower factors up to 1.2 the additional time invested in compression and consequently additional savings by $S2$ are marginal as the speedup of $S1$ is low, implying the additional time invested by $S2$ is short especially in relation to the time already invested by $S1$.

Looking at `matrix.bin` in Figure 5.8 an improvement of about 26 percentage points from compression level 4 to 5 is possible. Assuming $S1$ gets to compression level 4. At level 4 the compression factor of `matrix.bin` is at 1.1 implying $S1$ is saving 9% of the write calls duration. By instead using $S2$ this 9% write call duration speedup is dismissed and used for additional compression time. $S2$'s 9% compression time in relation to $S1$'s 91% is less than 10% additional compression time. Now to get to a compression factor of 1.53 at level 5 from 1.1 at level 4 `matrix.bin` requires 47% more compression time so $S2$ would not be able to achieve this with only 10% more compression time. Generally 91% compression time brought the factor to 1.1 so 100% of the write calls time will not get the factor significantly further. The additional savings plateau of $S2$ is at a factor of about 1.6 where it has about one-third of the write calls time and can improve the compression savings the most, as the factor is reasonably low to still allow a somewhat efficient increase by investing the available one-third more time. At higher factors passing 2.5 even small compression savings improvements are extremely time intensive as the factor has to increase much more.

Another example from Figure 5.8 is the file `silesia.tar` starting at a factor of 2.87 equaling compression savings of 65.2% at compression level 1. The maximum possible compression factor over the whole algorithm range is 3.68 limiting the maximum compression savings improvement to 7.6 percentage points and needing 100x the time for that. At a factor of 2.87 $S1$ uses 34.8% of the write calls duration allowing $S2$ to use the remaining 65.2% for additional compression. Implying $S2$ has nearly double the time $S1$ had for compression. Now to get to a compression factor of 3.16 at compression level 4 about 3.9x the time is needed. So if $S1$ would only achieve the minimal factor of 2.87 with almost triple the time (no speedup) $S2$ would not be able to achieve a factor of 3.16 which would correspond to a compression savings improvement of 3.2 percentage points. If $S1$ achieves higher initial factors the compression savings improvements of $S2$ become even worse as the compression algorithm efficiency gets worse with every level, as described in Section 5.2.1. This naturally applies to all other files starting with a higher compression factor.

The dashed red line represents the compression costs of $S2$. The costs increase with increasing factor as the compression of $S2$ is done in the time $S1$ would have saved through accomplishing a higher effective network speed. Meaning this time could have

been used by the application but is used for additional compression in $S2$. This time has to be considered costlier than the time $S1$ uses as that time could not have been used by the application. $S1$ performs compression while the application would have been blocked anyway thus only consumes energy above idle level. The additional compression of $S2$ in relation to $S1$ uses the whole machine exclusively as the write call could already have returned to the application by doing just $S1$. Now with increasing factor the time saved by $S1$ increases, so do the costs for $S2$. The green line represents the machine time costs saved by $S1$ and equally the costs additionally invested by $S2$. The total costs of $S2$ (dashed red line) are a little higher as the costs of $S1$ (solid red line) have to be included as well[1]. So the gap between the solid and dashed red line represent the additional costs invested in compression by $S2$. $S1$ redeemed its compression costs almost right away (solid yellow line). $S2$ of course takes longer to redeem its costs as no time savings can be accounted but in contrast these exact savings are used for compression thus accounted as additional costs of $S2$. The dashed yellow line indicates the days needed to redeem the additional costs of $S2$.

Every compression investment achieving a somewhat positive factor will redeem itself eventually as the storage savings are daily and the compression costs are unique. As long as the file lifetime is longer than the redemption period of course. The efficiency of $S2$ seems pretty bad but interestingly it redeems its costs within under 90 days for compression factors under 4.5. This is because of the huge gap between CPU time and storage costs. 90 days could be the file lifetime on hard disk storage before it gets transfered to a tape archive. Generally $S2$ is economically superior to $S1$ if the storage time is any longer than the redemption period. But moving the file to a tape archive before the redemption period is over would prolong it considerable maybe even beyond the file lifetime. This is due to the tape archive costs being much lower than corresponding hard disk storage costs, that were considered for the redemption period calculation. So the redemption days only apply for files saved on the hard disk storage. Also if the file gets deleted before the redemption period ended or even right away, the investment of $S2$ is partially or completely lost, whereas the machine time and daily storage savings of $S1$ are present right away if the factor is over 1.1 at least.

Strategy 3 ($S3$)

The dotted blue line represents the daily storage savings of $S3$. The gap between the dotted and dashed line are the additional savings of $S3$ over $S2$. In this example $S3$ always prolongs the write call duration by 50% to perform additional compression. Values of 25% or 100% would be reasonable as well but the write call duration should not be extended much further as the applications runtime will increase considerably.

The red dotted line represents the compression costs of $S3$. They are composed like the costs of $S2$ plus additionally 50% of the write calls duration as machine costs for $S3$'s additional compression. The yellow dotted line indicates the days until the compression investment redeems itself following $S3$.

[1] representing the costs of $S2$ before the machine is used exclusively to achieve further compression.

level	factor	%	time	$S2$ time	$S3$ time
1	2.87	65.2	.097	.278	.146
2	3	66.7	.115	.345	.173
3	3.12	67.9	.266	.830	.399
4	3.16	68.4	.378	1.194	.567
5	3.22	68.9	.357	1.150	.536
6	3.29	69.6	.495	1.629	.743
7	3.33	70.0	.642	2.138	.963
8	3.37	70.3	.786	2.649	1.179
9	3.38	70.4	.934	3.157	1.401
10	3.4	70.6	1.109	3.771	1.664
11	3.41	70.7	1.168	3.983	1.752
12	3.42	70.8	1.455	4.976	2.183
13	3.42	70.8	1.467	5.017	2.201
14	3.44	70.9	1.956	6.729	2.934
15	3.46	71.1	2.613	9.041	3.920
16	3.46	71.1	2.703	9.352	4.055
17	3.53	71.7	3.601	12.712	5.402
18	3.55	71.8	4.112	14.598	6.168
19	3.66	72.7	5.281	19.328	7.922
20	3.67	72.8	5.901	21.657	8.852
21	3.68	72.8	7.653	28.163	11.480
22	3.68	72.8	9.760	35.917	14.640

Table 5.1.: Compression of silesia.tar with Zstd. Time expressed in seconds.

Table 5.1 details the data represented for `silesia.tar` in Figure 5.8. The first 4 columns list the compression level, achieved factor, corresponding compression savings and required compression time of Zstd. The column "$S2$ time" contains the compression time available for $S2$ if $S1$ got to the current compression level. Consequently "$S3$ time" contains the compression time of $S3$ if $S2$ achieved the compression level of the same row.

Now for every compression level that $S1$ might have reached the corresponding compression time $S2$ would have had can be read and compared with the compression time of subsequent compression levels. Stopping at the highest compression level where that time is at least as long as the compression time indicates the compression level $S2$ would have achieved. By subtracting the compression savings of the initial level of the reached level the percentage point gain of $S2$ can be determined. For example assuming $S1$ got to a factor of 3.12 corresponding to compression level 3 and took .266 seconds $S2$ would have had .830 seconds compression time and could get up to level 8. From level 3 to level 8 the compression savings raise from 67.9 to 70.3 so 2.4 percentage points are gained. If on the other hand $S2$ would have gotten to level 3 using .266 seconds $S3$ would have additional 50% of that time resulting in .399 seconds for compression. With

that $S3$ can improve up to compression level 5 gaining 1 percentage point in compression savings over $S2$. Overall with 50% the additional compression time the savings of $S3$ across all factors are only about 2 percentage points.

Conclusion

$S2$ and $S3$ are extremely inefficient due to the characteristics of compression algorithms being pretty inefficient and especially because of the factor improvements having extreme diminishing returns. Usually both reasons apply right away after the first compression factor is reached. At low factors $S2$ has no additional time at all as no significant speedup can be generated with S1. At high factors the compression savings improvements are very small and the additional compression time extremely long and costly. So the savings improvement of $S2$ over $S1$ is pretty small even in the best case scenario with a factor of about 1.6, where the savings max at about 8%. $S3$ performs even worse, because the initial factor is always higher than for $S2$. The problems of $S3$ are basically the same as in $S2$. Nonetheless if the file lifetime on the storage is longer than the days until the costs of $S2$ or $S3$ redeem itself economically they are better. $S3$ probably will not be able to achieve that as the storage file lifetime would have to be longer than one-third of a year. Also the high potential of an application runtime speedup by following $S1$ might be more valuable than an at best 8 percentage point compression savings improvement by following $S2$ or $S3$. This would be of course also be influenced by the typical compression factor achievable in a certain domain.

5.3. Dynamic Compression Adaptation Evaluation

In this section the dynamic adaptation logic is evaluated. To evaluate the efficiency of the dynamic adaptation logic it is compared to all possible static compression configurations of compression algorithms suited for the used network speed.

Partdiff Application

To evaluate the adaptation logic it was tested with a modified version of the partdiff application. The partdiff application is a program for solving partial differential equations, which has initially been written at the Technical University of Munich and has since been extended by the Scientific Computing group at Universität Hamburg. The application has a computation and an I/O phase that can be adjusted in multiple ways by parameters. Therefore it was perfectly suited as exemplary HPC application for the evaluation of the dynamic adaptation logic. For the evaluation purpose it was modified to write the data of specified files, instead of the calculated matrix, to be able to evaluate different file structures. The partdiff application also simulates the background load, comparable to HPC applications, for the asynchronous write calls.

Procedure

For the evaluation of the dynamic adaptation logic it was tested on different common network speeds with multiple files having different file structures. The evaluation was conducted on the same machine and with the same files represented in Section 5.1. To simulate Lustre's RPC calls to the OSSs with different network speeds, a virtual network was used for the evaluation. The virtual network was realized by a timer waiting the exact amount of time that a particular sized chunk of data would need to be transfered over the network having a specified speed. As network speeds 117MB/s, 1200MB/s and 6000MB/s were used to represent Gigabit Ethernet, 10 Gigabit Ethernet and InfiniBand FDR with 4 links. Unfortunately achieving short exact timings without real time scheduling was rather difficult, causing the network speed to deviate slightly on faster network speeds.

All tests were done with 24 compression threads and 48 MB compression cache as the machine has 24 cores. For the asynchronous write calls partdiff was also run with 24 threads. Simulating a HPC application that uses all of the systems cores. The files were modified for the evaluation to have a similar length and to produce longer write calls. Therefore `silesia.tar` was five times concatenated and `matrix.bin` two times to get a length of about 1 GiB like `file.nc`.

5.3.1. Gigabit Ethernet Network

silesia.tar

Figure 5.10 shows the results of writing the file `silesia.tar` 5 times concatenated in a single synchronous or asynchronous write call. The x-axis specifies the LZ4 acceleration

factor from 50 to 1 and the Zstd compression level from 1 to 22. The y-axis denotes the write call speed measured from the application view. The asynchronous write call has 5 asynchronous application computation iterations before the application stops and waits for the write call to return. That equates to about 20% asynchronism if no compression were done. The red lines represent the synchronous write call and the blue lines the asynchronous one. The solid lines indicate the write call speed having a static compression configuration. On the left side with LZ4 and on the right side with Zstd. The dashed lines indicate the same write call being dynamically compressed following $S1$, the dashed and dotted lines following $S2$ and the dotted lines following $S3$. The yellow line represents the network speed and the green lines the effective network speed. The achieved compression factors and savings can be inspected in Table A.1.

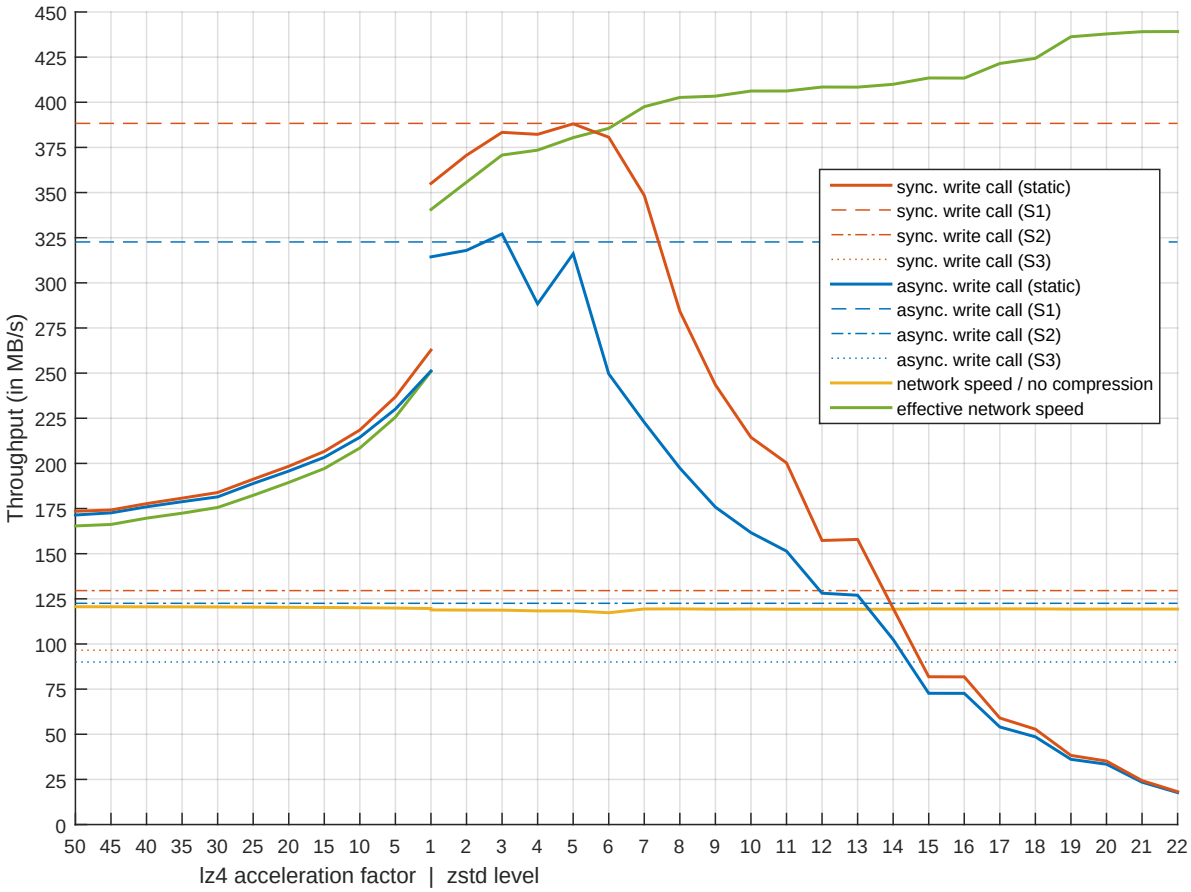

Figure 5.10.: Static compression, dynamic compression following $S1$-3 and no compression of a synchronous and asynchronous write call writing the file `silesia.tar` 5 times concatenated over Gigabit Ethernet.

Compression with LZ4 is not very efficient at slow network speeds like Gigabit Ethernet provides. The compression speed of LZ4 is of course fast enough, but the compression factor can not keep up with the factor provided by Zstd. The highest compression factor LZ4 can achieve is 2.1 (52.3%) at acceleration factor 1. The effective network speed

resulting from that factor limits the maximum write call speed. The write call speeds are a little faster than the effective network speed because of the cache effect the compression buffer memory has (described in Section 4.1). Increasing the acceleration factor of LZ4 decreases the compression factor to increase the compression speed. The compression speed of LZ4 is at all times way faster than required in this setting, so the only effect raising the acceleration factor has, is to decrease the compression factor and consequently the effective network speed and write call speed.

Compression with Zstd can achieve very high compression factors, thus very high effective network speeds as well. Furthermore the compression speeds at lower levels are fast enough to fully utilize the effective network speed, at least in the synchronous case. In the asynchronous case the compression speed still almost reaches the effective network speed. This results in very sped up write calls having a respectable compression factor too. The compression factor starts at 2.9 (65.1%) at level 1, quickly increases to 3.2 (68.9%) at level 5 and ends at 3.7 (72.8%) at level 22. Synchronous and asynchronous write call speeds differ visibly with Zstd. In the asynchronous case less CPU time is available for compression because the asynchronous application computation takes up half of it during the asynchronous portion of the write call. The speed difference between synchronous and asynchronous write call gets smaller every level starting at level 5. This is because the write call duration from there on only increases and the relation between the asynchronous portion and the write call duration gets smaller every level. The compression speed of LZ4 is so high that the asynchronous computation does not slow it below the limiting effective network speed, hence the effect is not visible in the write call speed. The compression speed of Zstd in the synchronous write call is limited by the effective network speed before level 6 as well. The write call speed can only be higher than the effective network speed because of the compression cache effect. At level 4 the Zstd compression algorithm seems to be very susceptible to CPU contention. The negative spike is not caused due to the asynchronous portion taking longer than the write call.

The dynamic adaptation algorithm following compression strategy $S1$ aims to maximize the write call speed. The starting compression configuration of the adaptation algorithm was Zstd at level 1. Having Gigabit Ethernet as network technology Zstd level 1 is a reasonable starting configuration as the compression speed will always be adequate while providing a higher compression factor than LZ4 with an acceleration factor of 1. For the synchronous write call the adaptation logic achieved a slightly higher speed and compression factor than the most suited static compression configuration did. This is possible through advantageous dynamic compression configuration adaptations to the file structure. The file `silesia.tar` being composed of various common data types is predestined for that. For the asynchronous write call the adaptation algorithm was only slower than one specific static configuration being level 3. Compression strategy $S1$ achieved a compression factor of 3.22 (68.9%) for the synchronous and 3.09 (67.6%) for the asynchronous write call.

The objective of compression strategy $S2$ is to maximize the compression factor while maintaining the original write call speed, as if no compression were done. The adaptive algorithm, following strategy $S2$, achieved to generally maintain the original write call

speed, although the synchronous write call is slightly sped up. For the synchronous write call a compression factor of 3.38 (70.4%) and for the asynchronous write call a factor of 3.36 (70.2%) was reached. That is a compression savings gain of 1.5 percentage points for the synchronous and 2.6 pp for asynchronous write call over $S1$, but at the expense of a over 3 times sped up write call.

Compression strategy $S3$ prolongs the write call by a set percentage to further increase the compression factor. $S3$ was configured to prolong the write call by 50%. Supposing 117 MB/s as network speed and consequently original write call speed, that equates to 78 MB/s. The speeds following $S3$ for both the synchronous and asynchronous write calls are a bit faster because of the compression cache effect. The achieved compression factors are 3.41 (70.6%) for the synchronous and 3.40 (70.6%) for the asynchronous write call. That is only a gain of 0.2 pp and 0.4 pp over $S2$.

Picking LZ4 with acceleration factor 1 (higher would be nonsense for Gigabit Ethernet) for static compression in this instance would be very inefficient, as the write call speed and the achieved compression factor would be way lower than if Zstd with a compression level ranging from 1 to 8 would have been chosen. The most efficient static configuration to pick for speeding up applications having only synchronous write calls with data structures similar to `silesia.tar` would be Zstd with level 5. For mostly asynchronous write calls Zstd level 3 would be best suited. However the dynamic adaptation algorithm following $S1$ would provide a very good result in both cases without the need to define a fixed compression configuration that could be inefficient for other data structures. Also the achieved compression factor is quite similar to the other compression strategies $S2$ and $S3$ Choosing the right static compression configuration to maintain the original write call speed of different files is hardly possible in advance. The adaptation algorithm following $S2$ on the other hand adapts the configuration to meet that objective very reliably. The same applies for $S3$.

matrix.bin

Figure 5.11 shows the results of writing the file `matrix.bin` 2 times concatenated in a single synchronous or asynchronous write call. The asynchronous write call has 10 asynchronous application computation iterations. That equates to about 33% asynchronism if no compression were done. The achieved compression factors and savings can be inspected in Table A.2.

LZ4 is not able to achieve a compression factor significantly higher than 1 for the `matrix.bin` file. Even compression factors below 1 occur on higher acceleration factors. This prevents an increased effective network speed, so no write call speed up is possible. However the write calls are sped up slightly by the compression cache effect.

The first 4 compression levels of Zstd achieve a compression factor of about 1.1 (9.1%). That is not much either, but enough to raise the effective network speed from 121 MB/s to 133 MB/s. Consequently the write calls are sped up more than by using LZ4. Beginning from level 5 Zstd is able to compress the binary matrix more efficiently, achieving a compression factor of about 1.53 (34.7%) for all subsequent levels. This raises the effective network speed to 185 MB/s. From level 5 up to level 14 Zstd is capable to compress

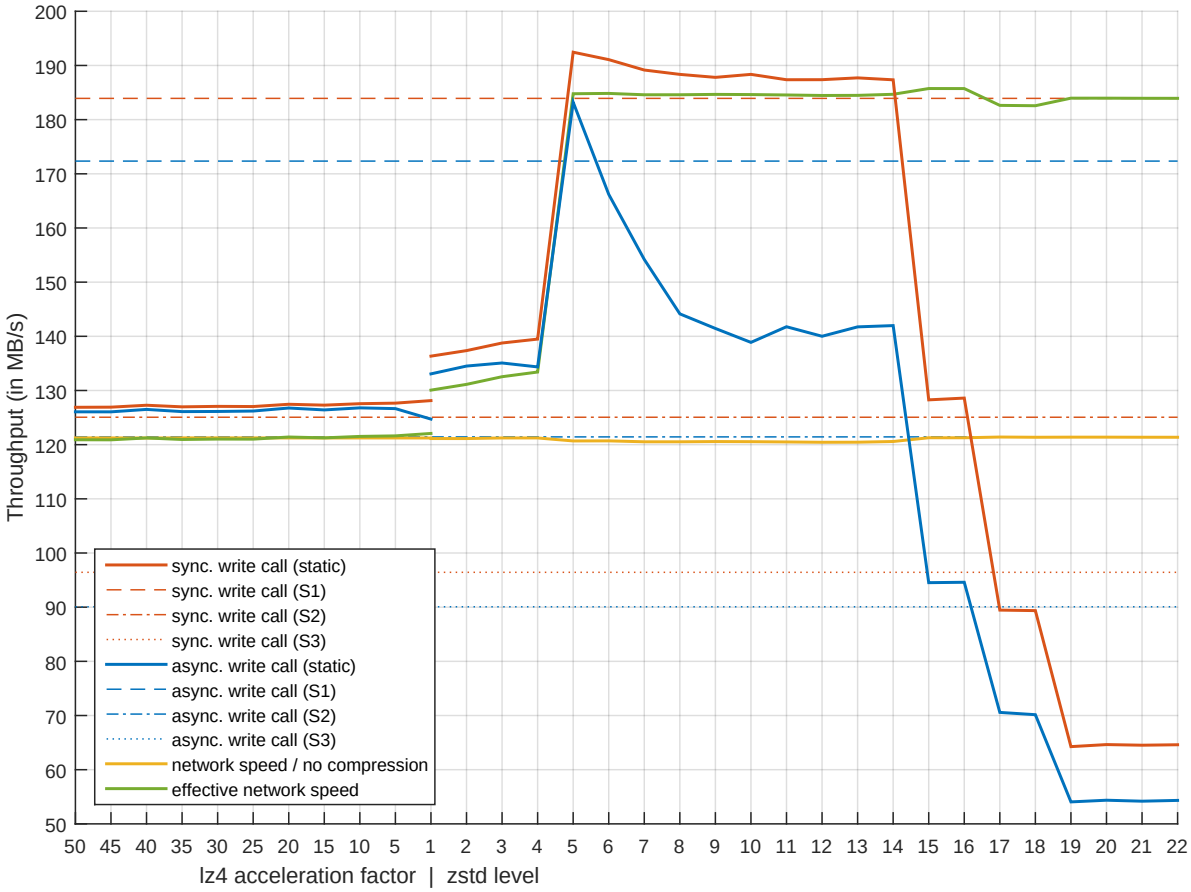

Figure 5.11.: Static compression, dynamic compression following $S1$-3 and no compression of a synchronous and asynchronous write call writing the file `matrix.bin` 2 times concatenated over Gigabit Ethernet.

faster or as fast as the effective network speed in the case of a synchronous write call having no CPU contention. Afterwards the compression speed and consequently write call speed drops drastically. During the asynchronous write call the compression speed of level 5 barely reaches the effective network speed and drops significantly thereafter. Compressing `matrix.bin` is special, because the compression factor does not increase any further after level 5.

The dynamic adaptive compression algorithm following strategy $S1$ achieves a decent synchronous write call speed being only 8 MB/s below level 5. A static configuration ranging from level 5 up to level 14 would achieve slightly better results. The first disadvantage of the adaptive compression algorithm in this case is starting at the compression configuration Zstd level 1, having a much lower compression factor. Secondly because level 5 achieves the highest compression factor and speed possible over the whole compression algorithm range, a static configuration using level 5 will always have a better result than any adaptive algorithm, that even once tries another level to search for a higher compression factor. Furthermore every level after level 5 is inefficient as the same

factor is achieved while consuming more CPU time. However the adaptation algorithm has no way to know that the compression factor will not increase at higher levels. For the asynchronous write call the adaptation algorithm is about 10 MB/s slower than the static level 5 configuration, but level 5 is the only faster static configuration. So in the synchronous case there would be a wider range to achieve a good speedup if at least level 5 was picked, whereas in the asynchronous case exactly level 5 has to be picked to achieve a better result than the adaptation algorithm.

Following the strategies $S2$ and $S3$ is very inefficient for this file, but there is no way of knowing that in advance if no evaluation of all compression speeds was done. $S2$ and $S3$ can not achieve a significantly higher factor than $S1$ but dismiss all the potential speedup by trying.

Considering the special behavior of Zstd with the `matrix.bin` file following $S1$ provides very good results. The only better option would be a static configuration with exactly level 5. However the only way to know that is trough an evaluation of the specific file.

file.nc

Figure 5.12 shows the results of writing the file `file.nc` in a synchronous and asynchronous write call. The asynchronous write call has 3 asynchronous application computation iterations. That equates to about 12% asynchronism if no compression were done. The achieved compression factors and savings can be inspected in Table A.3.

Although the data structure of `file.nc` is extremely well compressible, LZ4 is not quite able to get a compression factor as high as Zstd. LZ4 achieves a compression factor of 6.71 (85.1%) at acceleration factor 1. Even at acceleration factor 50 a compression factor of still 6.33 (84.2%) is reached. LZ4 is always able to compress fast enough here, but is limited by its compression factor.

Zstd starts with a compression factor of 7.78 (87.1%) at level 1. For Gigabit Ethernet that translates into an effective network speed of around 900 MB/s. Up to about level 3 the compression speed can keep up with the effective network speed, afterwards the write call speed is just boosted over the effective network speed by the compression cache effect.

Because of the very redundant data structure, even XZ can provide an adequate compression speed for Gigabit Ethernet, if speedup is not the goal of course. From level 1 to 3 XZ achieves almost exactly the physical network speed and outperforms the last two levels of Zstd in compression speed and factor. At level 6 XZ reaches the maximum compression factor of 8.7 (88.5%) at about 72 MB/s.

The adaptive compression algorithm following strategy $S1$ only achieves 888 MB/s out of possible 933 MB/s in the synchronous case and 646 MB/s out of possible 712 MB/s in the asynchronous case compared to the most suited static configuration. The adaptive algorithm often tries to compress at level 4 as the compression speed of level 3 is higher than the effective network speed, but level 4 is disproportionately slower than level 3 especially in the asynchronous case. This might be avoidable by better tuned thresholds of how much faster the compression speed of a compression level has to be than the effective network speed in order to switch to the next level. This thresholds should not

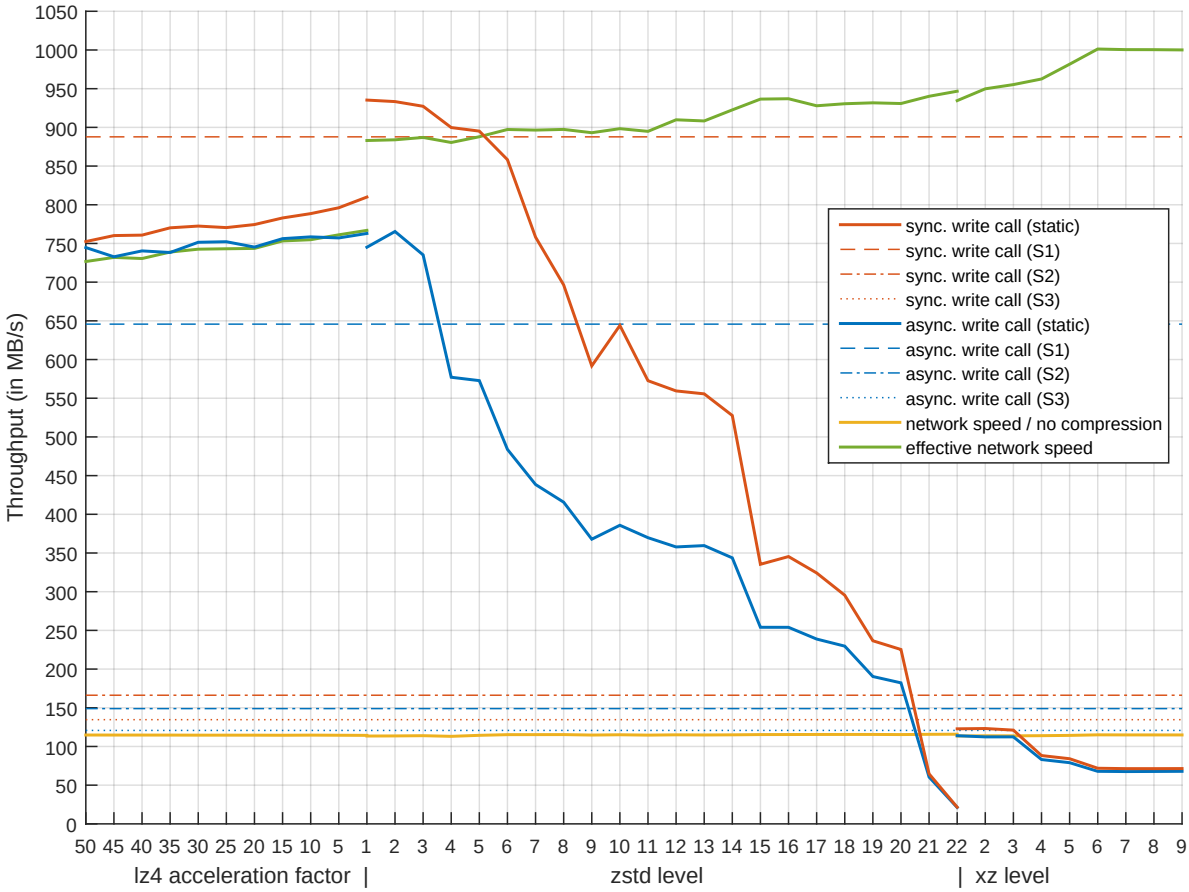

Figure 5.12.: Static compression, dynamic compression following $S1$-3 and no compression of a synchronous and asynchronous write call writing the file `file.nc` over Gigabit Ethernet.

be absolute values but rather be relative to the compression or effective network speed.

The strategies $S2$ and particularly $S3$ are a little too fast considering their purpose. Here the adaptation algorithms level jumps are to conservative. To much adaptation iterations are spent in faster compression levels before finally reaching XZ. This might also be tunable by incorporating the compression or effective network speed into the adaptation process. Basically the problem leading to the inefficiency of the adaptation algorithm with this file is the extreme high compression and effective network speeds as the file's data structure is so effectively compressible. The decision parameters for the adaptation process of Zstd are not well tuned to correctly consider such high speeds.

Considering this file the strategies $S2$ and $S3$ are absolutely inefficient anyways, as the achieved compression savings of neither compression configuration do significantly differ as the compression factor is already extremely high. The highest achievable compression factor is 8.7 (88.5%) with XZ level 9. Strategy $S3$ should reach a similar result with a proper tuned adaptation algorithm. However strategy $S1$ already achieves a factor of 7.69 (87%) in the asynchronous and 7.78 (87.1%) in the synchronous case and that over

9 to 12 times faster than $S3$ would be.

If kept in mind that the adaptation algorithm is meant to run inside the file system, which is used for many applications having different file structures, the results of the compression strategies especially $S1$ are decent.

Conclusion Gigabit Ethernet

Zstd is extremely well suited for write call compression of every possible data structure when having Gigabit Ethernet. Using Zstd at level 5 turned out to be a nearly perfect static configuration to achieve very sped up write calls for various data structures. The adaptation algorithm following strategy $S1$ was slightly behind that. However the adaptation decisions leading to inefficiencies should be improvable to some extent.

5.3.2. 10 Gigabit Ethernet

silesia.tar

Figure 5.10 shows the results of writing the file `silesia.tar` 5 times concatenated in a single synchronous or asynchronous write call. The asynchronous write call has 1 asynchronous application computation iteration, equating to about 37% asynchronism if no compression were done. The achieved compression factors and savings can be inspected in Table A.1.

Like before with Gigabit Ethernet, the highest compression factor for `silesia.tar` using LZ4 is 2.1 (52.3%). The compression speed in the synchronous case should be about 5000 MB/s and would still be fast enough to fully utilize the effective network speed of about 2500 MB/s, but at this speeds the compression and adaptation overhead becomes visible by dragging the synchronous write call speed down to 2426 MB/s. After acceleration factor 5 the effective network speed starts to limit the write call speed again and the write call speed lies on top through the compression cache effect. The overhead can have multiple causes for example a often preempted I/O thread, the internal synchronization of the asynchronous buffer queues or the virtual network thread used to simulate different network speeds for this evaluation. The asynchronous write call is much slower than the synchronous one. The asynchronous application computation extends to 62% of the write call duration because of the CPU contention with the compression, affecting the compression speed negatively as well.

The higher compression factor and consequently higher effective network speed does not enable Zstd to surpass LZ4 anymore, because the compression speed is too low. The effective network speed potential of 10 Gigabit Ethernet is way to high for Zstd in contrast to Gigabit Ethernet. Zstd at level 1 just manages to reach the physical network speed for the asynchronous write call.

The adaptation logic following strategy $S1$ achieves good results for both the synchronous and asynchronous write calls, even though the starting configuration was Zstd at level 1. Following strategy $S2$ ended up with a slightly too fast synchronous and slightly too slow synchronous write call. Also strategy $S3$ was slightly too fast particularly in

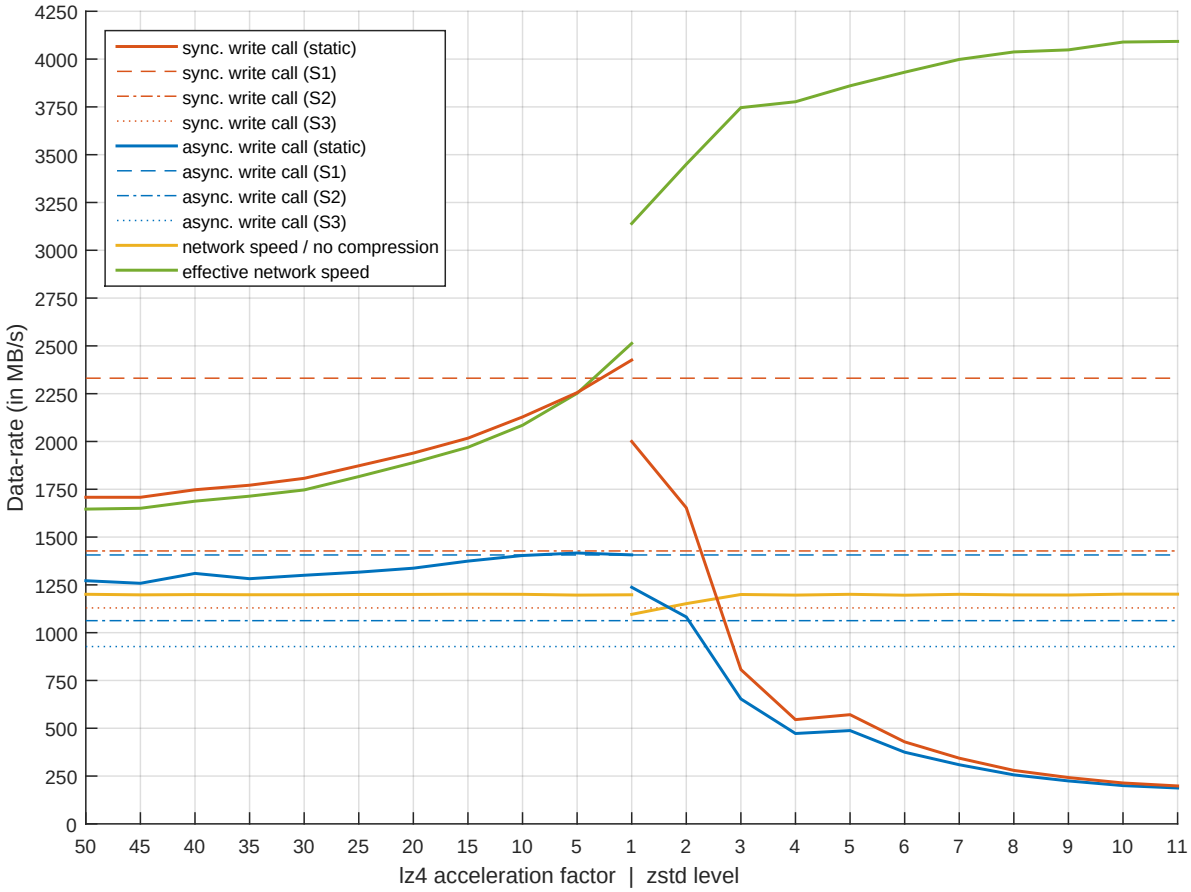

Figure 5.13.: Static compression, dynamic compression and no compression of a synchronous and asynchronous write call writing the file `silesia.tar` 5 times concatenated over 10 Gigabit Ethernet.

the synchronous case. This adaptation inefficiencies should be correctable by better fine tuning the adaptation algorithm. Not for a special data structure but the general case.

matrix.bin

Figure 5.14 shows the results of writing the file `matrix.bin` 2 times concatenated in a single synchronous or asynchronous write call. The asynchronous write call has 1 asynchronous application computation iteration, equating to about 28% asynchronism if no compression were done. The achieved compression factors and savings can be inspected in Table A.2.

LZ4 is not able to compress the binary matrix. In the synchronous case the compression is nonetheless fast enough to cause the compression cache effect. This is the first time that no asynchronous write call reaches the physical network speed. LZ4 provides fast enough compression speeds even in the asynchronous case, but again the compression and adaptation overhead prevents higher speeds. In situations of high CPU contention having 24 compression threads, 24 application computation threads, 1 I/O thread and 1

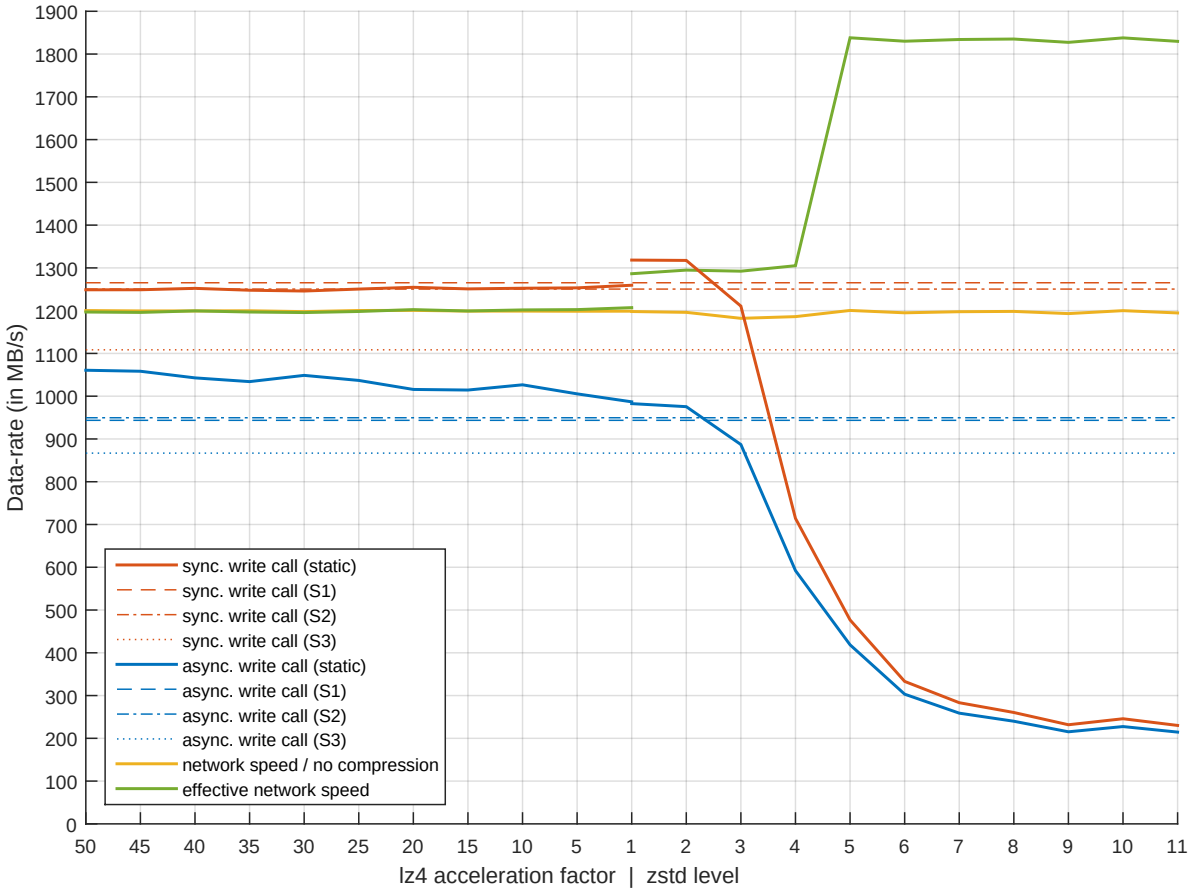

Figure 5.14.: Static compression, dynamic compression and no compression of a synchronous and asynchronous write call writing the file `matrix.bin` 2 times concatenated over 10 Gigabit Ethernet.

virtual network thread the compression and adaptation overhead and lack of real time scheduling for the I/O and virtual network thread become very visible.

Zstd is able to compress faster than the physical network speed for the first 3 levels in the synchronous write call. Afterwards the speed drops quickly. Like with LZ4 the compression speed of Zstd level 1 to 2 should be fast enough for the write call to at least reach the physical network speed, but the overhead prevents it.

The results of the adaptation algorithm following the strategies $S1$ to 3 for the synchronous write call are alright. The binary matrix is hardly compressible, so there is not much scope for achievements anyway. The initial configuration was Zstd level 1. For the asynchronous write call compression seems generally ineffective at least as long as the overhead problems are present. The worst one might be the synchronization of the parallel compression threads to transfer the compressed data in series to the network. So the adaptation algorithm and compression strategies can not really be evaluated in the asynchronous case expect for strategy $S3$. Strategy $S3$ achieves a compression factor of 1.09 (8.5%). The speed should be around 800 MB/s but is slightly faster.

The `matrix.bin` file has a unfavorable data structure for compression, that does not permit efficient compression in the asynchronous case. For the synchronous case strategy $S1$ achieves a compression factor of 1.09 (8.6%) and speeds up the write call slightly. The other strategies do not achieve significantly higher compression factors.

file.nc

Figure 5.15 shows the results of writing `file.nc` in a synchronous and asynchronous write call. The asynchronous write call has 1 asynchronous application computation iteration, equating to about 36% asynchronism if no compression were done. The achieved compression factors and savings can be inspected in Table A.3.

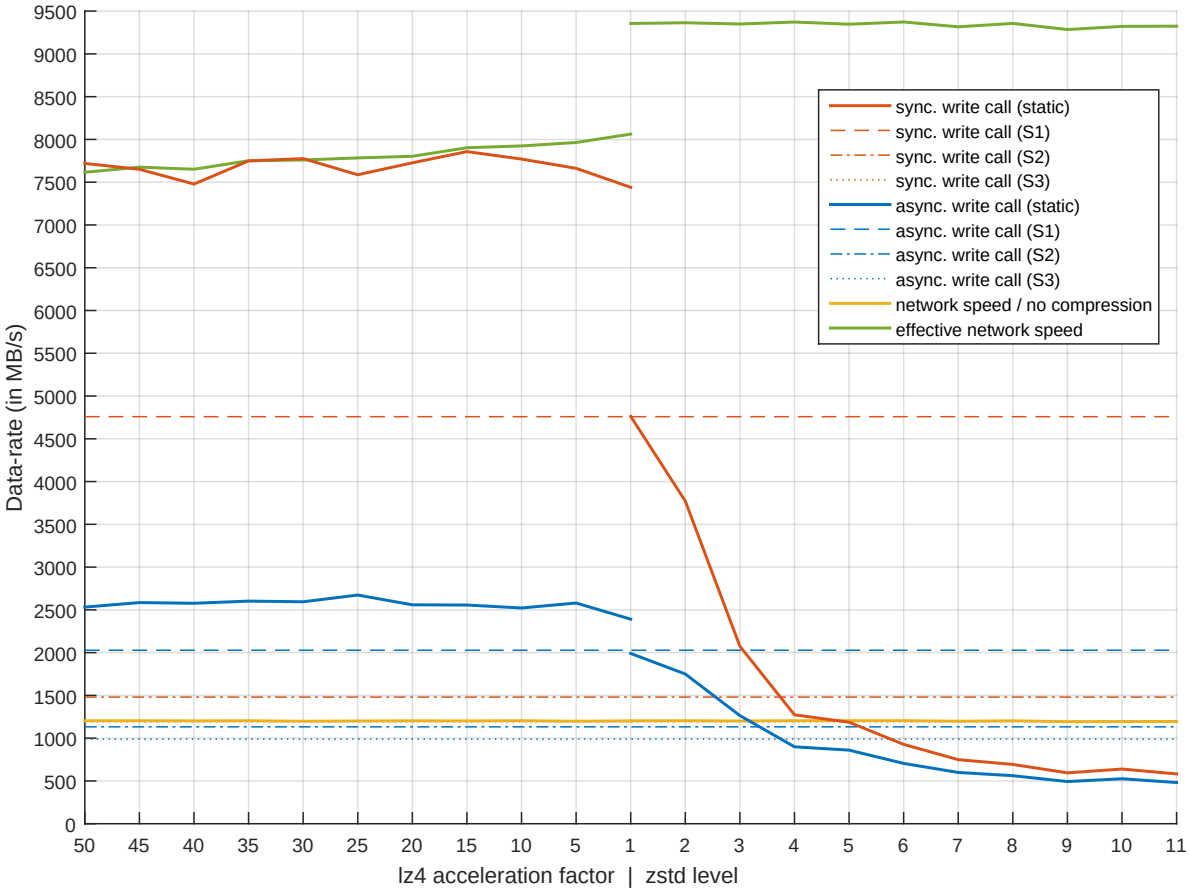

Figure 5.15.: Static compression, dynamic compression and no compression of a synchronous and asynchronous write call writing the file `file.nc` over 10 Gigabit Ethernet.

Because LZ4 is capable of very high compression speeds and this file provides such a compressible data structure, very high write call speeds can be achieved in the synchronous case. At acceleration factor 1 the compression speed is not as fast as the effective network speed, but starting at acceleration factor 15 the effective network speed is reached. The

asynchronous case is about 3 times slower than the synchronous case because of CPU contention and overall overhead, but because of the favorable data structure still double as fast as the physical network speed. Zstd achieves a maximum synchronous write call speed of up to 4800 MB/s at level 1. The asynchronous write call speed still reaches up to 2000 MB/s.

The adaptation logic following strategy $S1$ does not achieve a good result. It looks like exclusively Zstd level 1 was used, but that is not the case. Zstd is only used at the data sections where the compression factor is low. At those sections LZ4 has a way higher compression speed than the effective network speed. This causes an algorithm switch to Zstd which is even fast enough for some subsequent data sections thus increases up to Zstd level 3. However as soon as the redundant data sections occur again Zstd level 3 slows down the compression speed immensely until changed back to LZ4 in subsequent adaptation iterations. In principle that effect is intended, but the algorithm switching decision has to be even more conservative for higher effective network speeds, as the compression configuration speed differences are about 6000 MB/s. Furthermore such high speed differences are only present with extreme redundant data structures, where slower compression is very inefficient anyways. In the synchronous case LZ4 achieved a compression factor of 6.71 (85.1%) whereas $S1$ achieved 7.12 (86%) but was almost 3000 MB/s slower.

Strategy $S2$ managed to be roughly as fast as the physical network speed, but strategy $S3$ is too fast particularly in the synchronous case where it lies directly under the network speed. The problem of $S2$ and $S3$ in contrary to $S1$ is a to conservative level increase.

5.3.3. Infiniband FDR 4x Links

silesia.tar

Figure 5.16 shows the results of writing the file `silesia.tar` 5 times concatenated in a single synchronous write call. The achieved compression factors and savings can be inspected in Table A.1. The starting configuration for the compression strategies was LZ4 at acceleration factor 1.

At such high network speeds even only 1 asynchronous application computation iteration lasts longer than the write call duration. Also the compression speed would always be lower than the physical network speed because of the CPU and memory contention. Therefore no asynchronous write call is evaluated.

The synchronous write call speed increases by increasing the acceleration factor. The corresponding compression speed is higher than the resulting write call speed, but the pipeline overhead slows the write call speed down. The write call speed does not reach the effective network speed. Starting at acceleration factor 25 the write call speed starts to fall corresponding to the effective network speed, although it is not reached yet. The difference between write call speed and effective network speed from that point on can be seen as roughly the pipeline overhead. The overhead is less impacting when the effective or physical network speeds are lower.

Another problem of the adaptation logic is that it does not account for the pipeline

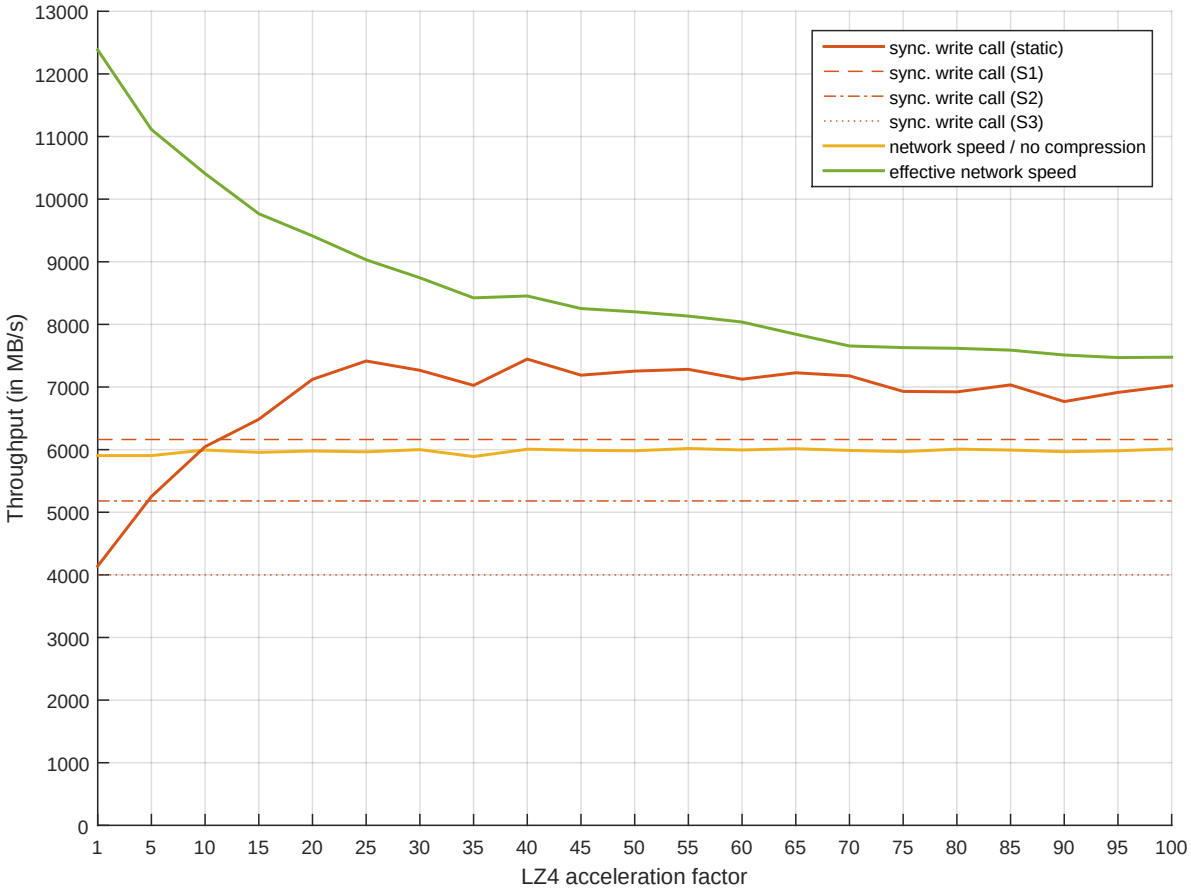

Figure 5.16.: Static compression, dynamic compression and no compression of a synchronous write call writing the file `silesia.tar` 5 times concatenated over Infiniband FDR with 4 links.

overhead. Only the compression speed is observed and adapted to a target value. Although everything in between the compression process, mainly the synchronization through the asynchronous queues to transfer the parallel compressed data in series over the network, will slow down the write call speed, but is not considered. So as the overhead can be seen between the write call speed and effective network speed it can also be seen between the maximum write call speed and the speed of the adaptation algorithm following strategy $S1$. But also the acceleration factor decrease decisions have to be more conservative, as incorrect decisions have a drastic speed impact on this high speed network speeds.

Also the adaptation of $S2$ does not consider the pipeline overhead and just adapts the compression speed to the physical network speed. The pipeline overhead in between the compression iterations is hidden for the decision logic and slows the write call speed under the target value of the physical network speed. The overhead is less here because it was only adapted to the physical network speed instead of the effective network speed.

Strategy $S3$ achieves exactly the speed it should achieve for a 50% prolonged write call.

At this high speeds and corresponding lower compression factors the strategies $S2$ and $S3$ become more reasonable as they achieve higher percentage point gains. $S1$ achieves a factor of 1.6 (37.3%), $S2$ 1.91 (47.6%) and $S3$ 2.17 (53.9%). Overall compression at this network speeds is still beneficial. Not for the speedup but the compression savings.

matrix.bin

Figure 5.17 shows the results of writing the file `matrix.bin` 2 times concatenated in a single synchronous write call. The achieved compression factors and savings can be inspected in Table A.2. The starting configuration for the compression strategies was LZ4 at acceleration factor 1.

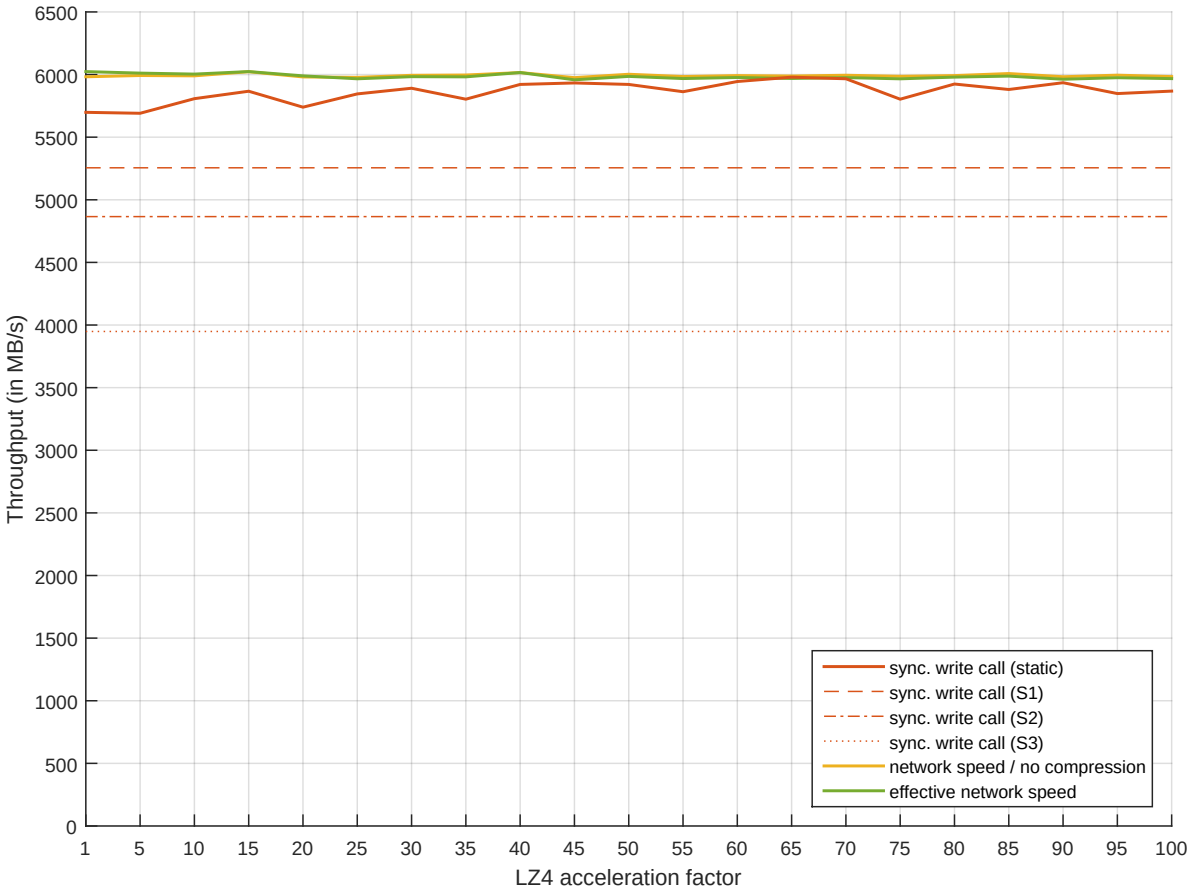

Figure 5.17.: Static compression, dynamic compression and no compression of a synchronous write call writing the file `matrix.bin` 2 times concatenated over Infiniband FDR with 4 links.

For LZ4 the file `matrix.bin` serves as an example for an incompressible file. Without any CPU contention LZ4 compresses the file with 8300 MB/s with 24 threads at acceleration factor 1. The pipeline overhead prevents the write call speed form reaching the physical network speed for all acceleration factors lower than 45. A static compression

configuration would not slow down the write call to much. However this would be a case for the "no compression" file hint, because hardly even positive compression savings are possible.

Strategy $S1$ is slower than LZ4 acceleration factor 1 because it uses Zstd for some data sections. Zstd can compress the whole file with about 4100 MB/s, but little file sections may be faster. However the adaptation algorithm should be adapted to not use Zstd at such high network speeds, because the time lost if the data structure changes and Zstd is still used, before changed to LZ4 in the next adaptation iteration, is affecting the compression speed too much. Strategy $S2$ has the exact same problem.

Strategy $S3$ achieves the correct targeted speed by oscillating between Zstd level 1 and LZ4 acceleration factor 1.

file.nc

Figure 5.18 shows the results of writing the file `file.nc` in a synchronous write call. The achieved compression factors and savings can be inspected in Table A.3. The starting configuration for the compression strategies was LZ4 at acceleration factor 1.

The write call speed compressed with LZ4 starts with over 11 GB/s and reaches up to 15 GB/s. The compression speed even reaches 18 GB/s starting from acceleration factor 5. Strategy $S1$ achieves about 13 GB/s. Strategy $S2$ periodically switches to Zstd level 1 from LZ4 acceleration factor 1 to achieve a slower write call and more compression, but slows down the write call a little to much to about 4700 MB/s. Strategy $S3$ is at 3500 MB/s also a bit on the slow side.

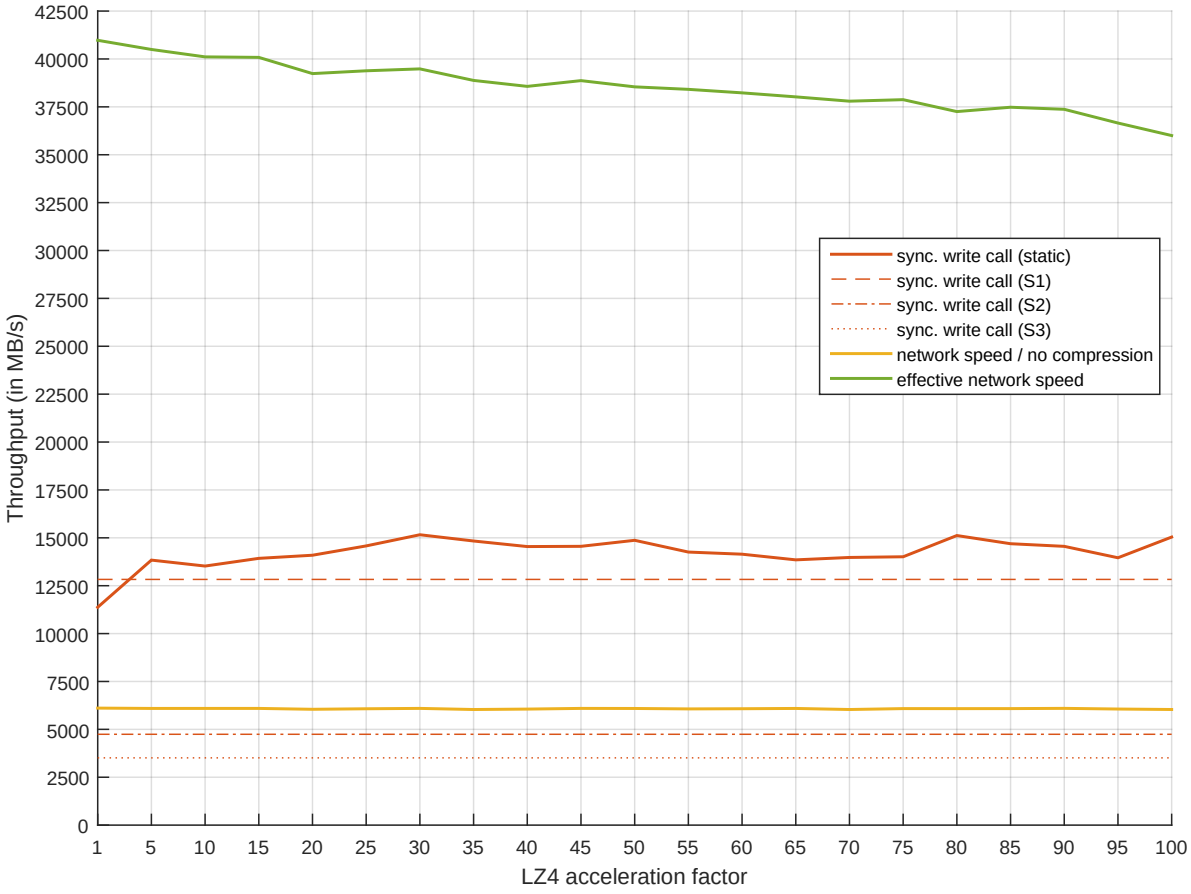

Figure 5.18.: Static compression, dynamic compression and no compression of a synchronous write call writing the file `file.nc` over Infiniband FDR with 4 links.

6. Conclusion and Future Work

This thesis investigated the conditions for client side compression in the context of HPC applications. Three compression strategies having different trade-offs between speed and compression were presented and evaluated for efficiency and worthwhileness. The costs, expectable profit and redemption time of compression has been calculated and evaluated. Furthermore an adaptive compression concept to maximize the compression efficiency and perform controlled compression in application write calls was designed and evaluated.

At network speeds under 10 Gigabit Ethernet the implemented adaptation algorithm in combination with the fairly recent compression algorithm Zstd enables to achieve very increased network transfer speeds, regardless of the write call being synchronous or asynchronous. The transfer speeds can easily reach multiple times the physical network speed, depending on the compressibility of the data. All test files used for the evaluation achieved very respectable compression factors with Zstd.

LZ4 is able to keep up with higher network speeds. At physical network speeds of 10 Gigabit Ethernet decent compression savings and improvements of the network transfer speed are still possible for at least somewhat compressible data. Approaching even faster network technologies like Infiniband FDR with 4 links the performance of LZ4 no longer suffices for asynchronous write calls. Synchronous write calls can still profit from compression savings but speedup is generally not possible unless the data is extremely compressible like the NetCDF output file of the ECOHAM5 simulation.

The costs calculations and the evaluation of the adaptation logic showed that performing compression is very beneficial in most situations, when the adaptation is done correctly. To achieve that the adaptation algorithm continuously seeks to find the most suited compression configuration for the current conditions to efficiently speed up, maintain or prolong the write call duration in a controlled way. Static compression is not suited to reliably achieve any of these objectives for varying files. A static compression configuration can at most consider the common compressibility of data. It can neither exploit exceptionally compressible data nor adapt to sections of different compressibility within the data. In addition it has to be chosen conservatively to be applicable to all situations and not degrade the performance. This wastes a lot of compression potential.

Synchronous write calls of HPC applications turned out to be perfectly suited to perform adaptive compression in with no risk of degrading the network transfer speed considerably for incompressible data. Considering for example a mean compression factor of about 1.8 for data in the climate research field[5], adaptive compression would be very beneficial to achieve compression savings and speedup.

Future work

The adaptation logic has definitely a lot of potential to be be improved. Faster and more precise level jumps adjusted to the operating compression algorithm would improve the performance and efficiency of the adaptation logic. Also the implementation might be improved. The implicit synchronization of the compression threads through the asynchronous queue caused considerable overhead when evaluated with very fast network speeds. At lower network speeds this was not conspicuous.

The file hints could be elaborated further for example by running productive HPC applications with the adaptation logic and evaluating possible shortcomings that might be improvable with certain file hints. The adaptation logic could be for example considered for HSM to compress data on storage before it gets transferred to a tape archive. By providing a file advice containing the amount of time the data should be archived the logic could decide how much resources for compression should be invested. Also a file hints for specific write calls instead of file sections are thinkable. For example to signal asynchronous write calls with a high asynchronism factor, so that the logic can decide to perform compression with the lowest resource impact or not at all.

Bibliography

[1] Lz4 compression. [Online]. Available: https://wiki.illumos.org/display/illumos/LZ4+Compression

[2] Compressing and defragmenting a btrfs file system. [Online]. Available: https://docs.oracle.com/cd/E37670_01/E37355/html/ol_use_case1_btrfs.html

[3] Lustre file system scalability and performance. [Online]. Available: http://doc.lustre.org/lustre_manual.xhtml#idm139974844356928

[4] Intel parallel computing center at universität of hamburg, scientific computing. [Online]. Available: https://software.intel.com/articles/intel-parallel-computing-center-at-university-of-hamburg-scientific-computing

[5] M. Kuhn, J. Kunkel, T. Ludwig, J. Dongarra, and V. Voevodin, "Data compression for climate data," pp. 75–94, 2016, 06. [Online]. Available: http://superfri.org/superfri/article/view/101

[6] Xyratex captures oracle's lustre. [Online]. Available: https://www.hpcwire.com/2013/02/21/xyratex_captures_oracle_s_lustre/

[7] J. Pujar and L. Kadlaskar, "A new lossless method of image compression and decompression using huffman coding techniques," pp. 18–23, 2010. [Online]. Available: http://www.jatit.org/volumes/research-papers/Vol15No1/3Vol15No1.pdf

[8] Smaller and faster data compression with zstandard. [Online]. Available: https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/

[9] Lz4 website. [Online]. Available: https://lz4.github.io/lz4/

[10] Lz4 - very fast compressor. [Online]. Available: http://phantasie.tonempire.net/t95-lz4-very-fast-compressor-x2-07-295-mb-s

[11] Lz4 source code now opened. [Online]. Available: http://fastcompression.blogspot.de/2011/04/lz4-source-code-now-opened.html

[12] Lz4-hc: High compression lz4 version is now open sourced. [Online]. Available: http://fastcompression.blogspot.de/2011/09/lz4-hc-high-compression-lz4-version-is.html

[13] Lz4 - extremely fast compression. [Online]. Available: https://github.com/lz4/lz4/

[14] Zstandard - a stronger compression algorithm. [Online]. Available: http://fastcompression.blogspot.de/2015/01/zstd-stronger-compression-algorithm.html

[15] Facebook open-sources new compression algorithm outperforming zlib. [Online]. Available: https://www.infoq.com/news/2016/09/facebook-zstandard-compression

[16] Zstandard - fast real-time compression algorithm. [Online]. Available: https://github.com/facebook/zstd

[17] Xz utils. [Online]. Available: https://tukaani.org/xz/

[18] S. Sucu and C. Krintz, "Ace: a resource-aware adaptive compression environment," in *Proceedings ITCC 2003. International Conference on Information Technology: Coding and Computing*, April 2003, pp. 183–188.

[19] R. Wolski, "Dynamically forecasting network performance using the network weather service," in *Journal of Cluster Computing*, January 1998, p. 119–132.

[20] P. A. H. Peterson and P. L. Reiher, "Datacomp: Locally independent adaptive compression for real-world systems," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, June 2016, pp. 211–220.

[21] H. Zou, Y. Yu, W. Tang, and H. M. Chen, "Improving i/o performance with adaptive data compression for big data applications," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, May 2014, pp. 1228–1237.

[22] Silesia compression corpus. [Online]. Available: http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

[23] Big buck bunny testfile. [Online]. Available: http://bbb3d.renderfarming.net/download.html

[24] Linux kernel 4.9.5. [Online]. Available: http://ftp.uni-kl.de/pub/linux/kernel/v4.x/linux-4.9.5.tar.xz

[25] Enhanced adaptive compression in lustre. [Online]. Available: http://wiki.lustre.org/Enhanced_Adaptive_Compression_in_Lustre

[26] N. Hübbe, A. Wegener, J. M. Kunkel, Y. Ling, and T. Ludwig, *Evaluating Lossy Compression on Climate Data*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 343–356. [Online]. Available: https://doi.org/10.1007/978-3-642-38750-0_26

Appendices

A. Measurement Data Tables

Static configuration

LZ4											
acc. fac.	1	5	10	15	20	25	30	35	40	45	50
factor	2.10	1.88	1.74	1.64	1.58	1.51	1.46	1.43	1.41	1.38	1.37
comp. sav.	52.3	46.9	42.4	39.0	36.5	33.9	31.4	30.1	28.9	27.4	27.1
Zstd											
level	1	2	3	4	5	6	7	8	9	10	11
factor	2.87	3.00	3.12	3.16	3.22	3.29	3.33	3.37	3.38	3.40	3.41
comp. sav.	65.1	66.6	68.0	68.3	68.9	69.6	70.0	70.3	70.4	70.6	70.6
level	12	13	14	15	16	17	18	19	20	21	22
factor	3.43	3.43	3.44	3.46	3.46	3.53	3.55	3.66	3.67	3.68	3.68
comp. sav.	70.8	70.8	70.9	71.1	71.1	71.7	71.9	72.7	72.7	72.8	72.8

Gigabit Ethernet (117 MB/s)

	asynchronous			synchronous		
	S1	S2	S3	S1	S2	S3
factor	3.09	3.36	3.40	3.22	3.38	3.41
comp. sav.	67.6	70.2	70.6	68.9	70.4	70.6

10 Gigabit Ethernet (1200 MB/s)

	asynchronous			synchronous		
	S1	S2	S3	S1	S2	S3
comp. Factor	2.29	2.78	2.88	2.36	2.97	3.03
comp. Sav.	56.4	64.0	65.3	57.6	66.3	67.0

Infiniband FDR 4x links (6000 MB/s)

	synchronous		
	S1	S2	S3
comp. Factor	1.60	1.91	2.17
comp. Sav.	37.3	47.6	53.9

Table A.1.: Compression factors and savings of compressing the file `silesia.tar` 5 times concatenated following a static configuration and the compression strategies 1-3.

Static configuration

LZ4											
acc. factor	1	5	10	15	20	25	30	35	40	45	50
comp. factor	1.007	1.003	1.002	1	1.001	0.998	0.998	0.998	1	0.997	0.997
comp. sav.	0.7	0.3	0.2	0.0	0.1	-0.2	-0.2	-0.2	0.0	-0.3	-0.3
Zstd											
level	1	2	3	4	5	6	7	8	9	10	11
comp. factor	1.07	1.08	1.09	1.10	1.53	1.53	1.53	1.53	1.53	1.53	1.53
comp. sav.	6.9	7.7	8.5	9.1	34.7	34.7	34.7	34.7	34.7	34.7	34.7
level	12	13	14	15	16	17	18	19	20	21	22
comp. factor	1.53	1.53	1.53	1.53	1.53	1.50	1.50	1.52	1.52	1.52	1.52
comp. sav.	34.7	34.7	34.7	34.7	34.7	33.5	33.5	34.0	34.0	34.0	34.0

Gigabit Ethernet (117 MB/s)

	asynchronous			synchronous		
	S1	S2	S3	S1	S2	S3
comp. factor	1.41	1.47	1.46	1.47	1.46	1.46
comp. sav.	29.2	31.9	31.7	32.0	31.7	31.5

10 Gigabit Ethernet (1200 MB/s)

	asynchronous			synchronous		
	S1	S2	S3	S1	S2	S3
comp. factor	1.09	1.09	1.09	1.09	1.10	1.10
comp. sav.	8.2	8.1	8.5	8.6	8.7	8.9

Infiniband FDR 4x links (6000 MB/s)

	synchronous		
	S1	S2	S3
comp. Factor	1.03	1.03	1.05
comp. Sav.	2.8	3.1	4.9

Table A.2.: Compression factors and savings of compressing the file `matrix.bin` 2 times concatenated following a static configuration and the compression strategies 1-3.

Static configuration

LZ4											
acc. factor	1	5	10	15	20	25	30	35	40	45	50
comp. factor	6.71	6.65	6.58	6.58	6.49	6.48	6.48	6.44	6.37	6.38	6.33
comp. sav.	85.1	85.0	84.8	84.8	84.6	84.6	84.6	84.5	84.3	84.3	84.2
Zstd											
level	1	2	3	4	5	6	7	8	9	10	11
comp. factor	7.78	7.78	7.79	7.79	7.76	7.78	7.78	7.78	7.78	7.80	7.80
comp. sav.	87.1	87.1	87.2	87.2	87.1	87.2	87.1	87.1	87.1	87.2	87.2
level	12	13	14	15	16	17	18	19	20	21	22
comp. factor	7.91	7.91	8.02	8.12	8.12	8.03	8.05	8.06	8.07	8.13	8.17
comp. sav.	87.4	87.4	87.5	87.7	87.7	87.6	87.6	87.6	87.6	87.7	87.8
XZ											
level	1	2	3	4	5	6	7	8	9		
comp. factor	8.23	8.35	8.40	8.45	8.59	8.70	8.70	8.70	8.70		
comp. sav.	87.8	88.0	88.1	88.2	88.4	88.5	88.5	88.5	88.5		

Gigabit Ethernet (117 MB/s)

	asynchronous			synchronous		
	S1	S2	S3	S1	S2	S3
comp. factor	7.69	8.10	8.17	7.78	8.12	8.18
comp. sav.	87.0	87.7	87.8	87.1	87.7	87.8

10 Gigabit Ethernet (1200 MB/s)

	asynchronous			synchronous		
	S1	S2	S3	S1	S2	S3
comp. factor	7.06	7.66	7.69	7.12	7.76	7.77
comp. sav.	85.8	86.9	87.0	86.0	87.1	87.1

Infiniband FDR 4x links (6000 MB/s)

	synchronous		
	S1	S2	S3
comp. Factor	6.48	7.35	7.53
comp. Sav.	84.6	86.4	86.7

Table A.3.: Compression factors and savings of compressing the file `file.nc` following a static configuration and the compression strategies 1-3.

List of Acronyms

API Application Programming Interface.

Btrfs B-tree File System.

CPU Central Processing Unit.

DKRZ Deutsches Klimarechenzentrum.

GPLv2 GNU General Public License Version 2.

HPC High Performance Computing.

HSM Hierarchical Storage Management.

HT Hyper-Threading.

I/O Input/Output.

MDS Meta Data Server.

MDT Meta Data Target.

NUMA Non-Uniform Memory Access.

OSS Object Storage Server.

OST Object Storage Target.

RPC Remote Procedure Call.

ZFS Zettabyte File System.

Zstd Zstandard.

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik 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.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift