Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Master Thesis

Quality of service improvement in ZFS through compression

vorgelegt von

Niklas Bunge

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

Studiengang:	Informatik
Matrikelnummer:	6745206
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn
Betreuer:	Anna Fuchs, Dr. Michael Kuhn

Hamburg, 31. Mai 2017

Abstract

This thesis evaluates the improved use of data compression to reduce storage space, increase throughput and reduce bandwidth requirements. The latter is an interesting field of application, not only for telecommunication but also for local data transfer between the CPU and the storage device.

The choice of the compression algorithm is crucial for the overall performance. For this reason part of this work is the reflection of which algorithm fits best to a particular situation.

The goal of this thesis comprises the implementation of three different features. At first, updating the existing lz4 algorithm enables support for the "acceleration" called lz4fast. This compression speed-up, in lieu of compression ratio, increases write speed on fast storage devices such as SSDs.

Second, an automatic decision procedure adapts the compression algorithms gzip-(1-9) and the new updated lz4 to the current environment in order to maximize utilization of the CPU and the storage device. Performance is improved compared to no compression but is highly depends on the hardware setup. On powerful hardware the algorithm successfully adapts to the optimum.

The third and last feature enables the user to select a desired file-write-throughput. Scheduling is implemented by delaying and prioritizing incoming requests. Thereby compression is adjusted to not impair the selected requirements while reducing storage space and reducing bandwidth demand respectively. By preferring "fast" files over "slow" files - high throughput over low throughput - the average turnaround time is reduced while maintaining the average compression ratio.

Contents

1 Introduction **5**

2 Background **7**
2.1 Data compression 7
2.2 ZFS Internal structure 8
2.2.1 Block size 8
2.2.2 File structure 8
2.2.3 ZFS transaction groups 8
2.2.4 ZFS pipeline 8

3 Design **14**
3.1 How lz4fast compression increases throughput 14
3.2 Auto compression feature 19
3.2.1 Estimation of the queue delay 19
3.2.2 Estimation of the compression throughput 20
3.2.3 Auto controller 20
3.2.4 Expected behavior 21
3.2.5 Summary 22
3.3 QoS compression feature 23
3.3.1 Single dataset 23
3.3.2 Multiple datasets 24

4 Related work **26**
4.1 Automatic selection of different compression algorithms 26
4.2 QoS and compression 27

5 Implementation **28**
5.1 ZFS features per dataset property 28
5.1.1 Adding a new compression property value 28
5.1.2 Adding a new feature 29
5.1.3 Adding lz4fast compression algorithm to ZFS 33
5.2 Floating point and average calculation 35
5.3 Adding auto compression algorithm to ZFS 35
5.3.1 Storage speed measurement 36
5.3.2 Compress_auto_min_queue_delay 37
5.3.3 Auto controller 39
5.3.4 compress_auto_update 42

5.4 Adding qos compression algorithm to ZFS 43

6 Evaluation **47**
6.1 Testing environment 47
6.2 Lz4 acceleration 47
6.3 Auto compression 49
6.4 QoS compression 54

7 Conclusion & future work **56**
7.1 Conclusions 56
7.2 Future work 57

Bibliography **59**

Appendices **62**

A Source code **63**
A.1 Auto compression controller 63

List of Figures **66**

List of Listings **67**

List of Tables **68**

1 Introduction

Purpose of this thesis is the evaluation of opportunities for the improved use of existing compression algorithms in ZFS. ZFS was initially developed by Sun Microsystems and commercialized more than 10 years ago. This flexible file system is capable to process very large quantities of data especially targeting storage applications. Today ZFS is synonymous for data management software used in large server pools which is said to combine high performance, ability for logical partitioning, volume manager software, and data protection algorithms [MNNW14].

The development of fast and efficient compression algorithms make data compression more and more attractive. Meanwhile compression throughput performed on commodity hardware exceeds hard drive throughput while reducing the amount of data by half. Reviewing the latest trend, computational speed is developing faster than storage speed which will improve data compression capabilities even more in the future [KKL16]. The use of compression reduces the amount of data to save storage capacity. On the other hand read and write throughput can also increase as less bytes are transferred. Most file systems provide, if any at all, only a subset of compression algorithms. The ZFS file system supports gzip-(1-9) for high data compression at long runtime, lzjb for fast compression at decent compression ratio, lz4 "a high-performance replacement for the lzjb algorithm" and zle for sparse files [zfsb]. However, the software version of lz4 is outdated. Even faster compression is possible in the new version. This makes an update worthwhile.

The default compression settings in ZFS is "OFF". To enable compression for ZFS datasets the choice of a single data compression algorithm is required. Gzip results in good compression ratios but takes a long time for compression while lz4 is much faster but achieves less compression ratio. The choice of the fastest compression algorithm usually results in an increase of write throughput however not necessarily the best possible result. A complex analysis of the system and the data is necessary to meet the optimal setting.

Due to the fact that further compression algorithms will be added in the future an automatic selection algorithm can save time and effort. Furthermore the periodic update is no longer necessary to adjust change in file compressibility and write throughput caused by fragmentation. The auto compress algorithm is designed as part of this thesis to automatically choose the optimal compression algorithm for the current hardware and data. Fast algorithms are preferred for fast storage devices, low computational power and good compressible data, while slow algorithms, which yield better compression ratios, significantly increase throughput for slow storage devices and fast CPUs.

Recent network routing devices present "quality of service" capabilities to distribute bandwidth individually and prioritize real time traffic to make, for example, video streams

unproblematic in a multi-person household. This management of limited resources has not yet found its way into the file system. However, complex internal scheduling for prioritizing read over write is already implemented for optimal performance in the general case.

Simultaneously running applications share the disk equally. The prioritization of individual data could be useful to separate into time critical grades and increase the overall performance. The additional use of data compression reduces the required bandwidth. Unfortunately this is a time consuming procedure. It has to be fine tuned to not reduce the write throughput.

2 Background

2.1 Data compression

Data compression reduces the number of bytes by eliminating redundancies. A variety of compression algorithms exist. They have often been developed and optimized for a single field of application.

Table 2.1 gives an overview of the already supported algorithms and shows how the new lz4fast throughput exceeds lz4.

Algorithm	Ratio	Compress $\frac{MB}{s}$	Decompress $\frac{MB}{s}$
lz4fast 1.7.5 -17	1.61	785	2601
lz4fast 1.7.5 -3	1.98	522	2244
lz4 1.7.5 (fast -1)	2.10	452	2244
lzjb 2010	1.73	218	402
zlib 1.2.11 -1	2.74	66	250
zlib 1.2.11 -6	3.11	20	267
zlib 1.2.11 -9	3.13	8	269

Table 2.1: Comparison of compression algorithms [lzb]

Compression ratio is defined as $\frac{uncompressed}{compressed}$.

Lz4

lz4 is used for compression and decompression of data files. It is available in a free program library and based on the LZ77 algorithm. A small sliding window searches for redundancies. The substitution table is kept small to fit in the very fast L1 cache and thereby is tuned for speed. The inner workings of LZ4 are described in [lz4c].

Gzip

Gzip is also based on LZ77. The additional used combination with the Huffman encoding achieves a higher compression ratio at an increase in time [gzi].

2.2 ZFS Internal structure

2.2.1 Block size

The default maximum block size for ZFS is 128 KiB. This is not a fixed value but the upper limit for big files. If small files are written in ZFS they are chunked into smaller pieces. Any power from 2 to 512 bytes are possible [**?**]. In comparison to a static block size the variable block size uses storage space more efficient especially for block compression where the ratio is also variable.

2.2.2 File structure

The data in ZFS is stored as a large tree of blocks. Each parent block has block pointers for their children. The bottom level of the tree where the data is stored is called L0 [dno]. The above parent blocks called indirect blocks have a block size up to 128 KiB [ind]. With a block pointer size of 128 byte one indirect block can address up 1024 children.

2.2.3 ZFS transaction groups

ZFS processes data in so called transaction groups [txg, 39] that are uniquely identifiable and pass through three transaction group states one at a time. These are called open, quiescing and syncing.

Whenever data is to be written or modified these request are assigned to the current open transaction group. The size limit is given by a partial amount of available memory. If the transaction group is full no more data is accepted until the current transaction group enters the next state and a new transaction group is opened.

Quiescing is a buffer state where the transaction group waits before it enters the ZFS pipeline.

In syncing state all the data are passed through the pipeline. In this stage all the IO requests are managed as ZIOs. In this ZIO struct all information whether it is a read or write, synchronous or asynchronous, the size before and after compression and which compression algorithm should be used is stored. All ZIOs are ordered hierarchically. This tree is the representation of all the modified (dirty) data in the file system [vid].

2.2.4 ZFS pipeline

When looking at the zio pipeline the data through-passes 3 main stages most relevant for writing. First the data is compressed by the compression algorithm. Then the compressed data is enqueued where it waits in case the storage is busy. Finally the reduced amount of data is written to storage.

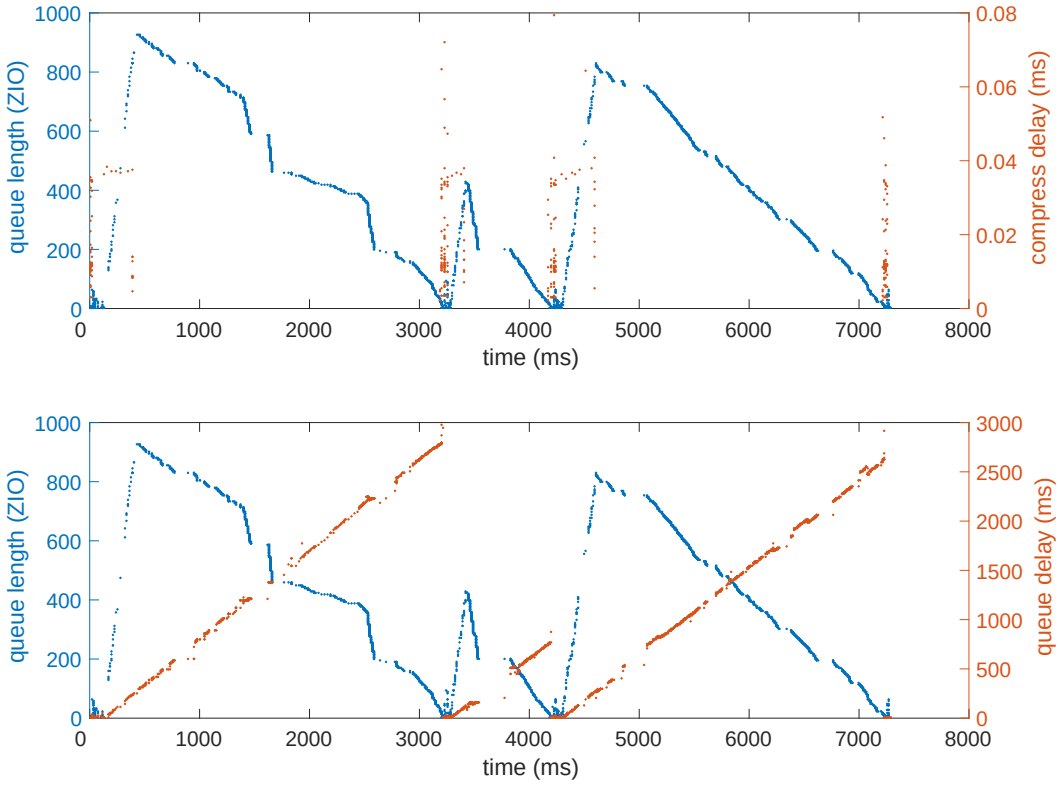


Figure 2.1: Write behavior of compression turned off

In this section data collected from a single file write is analyzed and visualized to demonstrate the pipeline write behavior in ZFS. In Figure 2.1 the change in number of ZIOs in the queue over time is shown in blue on all sub-figures. Shown red in each sub-figure is the delay of: first the duration of how long it took to compress the ZIO and second the time a ZIO has spent in the queue waiting to be written. Each peak represents a transaction group in syncing state. "Transaction groups enter the syncing state periodically so the number of queued asynchronous writes will quickly burst up and then bleed down to zero"[vde, 77]. Even though compression is turned off, the meta-data is compressed by lz4 causing the compression delay. The allocation delay is negligibly small and has no effect on the queue. The spent time in the queue is steadily increasing as the requests are processed after each other.

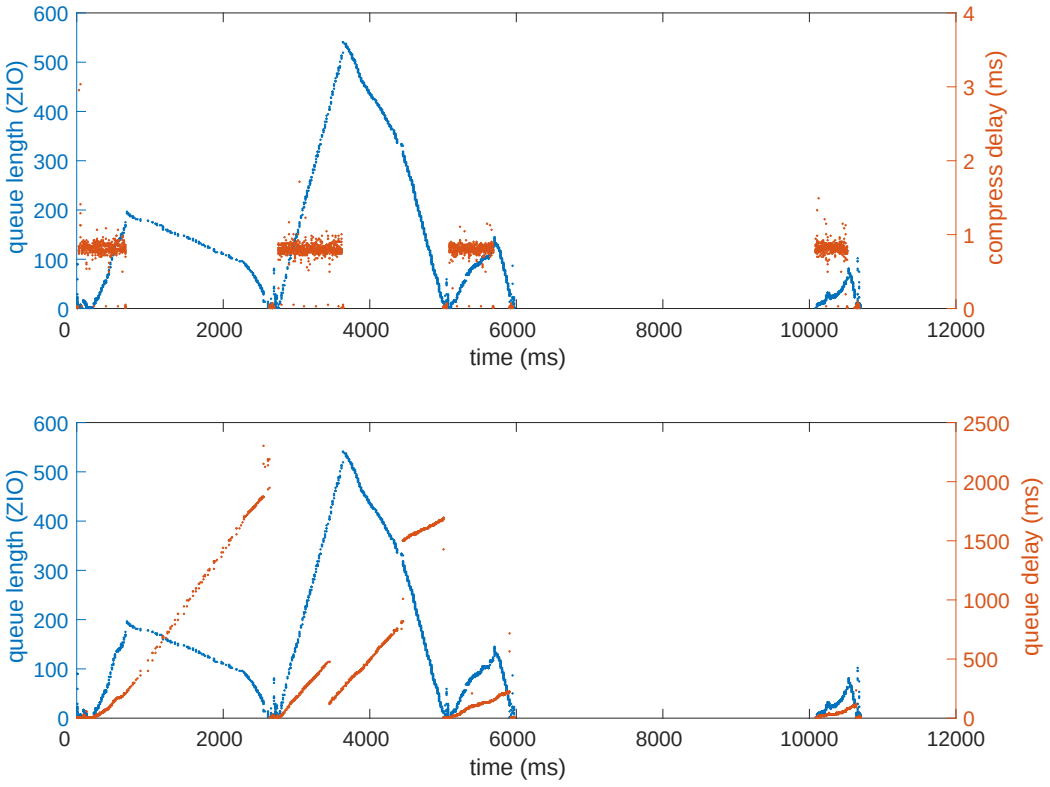


Figure 2.2: Write behavior of lz4

As compression and writing happen in parallel the queue grows if the throughput of the compression algorithm is higher than the transfer rate to the storage device as shown in 2.2. The Lz4 compression only takes place in the beginning of the syncing process and during subsequent writing no CPU resources are required. The ordering of sequential blocks might get mixed up when compressing in parallel but the queue tries to keep the logical block addressing order for asynchronous writes [vde].

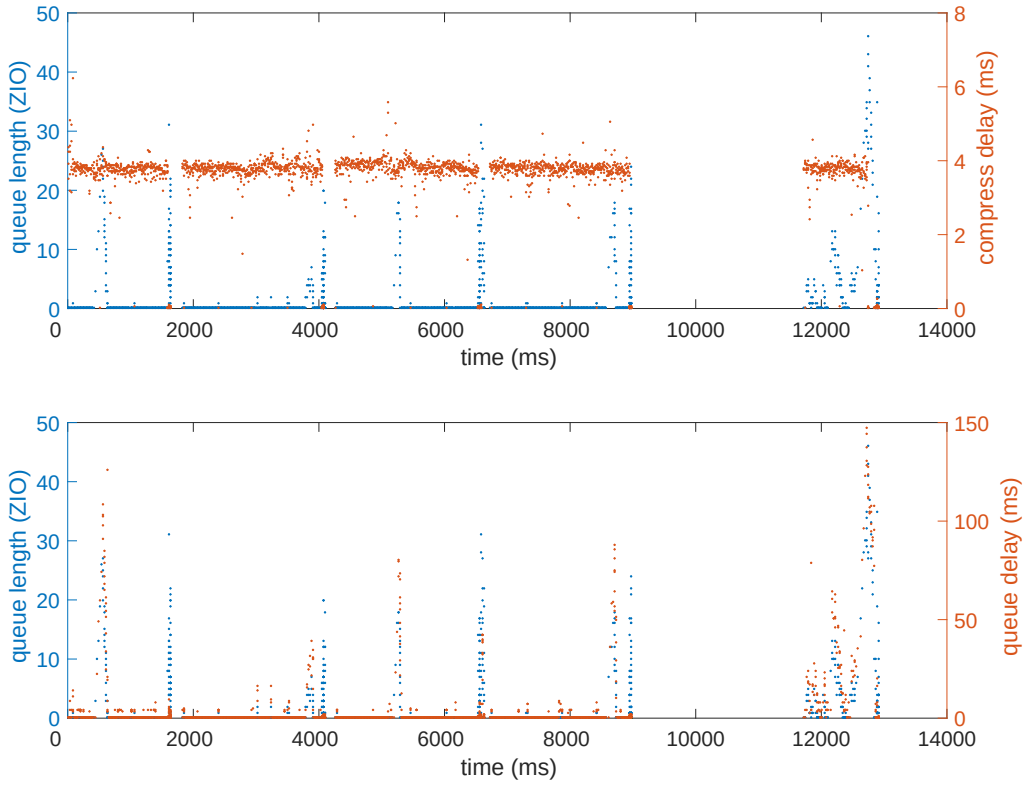


Figure 2.3: Write behavior of gzip-1

When using gzip-1 (Figure 2.3) the storage device is not running at full capacity because the data is too much delayed during compression. The compression takes four times as long as lz4. There is almost no queue, the queue delay is very small.

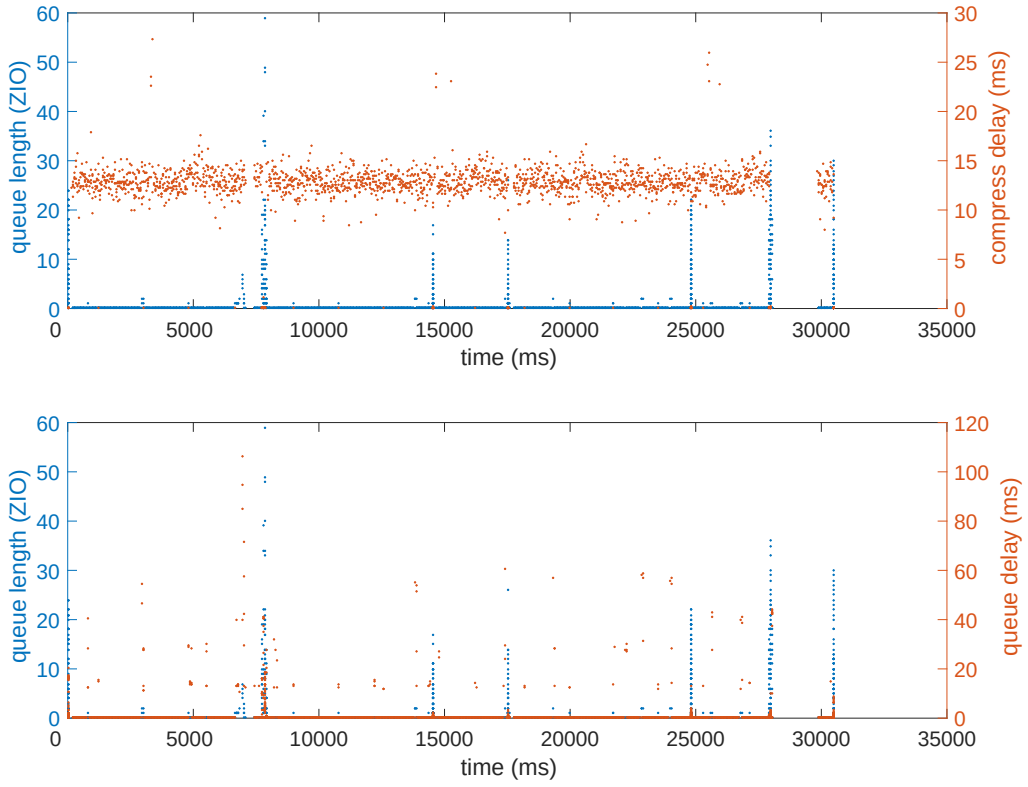


Figure 2.4: Write behavior of gzip-9

Gzip-9 compression (Figure 2.4) takes three times longer than gzip-1. There are a lot of outliers causing high variance. Write requests are delayed even more.

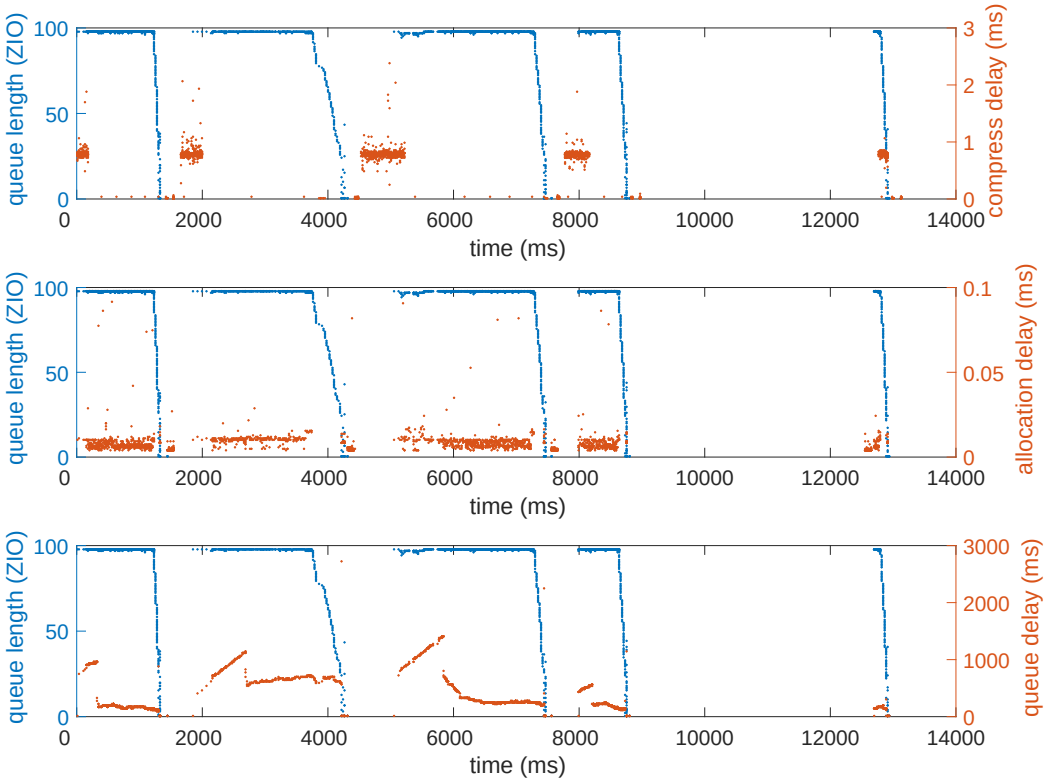


Figure 2.5: Write behavior of lz4 with throttle

When more than one storage device is part of the system the data must be distributed properly. When allocating the data equally in the beginning, fast disks will finish early and have to wait for the slow disks. This is prevented by "throttling block allocations in the ZIO pipeline"[zio]. This feature limits the number outstanding allocations - allocated but not yet written. A small portion is continuously allocated to provide data on demand. Auto compression is an extension to the ZIO write throttle as compression is used to delay the pipeline but with main focus on a single device. Either way the advantages for multiple disks occur. As can be seen in Figure 2.5 the number of ZIOs in the queue hits a limit at 100 block write requests. Compression is not delayed through the throttle. Even though allocations are delayed, the allocation delay measured in Figure 2.5 / 2 is negligibly small.

3 Design

In this chapter the design theory is discussed. First it is analyzed how data throughput can be increased by using the lz4fast algorithm. Second it is described how the auto feature chooses the compression algorithm byitself. In the last section the benefits of the QoS feature are described and a possible design concept is proposed.

3.1 How lz4fast compression increases throughput

Lz4 is a very fast compression algorithm. The actual version 1.7.5 supports acceleration noted as lz4fast and achieves compression speed even higher than the basic lz4. By passing an integer value to the algorithm the compression throughput is increased by about 3-4% [lz4e] for the cost of a decrease in compression ratio. Since lz4 support was added in 2013 the main algorithm of lz4 has been unchanged. By updating to the newest version, ZFS will benefit from the additional acceleration factor to increase write performance. The decompression is unaffected and fully compatible to the improved code. For this reason the lz4fast feature can be implemented read-only backward compatible.

The equations below were designed by approximating lz4 benchmark values taken from [lzb] and [lz4e]. This gives an overview of the compression behavior of lz4fast for nowadays typical data types [sil]. The z4fast-1 algorithm is the same as lz4.

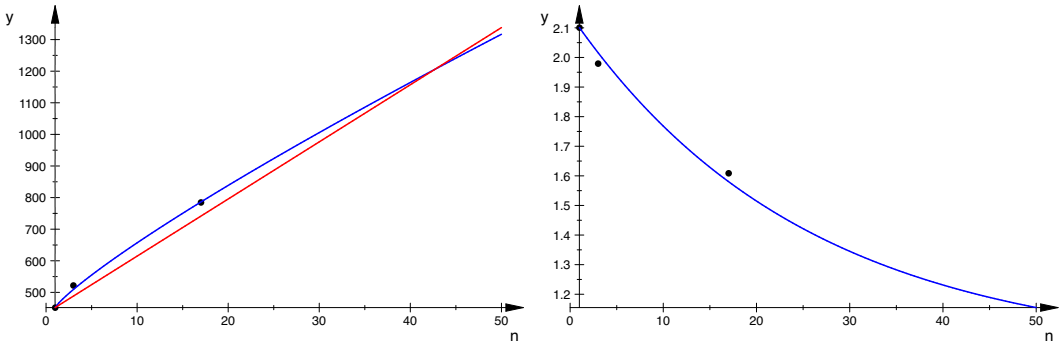
$$compress\_through = lz4throuth + lz4throuth * acceleration * (n-1)^a * b \quad (3.1)$$

$$ratio = 1 + (lz4ratio - 1) * e^{(-(n-1)*acceleration)} \quad (3.2)$$

$$transfer\_throughput = storage\_throughput * ratio \quad (3.3)$$

The compression throughput (Figure 3.1a) and ratio (Figure 3.1b) is calculated visualized based on Equation (3.1) and Equation (3.2) and the data from Table 2.1. The staight line is the approximation of an increase of 4% per acceleration factor (a=1 and b=1). The exponential approximation (a=0.85 and b =1.75) shown in blue is used for a better representation for this particular benchmark where a and b were calculated by regression. The compression throughput varies on different systems and the compression

ratio is dependent on the data. As the transfer throughput (Equation (3.3)) is dependent on the ratio it can be controlled by compression.



(a) Compression throughput in MB/s (b) Compression ratio

Figure 3.1: Compression throughput and ratio as function of the acceleration factor n

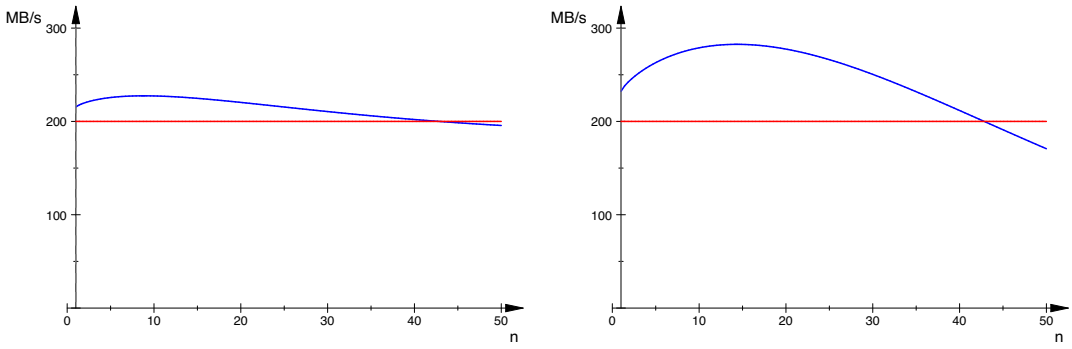
Single file approach

If only a single file needs to be compressed and transfered the Equation (3.4) holds for the throughput. The overall time needed is calculated as sum of compression and transfer.

$$
\begin{aligned}
datathroughput &= \frac{amount\_of\_data}{compress\_duration + transfer\_duration} \\
compress\_duration &= \frac{amount\_of\_data}{compression\_throughput} \\
transfer\_duration &= \frac{amount\_of\_data}{ratio * storage\_throughput}
\end{aligned}
$$

Shortend:

$$
datathroughput = \frac{1}{\frac{1}{compression\_throughput} + \frac{1}{ratio * disk\_throughput}} \tag{3.4}
$$



(a) Sequential data throughput (b) Data reduction throughput

Figure 3.2: Throughput behavior for lz4fast as function of acceleration factor n

The write throughput behavior is demonstrated in Figure 3.2a. The values are again based on the same Table 2.1. The algorithm speed starts at 452 MB/s and the acceleration is 4%. Storage speed is chosen to be 200 MB/s to demonstrate the benefit of lz4fast. The data throughput transcends the physical disk throughput of 200 MB/s. This is possible if the compression algorithm can reduce the amount of data faster than the storage can write. Figure 3.2b demonstrates how fast the compression algorithm processes the data based on how much data is saved at current compression rate calculated by Equation (3.5).

$$data\_reduction\_throughput = compression\_throughput * (1 - \frac{1}{ratio}) \tag{3.5}$$

For sequential data transfer, in this scenario, gzip (zlib) will only become effective for storage throughput below 42 MB/s. A further statement reflected in Table 3.1 is that lz4fast-17 is the most efficient algorithm and the efficiency drops with ascending compression ratio. Also for values above fast-17 there is a loss in efficiency for the lz4 algorithm. Choosing lz4fast-17 over lz4 increases speed by factor 1.7 for the increases in storage requirement by 15%.

Algorithm	Data reduction throughput
lz4fast 1.7.5 -17	$297^{MB}/_{s}$
z4fast 1.7.5 -3	$258^{MB}/_{s}$
lz4 1.7.5	$237^{MB}/_{s}$
zlib 1.2.11 -1	$42^{MB}/_{s}$
zlib 1.2.11 -6	$14^{MB}/_{s}$
zlib 1.2.11 -9	$6^{MB}/_{s}$

Table 3.1: Calculated data reduction throughput based on Table 2.1

Multiple file approach

A filesystem is designed to handle multiple files in parallel. In ZFS even a single file is split up into multiple smaller blocks to add parallelism. The idea is justified that smaller blocks have less redundancies and the benefit of compression is destroyed. But the results of a small user-space benchmark in Table 3.2 show that lz4 achieves a good compression ratio even on small block sizes. A one megabyte file results in almost the same compression ratio as a one gigabyte file. The same holds for the gzip compression algorithm. This makes it possible for ZFS to perform block compression and block writes in parallel to achieve higher throughput improvements describe in the following.

Filesize	Ratio lz4 without acceleration	Ratio gzip-6
1MB	1.71	2.69
1GB	1.74	2.73

Table 3.2: User space compressed file with lz4 and gzip

From the view of a single block, it first has to wait until CPU resources (t_{cr}) for compression are available and then apply the compression (t_c) itself. Then again wait for resources to write to storage (t_{sr}) and finally the writing/transfer to disk (t_s) itself. On one hand slowing down compression increases the write throughput but can also slow down succeeding data blocks by blocking the resources. On the other hand slowing down compression is useful to bridge the time spent anyway in the write queue and would in addition increase write throughput for following data blocks because the amount of data is reduced. The duration the block takes can be calculated by summing up all the four stages as demonstrated in Equation (3.6). Considering a continuous data flow of multiple blocks passing through a pipeline, the throughput is described by a minimum of compression throughput and transfer throughput (Equation (3.7)).

$$throughput = \frac{amount\_of\_data}{t_{cr} + t_c + t_{sr} + t_s} \tag{3.6}$$

$$= min(compression\_throughput, transfer\_throughput) \tag{3.7}$$

With higher compression ratio the compression throughput is decreasing and the transfer is increasing. The maximum throughput can be achieved when the compression throughput is equal to the transfer throughput. This behavior is demonstrated in Figure 3.3. The algorithm speed again is starting at 452 MB/s for lz4fast-1. The storage throughput is chosen to be 500 MB/s. This could be a fast sata 3 SSD hard drive [sat].

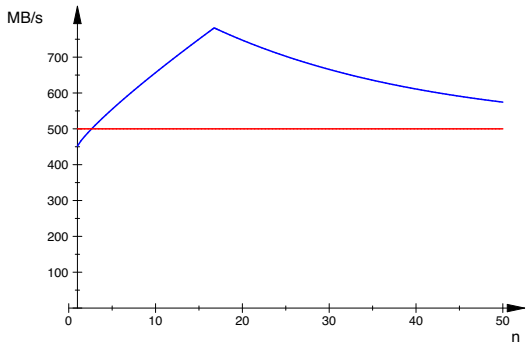


Figure 3.3: Parallel throughput in MB/s on the y axis as function of the acceleration factor n

For lz4, the data throughput is reduced because the compression is the bottleneck. At about lz4fast-3 the compression throughput reaches the same value as the storage throughput. At about lz4fast-20 compression throughput is equal to the transfer throughput and has reached the highest write throughput. Further acceleration will reduce the compression ratio because than the transfer throughput is the bottleneck.

3.2 Auto compression feature

In Section 3.1 the throughput behavior has already been discussed. The result has shown that the maximum write throughput can be achieved when compression throughput is equal to storage transfer throughput. Both CPU and storage will be running at maximum capacity. The auto compression algorithm uses a queue based design. If the queue between compression and transfer is alway kept at a constant level the throughput condition is satisfied. For this approach it is important that compression does not impair the transfer rate to achieve the theoretical gain. The on disk write caches should be helpful to receive enough data to perform the actual data writing independent of the CPU. The goal is to keep the queue at a constant small level by spending as much time on compression as the data would spend in the queue.

3.2.1 Estimation of the queue delay

The more data is waiting in the queue the more time is available. When the ZIO enters the queue it is likely that the data that are already waiting in the queue are served first. The time that has to be waited can be estimated by how fast the queue processed (Equation (3.8)). The *queue_size* in bytes is summed up every time compressed data is added to the queue. The *avg_storage_throughput* is approximated from previous measurements between the average time a ZIO has left the queue and the finish response.

$$est\_queue\_delay = \frac{queue\_size}{avg\_storage\_throughput} \tag{3.8}$$

Queue offset

A reasonable large queue size is very important to maximize throughput. In case of a too large queue less compression is applied to fill the queue resulting in less compression ratio and less data throughput respectively. If the queue is kept too small there is a high risk of it running dry and the storage becomes idle. Furthermore as compression is parallised on multicore systems the logical block addressing order is mixed up. For asynchronous writes the queue is reordered to LBA [vde] restore sequential order.

$$\begin{aligned} &if(queue\_size >= buffer) \\ &\qquad queue\_size- = buffer \\ &else \\ &\qquad queue\_size = 0; \end{aligned} \tag{3.9}$$

A pre defined offset ($buffer$) is used to simulate a smaller queue size as demonstrated in Procedure 3.9. This manipulation effects the queue delay estimation to a lower value hence a faster algorithm will be chosen and the queue will grow. The size of the buffer used in the implementation is discussed in Section 5.3

Handling multiple storage devices

The compression is applied before allocating space on the storage device, as the compressed size is smaller. When assessing the available time for compression all devices must be regarded as it is unclear which one will be chosen to write the data to. Each device keeps track of its own queue and throughput. The auto compression algorithm determines the *minimum_queue_delay* of all devices demonstrated in Equation (3.10). This is the shortest possible case a data block has to wait the before it is transfered to storage.

forall storage devices

$$est\_min\_queue\_delay = min(est\_min\_queue\_delay, est\_queue\_delay) \tag{3.10}$$

3.2.2 Estimation of the compression throughput

Every time a data block is compressed the time between the compression start (t_cst) and the finish(t_cend) is measured. In Equation (3.11) this delay is converted into throughput to handle variable block size (Section 2.2.1). Each algorithm has its own entry. To eliminate variance the average of a small number of blocks is calculated (details in 5.3).

$$compress\_troughput[algorithm] = \frac{uncompressedsize}{t_cend - t_cst} \tag{3.11}$$

3.2.3 Auto controller

The auto controller has a subset of compression algorithms available to choose from listed in Table 3.3. They are in ascending ordered in terms of compression duration and in descending order in term of compression ratio respectively. Lz4fast-100 is the fastest available algorithm and gzip-10 the slowest available to cover a big range of adaptiveness.

level:	level-0	level-1	level-2	level-3	level-4	level-5	level-[6-14]
Algorithm:	lz4fast-100	lz4f-20	lz4f-10	lz4f-5	lz4	gzip-1	gzip-[2-10]

Table 3.3: Chosen compression algorithms

The goal is to keep the queue at a constant small size by estimating the available duration a data block would stay in the queue. The selection algorithm is based on a proportional control system[pco]. The more time available the longer the compression can take as demonstrated in Equation (3.12).

$$y = Kp * e \tag{3.12}$$

y is the duration for compression. The error e represents the available time that is minimized. The more time available the more time can be spent on compression. Kp is

the proportional factor. It is <= 1 because the algorithm is only selected if more time is available than the duration the algorithm needs for compression. To prevent oscillation between very slow and very fast algorithms, the last selected compression algorithm that has been applied is memorized. Only the next slower algorithm can be selected if the condition is fulfilled (Procedure 3.13). Big jumps to faster algorithms are allowed in case the queue empties fast (Procedure (3.14).

$$if \qquad (\frac{uncompressedsize}{avg\_compress\_troughput[level]} < est\_min\_queue\_delay) \qquad (3.13)$$
$$level++$$

$$else$$

$$while(\frac{uncompressedsize}{avg\_compress\_troughput[level]} > est\_min\_queue\_delay) \qquad (3.14)$$
$$level--$$

3.2.4 Expected behavior

As there is no algorithm available for every throughput the alternation of two algorithms converges to the desired throughput. The average duration and the average ratio model the new compression properties shown in Equation (3.15) and Equation (3.16). 1 over $throu1$ is the time needed to compress with compression algorithm one. The average time used for two compression algorithms is the sum of both compression durations divided by 2. The average time is then re-converted into throughput. The expected compression ratio is the average of both ratios.

$$newthrough = \frac{1}{\frac{\frac{1}{throu1}+\frac{1}{throu1}}{2}} \qquad (3.15)$$

$$newratio = \frac{ratio1 + ratio2}{2} \qquad (3.16)$$

3.2.5 Summary

The control system visualized in Figure 3.4 is subdivided into three cyclic steps: Measure, compare and control.

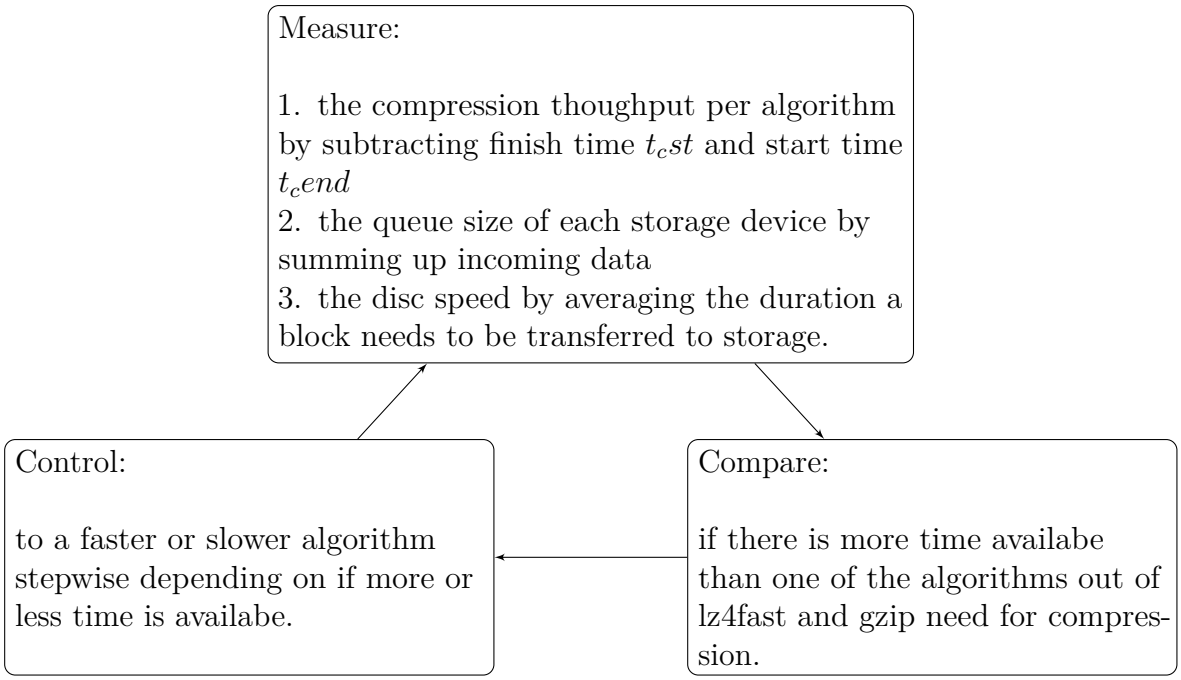


Figure 3.4: Auto feature control system

3.3 QoS compression feature

This feature is divided into two main parts. The first idea is to choose compression by the throughput it can achieve. This could be useful if the user or some application is interested in saving storage but demands less write bandwidth than available. A possible use case is that data is generated or collected slower than it can be stored. The minimal requirements can be defined individually to the current demand.

Second, these chosen requirements should be scalable for multiple applications. Each application can write to its own dataset with its individual bandwidth demand described later in this Section 3.3.2.

3.3.1 Single dataset

Table 3.4 lists the algorithms that are used to approximate the selected throughput. The expected behavior of combining multiple algorithms has already been discussed in Section 3.2.4.

level:	level-0	level-1	level-2	level-3	level-4	level-5	level-[6-14]
Algorithm:	lz4fast-100	lz4f-20	lz4f-10	lz4f-5	lz4	gzip-1	gzip-[2-10]

Table 3.4: Chosen QoS compression algorithms

The pipeline model described in Section 3.1 has shown that if compression is slower than the transfer throughput, this compression also defines the pipeline throughput. With this in mind the overall write speed can be controlled by compression. Procedure 3.17 demonstrates the controlling algorithm. In case the average pipeline speed (avg_pipe) is higher than the requested QoS throughput the compression level is reduce and vice versa.

$$\begin{aligned} if\ (avg\_pipe < QoS) \\ level++ \\ else\ if\ (avg\_pipe > QoS) \\ level-- \end{aligned} \tag{3.17}$$

There are multiple ways to determine the pipeline speed. The easiest approximation would be to measure the compression duration and calculate the throughput. This would only be possible if multiple blocks are not compressed in parallel. To estimate the throughput for parallel execution the measurement needs to be performed over a period of time(Equation (3.18)). $\sum uncompressedsize$ is the amount of data that passed through the compression stage and δt is the interval $t2 - t1$ between measurement start $t1$ and measurement end $t2$.

$$pipespeed = \frac{\sum uncompressedsize}{\delta t} \tag{3.18}$$

If the size of the interval δt is small the selection algorithm can adapt fast to the current data stream. If δt is large more data blocks will be taken into the average. Different QoS parameters are settled at this point. For example it can be declared to provide throughput for a single file from the time it is created until it is destroyed but these guaranties are more difficult to be fulfilled.

In this approach δt is the time difference between a start of a transaction group synchronization and the current time after compression. This could achieve improved results.

3.3.2 Multiple datasets

QoS compression should not only meet the requirements for a single dataset but also for the use of many datasets in parallel. Because the datasets share the same storage they also need to share the storage write throughput with each other.

QoS bandwidth

The QoS property value can also be interpreted as bandwidth that is allocated for a dataset. The maximum available bandwidth is equal to the maximum transfer throughput that can be achieved.

The use of lower QoS bandwidth results in a higher ratio and therefore more bandwidth is available for other datasets as Equation (3.19) shows.

$$used\_bandwidth = QoS * \frac{1}{ratio} \tag{3.19}$$

Hierarchical structure

ZFS datasets allow data organization in separate folders with additional functionality. Properties, like compression, can be set for each dataset individually. Datasets can also be nested into other datasets and form tree structures. The properties are being handed down the hierarchy. Child datasets inherit from their parents. For performance relevant datasets more bandwidth can be assigned than for the irrelevant. In case of inheritance, parents share allocated bandwidth with their children. But nested child dataset can overwrite "inherit" with its own value. It is then treated independently from its parent.

Scheduling

The idea that compression can be used to prioritizes the data is exemplified later in the Related Work 4.2. The user typically uses lz4 for high throughput and gzip to save storage. Lz4 needs less time to compress than gzip. When compression happens in parallel the lz4 data would overtake the gzip data during its long compression. The same would be applied for qos compression. At this point no further work would have to be done.

Unfortunately, this behavior could not be shown in ZFS. The reason for this is the use of transaction groups (Section 2.2.3). Data is collected in memory as TXG before entering the ZIO pipeline. Only one single group at the time can write out data and is finished when *all* data in the group has passed the pipeline.

When writing the same amount of data into multiple datasets they will finish at the same time. For this reason prioritization has to be done right before the transaction group assignment.

If too much high throughput dataset data occupies the TXG the low throughput dataset writes are delayed and cannot achieve their throughput goals. On the other hand if too much low throughput data is assigned to the TXG the sync can not be done in time to meet the criteria for high throughput datasets. The right ratio of "fast" and "slow" data is achieved by fitting the TGX assignment throughput to the compression throughput which is also equal to the TGX synchronization throughput.

The dataset writes can be approximated as running in parallel because each one of them performs its controlling individually.

Example

Given are 2 datasets with different QoS settings.
Dataset 1: QoS 100 MB/s
Dataset 2: QoS 10 MB/s

During one second, 100MB from dataset 1 is assigned to the TGX and 10MB is assigned from dataset 2.
To fulfill the requirements dataset 1 needs 100MB/s throughput for the TGX sync. The time the data for dataset 1 spent in pipeline because of compression is $^{100MB}/_{100^{MB}/_{s}} = 1$ second . Dataset 2 processes 10MB at 10MB/s compression throughput and also needs 1 second. The TGX synchronization finishes after 1 second and the throughput criteria is fulfilled for both datasets.

4 Related work

This chapter provides a short overview of concepts around the use of compression algorithms relevant for the context of this thesis.

4.1 Automatic selection of different compression algorithms

The paper [OOOY13] uses dynamic switching of compression to fully utilize the network bandwidth. The algorithms lzo-(1,5,9,99), zlib-(1,3,5) and lzma-(0,4) are refered to the compression level 1 to 9. Different kinds of file types are transferred through a variety of network bandwidth. Figure 4.1 shows the schematic setup.

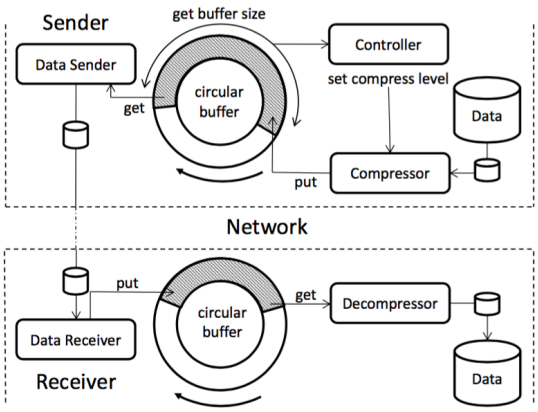


Figure 4.1: Queue based schematic setup

The data passes through the compressor and is stored inside a buffer. When network resources are available the size of the buffer is determined and all the data is sent at once. The controller decides whether the compression level should be increased or reduced depending on a defined threshold for the recent measured buffer size. The subsequent paper [OYO16] chooses different subsets of the available compression algorithms called configurations and compares them on low, middle and high throughput network.

[WWZ16] presents an auto-adaptive model for the Hadoop Distributed File System. Compression rate CR, compression ratio R and transmission rate TR are estimating from historic data. Then the compression algorithm that minimizes following formula is selected:

$$|CR * R - TR| \text{ , } CR > TR \text{ and } R < 0.8$$

The compression algorithms used here are snappy, qicklz and zlib. If none of the algorithms satisfy the constrains compression is turned off for a certain number of batch writes.

[Ehm15] uses a cost function to determine the quality of a currently applied compression algorithm. Important aspects that were considered are the duration of compression, the energy consumed and the achieved compression ratio.

The auto compression designed in this paper combines the ideas of estimating the available time based on compression throughput and transmission and considering the size of the queue.

4.2 QoS and compression

The paper "Design of a QoS Gateway with Real-time Packet Compression"[HLCY07] uses LZSS compression to prioritize network packets. For high priority applications the algorithm uses a small sliding window. Low priority applications are compressed longer by a large sliding window. "Because the fast compression algorithm consumes less processing time than the slow compression algorithm, the high priority application can obtain more transmission chances and more network bandwidth". This approach could not be realized in the ZFS pipeline but would be possible in other parts of the file system.

5 Implementation

5.1 ZFS features per dataset property

ZFS features are an "alternative to traditional version numbering"[zfsa]. When the on disk format is changed by new implementations this is noted as feature to prevent incompatibility to older versions. This section describes how to integrate a feature that is activated by setting a dataset property and deactivated at dataset destruction.

5.1.1 Adding a new compression property value

Dataset property values are stored in a table. Each compression algorithm input string is represented by its own constant value ZIO_COMPRESS_X.

When extending the property list atLine 9 and Line 12, previous ZFS versions are not able to handle the "new" values and might crash. To prevent damage, a feature guards the pool as soon as a dataset uses the "new" property value.

```
1  void zfs_prop_init(void)
2  {
3  ...
4    static zprop_index_t compress_table[] = {
5    { "on",    ZIO_COMPRESS_ON },
6    { "off",   ZIO_COMPRESS_OFF },
7  ...
8    { "lz4",   ZIO_COMPRESS_LZ4 },
9    { "!!NEW!!",   ZIO_COMPRESS_NEW },
10   { NULL }}
11 ...
12   zprop_register_index(ZFS_PROP_COMPRESSION, "compression",
       ↪ ZIO_COMPRESS_DEFAULT, PROP_INHERIT,
       ↪ ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME, "on | off |
       ↪ auto | lzjb | gzip | gzip-[1-9] | zle | lz4 |
       ↪ !!NEW!!",  "COMPRESS", compress_table);
13 ...
14 }
```

Listing 5.1: module/zcommon/zfs_prop.c : Adding a new property value

5.1.2 Adding a new feature

First of all a new feature is created. The list of available features is extended by lz4fast, auto and qos compression.

```
1  typedef enum spa_feature {
2  SPA_FEATURE_NONE = -1,
3  ...
4  SPA_FEATURE_LZ4_COMPRESS,
5  ...
6  SPA_FEATURE_LZ4FAST_COMPRESS,
7  SPA_FEATURE_COMPRESS_AUTO,
8  SPA_FEATURE_COMPRESS_QOS,
9  SPA_FEATURES
10 } spa_feature_t;
```

Listing 5.2: module/zfs/zfs_ioctl.c: List of features

These features depend on existing features that are essential. Lz4fast is an extension of lz4 feature and both "auto" and "qos" make use of lz4 and lz4fast respectively.
The extensible dataset feature takes care of setting the dependent feature from active back to enabled after all datasets that have ever used feature are destroyed to restore full compatibility to older versions [zpo].
In case the feature is active the pool can allow read only backward compatibility if agreeing with the implementation.

```
1  void zpool_feature_init(void)
2  { ...
3    static const spa_feature_t lz4fast_compress_deps[] = {
4    SPA_FEATURE_LZ4_COMPRESS,
5    SPA_FEATURE_EXTENSIBLE_DATASET,
6    SPA_FEATURE_NONE };
7    zfeature_register(SPA_FEATURE_LZ4FAST_COMPRESS,
       ↪ "org.zfsonlinux:lz4fast_compress",
       ↪ "lz4fast_compress", "LZ4fast compression algorithm
       ↪ support.", ZFEATURE_FLAG_PER_DATASET |
       ↪ ZFEATURE_FLAG_READONLY_COMPAT,
       ↪ lz4fast_compress_deps);
8  ...
9    zfeature_register(SPA_FEATURE_COMPRESS_AUTO, ... )
10   zfeature_register(SPA_FEATURE_COMPRESS_QOS, ... )
11 ... }
```

Listing 5.3: module/zfs/zfeature_common.c

If the feature is not enabled the user is refused to set the property. This is the case if a pool is imported that does not support(Line 13) the feature. The implementation is equivalent for lz4 as for the new implemented values.

```
1  static int zfs_check_settable(const char *dsname, nvpair_t
     ↪ *pair, cred_t *cr)
2  {
3    ...
4    switch (prop) {
5      case ZFS_PROP_COMPRESSION:
6      ...
7      if (intval == ZIO_COMPRESS_LZ4) {
8        spa_t *spa;
9        if ((err = spa_open(dsname, &spa, FTAG)) != 0)
10         return (err);
11       if (!spa_feature_is_enabled(spa,
           ↪ SPA_FEATURE_LZ4_COMPRESS)) {
12         spa_close(spa, FTAG);
13         return (SET_ERROR(ENOTSUP));
14       }
15       spa_close(spa, FTAG);
16     }
17
18     if (intval >= ZIO_COMPRESS_LZ4FAST_1 && intval <=
         ↪ ZIO_COMPRESS_LZ4FAST_100) {...}
19     if (intval == ZIO_COMPRESS_AUTO) {...}
20     if (intval >= ZIO_COMPRESS_QOS_10 && intval <=
         ↪ ZIO_COMPRESS_QOS_1000) {...}
21   }
22 }
```

Listing 5.4: module/zfs/zfs_ioctl.c: Refuse property

The LZ4 feature is instantly activated when a pool is created (ZFEATURE_FLAG_-ACTIVATE_ON_ENABLE). Usually the activation is performed at first use of the feature. This would make no difference for lz4 as the initial meta data is already lz4 compressed. For extensional compression algorithms it would be a possible approach to activate the feature after the first file is compressed by the new algorithm as applied in [lz4b]. Unfortunately, the current ZFS version is designed to message an error if unknown compression values are used. Using the new compression property without activating the feature(writing data) will cause an error. For this reason the feature is activated instantly when setting the property.

```
1
2  static int zfs_prop_set_special(const char *dsname,
       ↪ zprop_source_t source, nvpair_t *pair)
3  {
4    const char *propname = nvpair_name(pair);
5    zfs_prop_t prop = zfs_name_to_prop(propname);
6    ...
7    switch (prop) {
8    case ZFS_PROP_COMPRESSION:
9      if (intval>=ZIO_COMPRESS_LZ4FAST_1 && intval <=
         ↪ ZIO_COMPRESS_LZ4FAST_100){
10       dsl_dataset_activate_lz4fast_compress(dsname);
11     } else if (intval==ZIO_COMPRESS_AUTO){
12       dsl_dataset_activate_compress_auto(dsname);
13     } else if (intval >= ZIO_COMPRESS_QOS_10 && intval <=
         ↪ ZIO_COMPRESS_QOS_1000 ){
14       dsl_dataset_activate_compress_qos(dsname);
15     }
16   break;
17 }
```

Listing 5.5: module/zfs/zfs_ioctl.c: Aktivate feature

Validation

The functionality of the feature implementation has been validated by the following test procedure.

1. create a pool with old ZFS version
2. import the pool to new ZFS version -> ok
   a) zpool get feature -> deactivated
   b) zfs set property -> cannot set property for 'pool': pool and or dataset must be upgraded to set this property or value
   c) zfs get property -> not set
   d) zpool upgrade pool
   e) zpool get feature -> enabled
   f) zfs set property -> ok
   g) zfs get property -> value is set
   h) zpool get feature -> active
3. import the pool to old ZFS version -> fail
4. import the pool to old ZFS version as read only -> ok
5. import the pool to new ZFS version -> ok
   a) zfs destroy dataset -> OK
   b) zpool get feature -> enabled
6. import the pool to old ZFS version -> ok

The pool needs to be destroyed to deactivate the feature. It is not sufficient to change the property so that the dataset is not using the feature. Once activated only the destruction can do a reset.

5.1.3 Adding lz4fast compression algorithm to ZFS

This feature extends the compression property values by lz4fast-(1-100) to make use of lz4fast compression algorithm.

```
1 zfs set compression=lz4fast-X
```

Listing 5.6: How to activate lz4fast compression

How to add a new property and a new feature has already been described in Chapter 5.1. This section continues with adding and integrating new values for compression algorithms.

Every compression algorithm is represented by its own value. This value is stored inside each block written to storage to identify the algorithm needed for decompression. The number of values that can be stored on storage is limited to 128 [lz4a]. For this reason in this proposed implementation the enum is split into two parts. The first part is for values that are stored in the block pointer and are written to storage. The second part is for compression algorithm that do not need an additional identification. The lz4fast compressed data uses the same for decompression algorithm as lz4 whose identification value is already existent.

```
1 enum zio_compress {
2   ZIO_COMPRESS_INHERIT = 0, ZIO_COMPRESS_ON,
      ↪ ZIO_COMPRESS_OFF,
      ↪ ZIO_COMPRESS_LZJB,ZIO_COMPRESS_EMPTY,
      ↪ ZIO_COMPRESS_GZIP_1, ZIO_COMPRESS_GZIP_...,
      ↪ ZIO_COMPRESS_GZIP_9, ZIO_COMPRESS_ZLE,
      ↪ ZIO_COMPRESS_LZ4,
3   ZIO_COMPRESS_FUNCTIONS,
4   ZIO_COMPRESS_LZ4FAST_1, ZIO_COMPRESS_LZ4FAST_2,
      ↪ ZIO_COMPRESS_LZ4FAST_..., ZIO_COMPRESS_LZ4FAST_100,
5   ZIO_COMPRESS_META_FUNCTIONS
6 }
```

Listing 5.7: include/sys/zio_compress.h: New compression value

The mapping between compression algorithm and block pointer value is applied after compression(Line 5). The value for the compression algorithm is overwritten by the value for the decompression algorithm.

```
1 static int zio_write_compress(zio_t *zio)
2 {...
3   if (compress != ZIO_COMPRESS_OFF && psize == lsize) { ...
4     psize = zio_compress_data(compress, zio->io_abd, cbuf,
        ↪ lsize);
5     compress = zio_compress_table[compress].bp_value;
6 ... }
```

Listing 5.8: include/sys/zio.c: Compression pipeline stage

In the *zio_compress_table* it is defined which compression and decompression algorithm is represented by which block pointer value. As shown in listing 5.9 lz4 and lz4fast store the same $ZIO\_COMPRESS\_LZ4$ value.

```
1 zio_compress_info_t
    ↪ zio_compress_table[ZIO_COMPRESS_META_FUNCTIONS] = {
2   ...
3   {"lz4", 0,  lz4_compress_zfs, lz4_decompress_zfs,
      ↪ ZIO_COMPRESS_LZ4},
4   ...
5   {"lz4fast-1", 1,  lz4_compress_zfs, lz4_decompress_zfs,
      ↪ ZIO_COMPRESS_LZ4},
6   ...
7 }
```

Listing 5.9: include/sys/zio_compress.c: Compression table

5.2 Floating point and average calculation

The delay is measured by *gethrtime* delivers high resolution in nanoseconds and the data is measured in bytes. When calculating throughput data loss might occur at integer division of bytes and nanoseconds. Floating-point arithmetic is expensive to compute and is not permitted. A transformation from byte per nanosecond to byte per second is applied. To minimize errors transformation 5.1 is used.

$$\begin{aligned} transform &= 1000 * 1000 * 1000 \\ troughputBps &= \frac{byte}{\frac{nanosecond}{transform}} \\ &= \frac{byte * transform}{nanosecond} \end{aligned} \tag{5.1}$$

For moving average calculations without keeping track on the current index equation Equation (5.2) is used. In case the old value is zero the new value forms the new average. Zeros as a new value are ignored.

$$average = (new + old * n - 1)/n \tag{5.2}$$

5.3 Adding auto compression algorithm to ZFS

This feature extends the compression property values by "auto".

```
1 zfs set compression=auto
```

Listing 5.10: How to activate auto compression

The important ZIO pipeline stage where ZFS applies compression is called *zio_write_compress*. The compression property value is accessed at this point to call the data compression function for the selected algorithm. The auto compression needs more information than conventional algorithms to adapt to the environment. For this reason the *compress_auto* wrapper is explicitly called if the auto compression is requested. In figure 5.1 a small subset of the call stack is visualized. If compression is set to one of the existing algorithms the *zio_write_compress* pipeline stages calls *zio_compress_data* both shown in white. If auto compression is chosen the path to goes though the *compress_auto* method first. *zio_unique_parent* is another pre-existing method which returns the parent of the current ZIO. This parent provides feedback information from previous blocks such as which compression algorithm has been used last and how long previous compression took in average. The same parent will be updated with information of the current ZIO at the end of the auto compression procedure by *compress_auto_update* explained last in this section. Before that the implementation of the minimal queue delay estimation and the main controlling algorithm is explained.

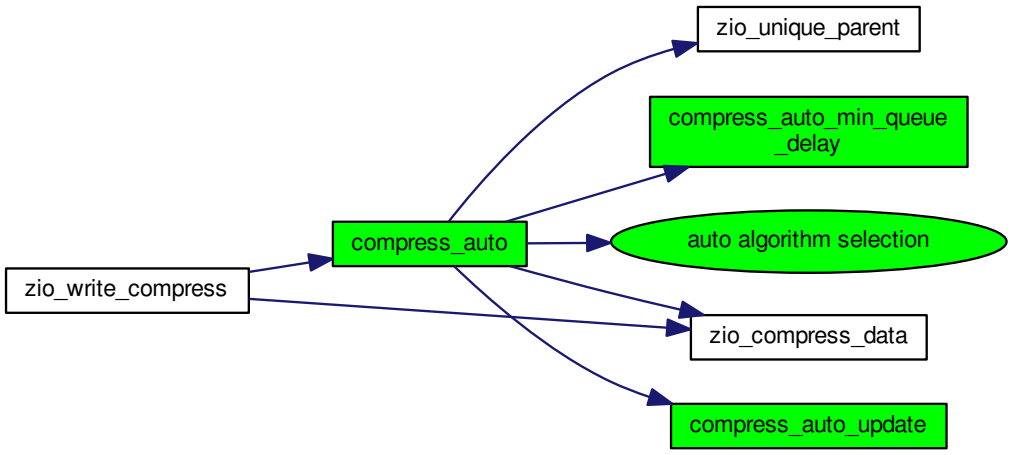


Figure 5.1: Write compress call stack

5.3.1 Storage speed measurement

The write *io_delay* for the device is measured between the two pipline stages *zio_vdev_io_start* and *zio_vdev_io_done*. The device throughput is then calculated by dividing delay and data block size.

The determination of the device throughput is independent regardless of compression. The overhead is negligible as the time measurements for the device *io_delay* are pre-existent. To compensate fluctuation the proximate average value for the last 1000 measurements is calculated.

```
1 void
2 vdev_stat_update(zio_t *zio, uint64_t psize)
3 {...
4 if (zio->io_delta && zio->io_delay) {
5 uint64_t trans = 1000*1000*1000;
6 int n = 1000; // average over 1000 zios
7 compress_auto_calc_avg_nozero((trans * psize) /
    ↪ zio->io_delay, &vsx->vsx_diskBps[type], n);
8 ...}...}
```

Listing 5.11: module/zfs/vdev.c: Disk throughput estimation

5.3.2 Compress_auto_min_queue_delay

Devices (vdev) are ordered in a tree structure. The parent vdevs are logical and have different purposes depending on the chosen software raid. The vdev leafs represent physical devices. The root vdev is passed to this function and transversed recursive. As long as the current vdev is not a leaf all children are asked for their queue delay (Line 9). The minimum of all answers is chosen and passed on to the control algorithm.

```
1  uint64_t compress_auto_min_queue_delay(vdev_t *vd, uint64_t
     ↪ size) {
2    uint64_t min_time = 0;
3    if (!vd->vdev_children) { // is leaf
4      (... to be continied ...)
5      return min_delay;
6    } else {
7      int i;
8      for (i = 0; i < vd->vdev_children; i++) {
9        uint64_t time = compress_auto_min_queue_delay(
           ↪ vd->vdev_child[i], size);
10       if (time) {
11         if (min_time == 0) {
12           min_time = time;
13         } else if (time < min_time) {
14           min_time = time;
15   } } }  }
16   return (min_time);
```

Listing 5.12: module/zfs/compress_auto.c: Queue delay

If the current vdev is a leaf the queue delay can be determined. The queue size and the throughput is accessed through the vdev.

```
1  (continued)
2  if (!vd->vdev_children) { // is leaf
3    uint64_t vd_queued_size_write =
       ↪ vd->vdev_queue.vq_class[ZIO_PRIORITY_ASYNC_WRITE]
       ↪ .vqc_queued_size;
4    uint64_t vd_writespeed =
       ↪ vd->vdev_stat_ex.vsx_diskBps[ZIO_TYPE_WRITE];
5  (to be continued)
```

Listing 5.13: module/zfs/compress_auto.c: Queue delay part1

A buffer is used to keep a small amount of data in the queue to prevent it from running dry. The size must not be chosen to big to not get in conflict with the ZIO write throttle (Section 2.2). By default the *max_queue_depth* is 100 blocks before the block allocation is delayed. To not get in conflict with the write throttle the queue must be kept below the max queue depth. The auto compression algorithm is configured to keep at least 25 (max/4) times the current block size in the queue which is one quarter of the maximum queue depth for allocation throttling. In other words, in case of fluctuation compression can take 25 times as long until the queue runs dry. On the downside the queue size is defined by the compressed data. In case the compression ratio is higher than 4 there will be over 100 ZIOs in the queue to preserve the safety mechanism. Consequently the write throttle is activated. This will cause auto compression to stick with the fastest algorithm.

```
1 (continued)
2   uint32_t max_queue_depth =
      ↪ zfs_vdev_async_write_max_active *
      ↪ zfs_vdev_queue_depth_pct / 100;
3   uint64_t buffer = size * (max_queue_depth / 4);
4   if (vd_queued_size_write >= buffer) {
5     vd_queued_size_write -= buffer;
6   } else {
7     vd_queued_size_write = 0;
8   }
9 (to be continued)
```

Listing 5.14: module/zfs/compress_auto.c: Queue delay part2

After applying the buffer offset to the queue the function now returns the estimated delay in nanoseconds. Division through zero is considered.

```
1 (continued)
2   if (vd_writespeed) {
3     uint64_t trans = 1000 * 1000 * 1000;
4     return ((vd_queued_size_write * trans) / vd_writespeed);
5   }
6   return (0);
7 }
```

Listing 5.15: module/zfs/compress_auto.c: Queue delay part3

5.3.3 Auto controller

The throughput table for every compression algorithm is stored in the parent ZIO (*io_compress_auto_Bps*). Each parent has up to 1024 children (Section 2.2.2) that have access to the previous measurements. Using the parent as storage unit keeps the measurement for the compression and the data stored in the child ZIO logically close together and is easily accessible. As side effect the table has to be rebuilt periodically after one parent and its children have finished.

```
1 struct zio {
2 ...
3   uint8_t    io_compress_level;
4   zio_t      *io_temp_parent;
5   boolean_t  io_compress_auto_exploring;
6   uint64_t   io_compress_auto_Bps[COMPRESS_AUTO_LEVELS];
7   hrtime_t   io_compress_auto_delay;
8 }
```

Listing 5.16: module/zfs/zio.h: Feedback information

All compression algorithms that should be considered to be chosen are listed in an array. The number in total is 14 different levels.

```
1 #define COMPRESS_AUTO_LEVELS 14
2
3 enum zio_compress ac_compress[COMPRESS_AUTO_LEVELS] = {
4 ZIO_COMPRESS_LZ4FAST_100, ZIO_COMPRESS_LZ4FAST_20,
    ↪ ZIO_COMPRESS_LZ4FAST_10, ZIO_COMPRESS_LZ4FAST_5,
    ↪ ZIO_COMPRESS_LZ4, ZIO_COMPRESS_GZIP_1,
    ↪ ZIO_COMPRESS_GZIP_[2-9] };
```

Listing 5.17: Sorted array of compression algorithms

The following describes the decision procedure also visualized in Figure 5.2. At the beginning of a file-write there is no information about how long each compression algorithm takes (v_c not existent). Therefore the first algorithm is selected as it is the fastest. As no control information is available the data is compressed with this algorithm. The compression time is measured and the calculated throughput stored inside the parent ZIO. In addition the index of the current algorithm is stored (*io_compress_level*). The compressed datablock passes though succeeding pipeline stages until it ends up in the vdev queue and is written to disc. Meanwhile the next block enters the compression stage. The throughput v_c for the previous compression algorithm is now available. The expected available queue delay is now determined. The previous datablock has direct influence on the outcome either if it has not reached, through passed or is waiting in the queue. The available queue delay t_{min} is compared to the expected duration $t_c = size/v_c$ of the first compression algorithm.

```
1 (transform * zio->io\_size) /
    ↪ pio->io\_compress\_auto\_Bps[level] <
    ↪ exp\_queue\_delay)
```

Listing 5.18: Compare function of estimated compression and estimated queue delay

In case less time is available than the algorithm needs, there is no other choice than choosing the first algorithm again as there is no faster one available. If the duration for compression and queue delay are equal the choice is clear to choose the same algorithm again. But if more time is available for compression it is determined if there is a slower algorithm available. If so this slower algorithm is checked if the time needed $t_{more} = size/v_{less}$ is below the available duration t_{min}, before this algorithm is selected. Otherwise there is no change. An exception is made if there is not yet an entry for the next slower algorithm. The new algorithm is selected but before compression immediate feedback is given by marking the parent as "exploring" to prevent multi-core systems to simultaneously select an algorithm of unknown compress duration.

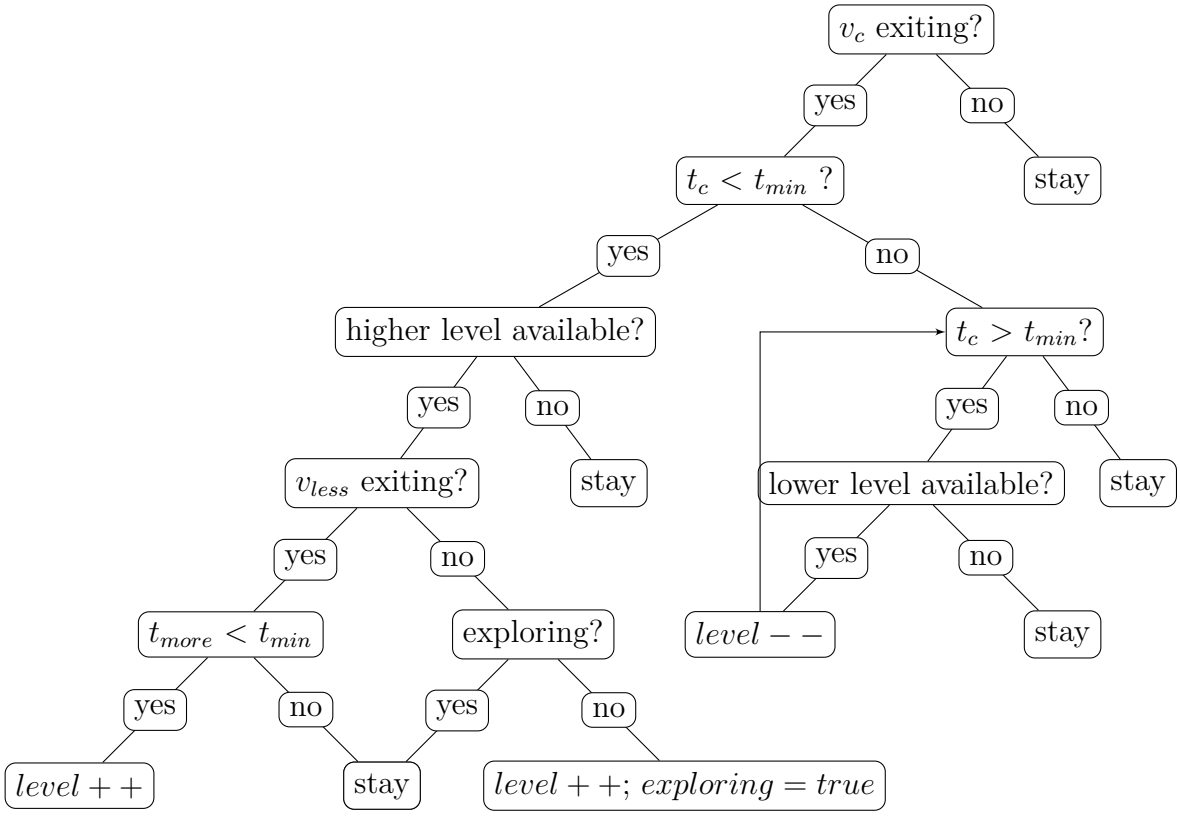


Figure 5.2: Decision tree for the controller

5.3.4 compress_auto_update

After all the steps of the control algorithm have finished, the data has been compressed and the duration of compression has been measured, feedback is given to subsequent blocks. The compression duration is transformed to compression throughput and averaged. In case the compression was an exploration to collect data the current level is not updated to prevent an immediate second exploration of the succeeding block.

```
1  void compress_auto_update(zio_t *zio)
2  {
3    zio_t *pio = zio->io_temp_parent;
4    int n = 10;
5    uint64_t trans = 1000 * 1000 * 1000;
6    uint64_t compressBps = (zio->io_lsize * trans) /
       ↪ zio->io_compress_auto_delay;
7    compress_auto_calc_avg_nozero(compressBps,
       ↪ &pio->io_compress_auto_Bps[zio->io_compress_level],
       ↪ n);
8
9    if (zio->io_compress_auto_exploring) {
10     pio->io_compress_auto_exploring = B_FALSE;
11     zio->io_compress_auto_exploring = B_FALSE;
12   } else {
13     pio->io_compress_level = zio->io_compress_level;
14   }
15 }
```

Listing 5.19: module/zfs/compress_auto.c: Feedback

5.4 Adding qos compression algorithm to ZFS

This feature extends the compression property values by "qos-X"

```
1 zfs set compression=qos-X
```

Listing 5.20: How to activate qos compression

The qos compression is wrapped around the *zio_compress_data* the same as auto compression. The array of compression algorithms is also equivalent to Listing 5.17.

```
1  static int
2  zio_write_compress(zio_t *zio)
3  {
4    ...
5    if (compress == ZIO_COMPRESS_AUTO) {
6      psize = compress_auto(zio, &compress, zio->io_abd,
         ↪ cbuf, lsize);
7    } else  if (compress >= ZIO_COMPRESS_QOS_10 && compress
       ↪ <= ZIO_COMPRESS_QOS_1000) {
8      psize = qos_compress(zio, &compress, zio->io_abd, cbuf,
         ↪ lsize);
9    } else {
10     psize = zio_compress_data(compress, zio->io_abd, cbuf,
         ↪ lsize);
11   }
```

Listing 5.21: module/zfs/zio.c: Compression wrapper

The *wanted_throughput* is set by the user. To estimate the current throughput (*exp_pipespeed_avg*) the measured compression duration average from auto compression can not be used because of parallel execution. Hence, all previous bytes that have already passed the compression stage are divided by the time elapsed so far seen in Line 4.

```
1 enum zio_compress qos_compress_select(zio_t *zio, int
    ↪ wanted_throughput) {
2   zio_t *pio = DMU_META_DNODE(zio->io_prop.zp_os)->dn_zio;
3   uint64_t compress_time = (gethrtime() -
      ↪ pio->io_qos_timestamp);
4   uint64_t exp_pipespeed_avg = (pio->io_qos_lsize * trans)
      ↪ / compress_time;
```

Listing 5.22: module/zfs/compress_qos.c: QoS controller part1

To improve accuracy not the next parent but the most distant reference point is chosen. The first root ZIO, representing the dataset, that is created at sync call (Line 4) allows to calculate the average compression throughput for the data of a complete transaction group for the objset.

```
1 void
2 dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
3 { ...
4   zio = arc_write(pio, os->os_spa, tx->tx_txg, ... , os,
      ↪ ...);
5   /*
6   * Sync special dnodes - the parent IO for the sync is the
      ↪ root block
7   */
8   DMU_META_DNODE(os)->dn_zio = zio;    ...
```

Listing 5.23: module/zfs/dmu_objset.c: Dataset parent ZIO

The *arc_write* method above also calls the *zio_create* method. Here the current timestamp is noted at creation time. Here the QoS condition is settled to be for the single TGX sync.

```
1 static zio_t *
2 zio_create(zio_t *pio, spa_t *spa, uint64_t txg, ...)
3 {
4 ...
5   zio->io_qos_timestamp = gethrtime();
6 ...
7 }
```

Listing 5.24: module/zfs/zio.c: QoS garanties reference point for single datasets compression

The qos control algorithm compares current throughput and wanted throughput and chooses either a slower or fast algorithm level.

```
1  uint8_t next_level = pio->io_compress_level;
2  if (exp_pipespeed_avg) {
3    if (exp_pipespeed_avg < wanted_throughput) {
4      if (next_level > 0) {
5        next_level--;
6    } }
7    else if (exp_pipespeed_avg > wanted_throughput) {
8      if (next_level < QOS_COMPESS_LEVELS - 1) {
9        next_level++;
10   } }
11   zio->io_compress_level = next_level;
12 }
13 return res = qos_compression[next_level];
```

Listing 5.25: module/zfs/compress_qos.c: QoS controller part2

Hierarchy

To compare the throughput across the hierarchy a jointly used structure is required. Because parent dataset is not accessible through the current one an alternative path is chosen. The same *dsl_dir* can be accessed by both parent and child dataset.

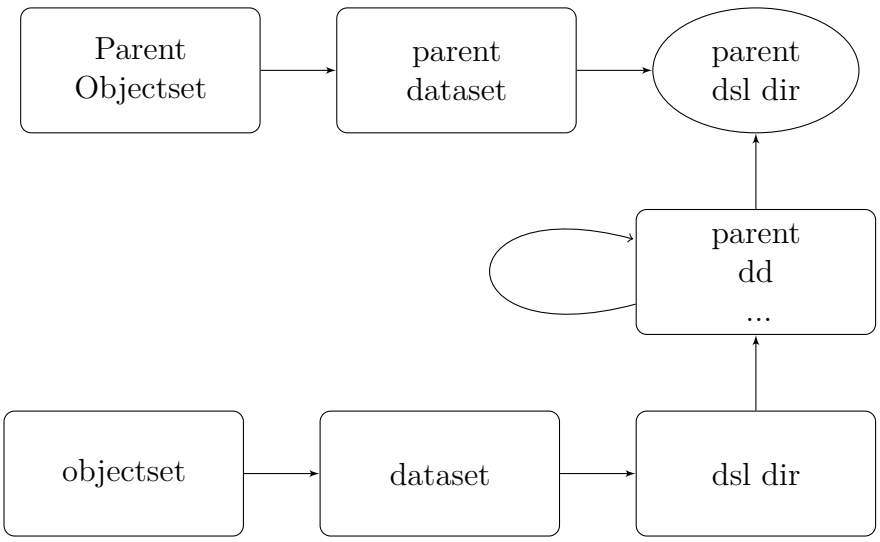


Figure 5.3: Hierarchal implementation

Prioritization

All the bytes of all data block entering the ZFS memory are summed up (Line 6) for each dataset individually in the parent *dsl_dir*. If the amount of data has been added faster than the given bandwidth, the current throughput is reduced by delaying the incoming data (Line 14).

```
1  int
2  zfs_write(struct inode *ip, uio_t *uio, int ioflag, cred_t
       ↪ *cr)
3  {
4  ...
5    mutex_enter(&qos_dd->dd_lock);
6      qos_dd->dd_qos_size += nbytes;
7      size = qos_dd->dd_qos_size;
8      dur = gethrtime() - qos_dd->dd_qos_ts;
9    mutex_exit(&qos_dd->dd_lock);
10 ...
11   uint64_t trans = 1000;
12   uint64_t delay = ((size * trans) /  (through_MBps));
13   if (delay > dur) {
14     zfs_sleep_until(gethrtime() + (delay - dur));
15   }
16 ...
17 }
```

Listing 5.26: module/zfs/zfs_vnops.c: Prioritisation through delaying

The timer and the counted bytes are reset in the same *dmu_objset_sync* method used before where the root ZIO for each dataset is created (see Listing 5.24)

```
1  void
2  dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
3  {
4  ...
5    mutex_enter(&dmu_objset_ds(os)->ds_dir->dd_inherit_parent->
       ↪ dd_lock);
6      dmu_objset_ds(os)->ds_dir->dd_inherit_parent->
         ↪ dd_qos_size = 0;
7      dmu_objset_ds(os)->ds_dir->dd_inherit_parent->
         ↪ dd_qos_ts = gethrtime();
8    mutex_exit(&dmu_objset_ds(os)->ds_dir->dd_inherit_parent->
       ↪ dd_lock);
9  }
```

Listing 5.27: module/zfs/dmu_objset.c: QoS garanties reference point for multiple datasets

6 Evaluation

6.1 Testing environment

All data illustrations for ZFS write behavior such as performance uncritical measurements like the QoS evaluation are done inside a virtual machine. The specification for the host systems are 2,4 GHz Intel Core 2 Duo, 8GB DDR 3 main memory, HDD 5400 rpm with transfer about 50 MB/s. The guest system is running ubuntu 16.04 and is limited to 4GB main memory. The same host is also used in the auto evaluation labeled as low performance setup.

For the lz4fast and high performance auto evaluation measurements are taken on an 2.80GHz Intel Xeon X5560 with 8 cores and 12 GB main memory and HDD speed at about 100 MB/s.

The data used is a subset from the English Wikipedia dump [dat]. For high and low compression ratio testing a database and a binary file from [sil] are chosen.

6.2 Lz4 acceleration

Writing files into a dataset, with the lz4fast compression algorithm enabled, is tested (Table 6.1). As the lz4fast compression algorithm is deterministic the compression ratio is unchanged for the same data. From lz4fast-20 upwards the ratio is about zero for the medium compressible data. For the first test series one CPU core is active and the available storage has 100 MB/s throughput. The throughput for lz4 is the highest and drops with more acceleration. The storage throughput is the limiting factor of the pipeline. For this reason the ratio is important to increase write throughput. Lz4 has the highest ratio and the measured speed of 191 MB/s almost matches 108*1.71 = 184 MB/s. The second test series only uses one CPU core and to simulate fast storage the file system is mounted in memory. The compression algorithm is now the bottleneck. Compression speed starts at 228 MB/s and increases with more acceleration. Lz4fast-10 is 1.6 times faster than lz4 at compression ratio of 1.24. The third test uses 8 cores and is also preformed in memory. At high speeds other pipeline stages effect the throughput more significantly. At lz4fast-5 the compression algorithm is most efficient and can increase throughput. Other compression levels cause inefficient use of CPU resources and reduce throughput.

Name	Ratio	1 core 100 mb/s disk	1 core in memory	8 core in memory
		Throughput in MB/s		
lz4	1.71	191	228	1210
z4fast-2	1.62	180	249	1277
lz4fast-3	1.54	177	266	1322
lz4fast-4	1.47	170	282	1327
lz4fast-5	1.41	162	298	1339
lz4fast-7	1.32	148	329	1316
lz4fast-10	1.24	140	370	1282
lz4fast-20	1.02	113	469	1205
lz4fast-30	1.01	112	546	1220
lz4fast-50	1.004	110	634	1235
lz4fast-100	1.002	110	690	1245
off	1	108	744	1277

Table 6.1: Lz4fast results for write throughput on different hardware

6.3 Auto compression

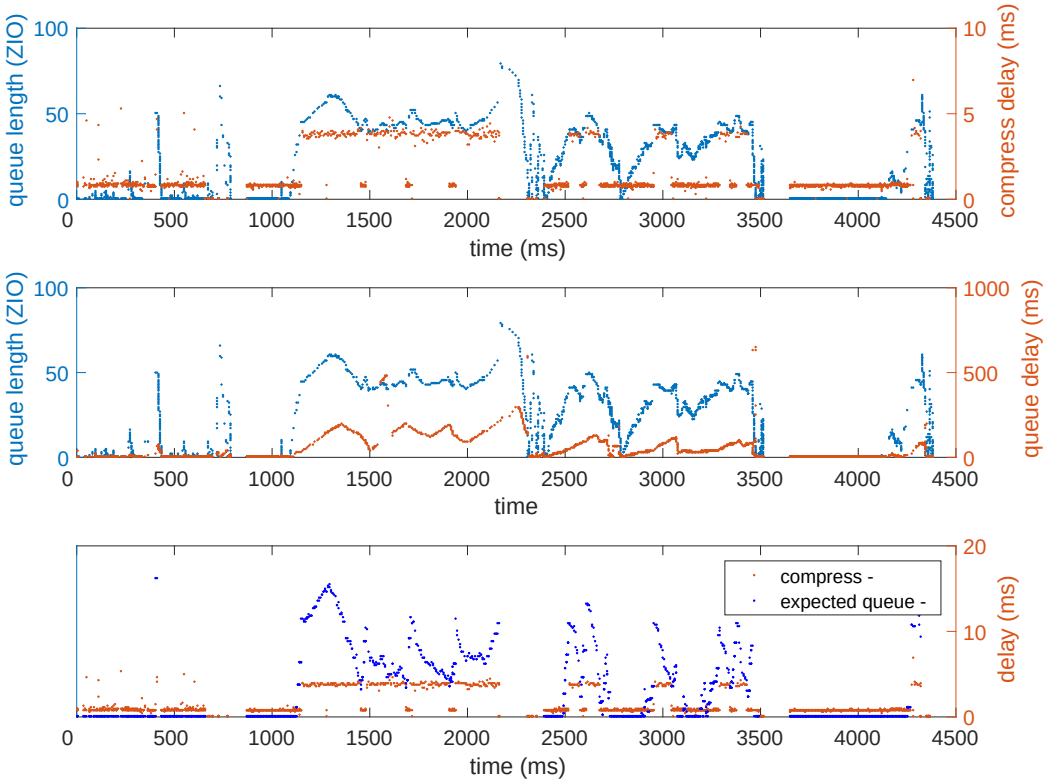


Figure 6.1: Auto compression between lz4 and gzip

A mixture of lz4 and gzip-1 keeps the queue size at constant level of about 50 ZIOs and provides maximum utilization for CPU and storage. In Figure 6.1 / 1 the compression delay is measured. Low values represent lz4 compression and high values represent gzip-1. The bottom Figure 6.1 / 3 is a direct comparison between the current time spent for compression (red) and the expected approximated queue delay (blue) at this point in time. The queue delay never drops below the compress delay.

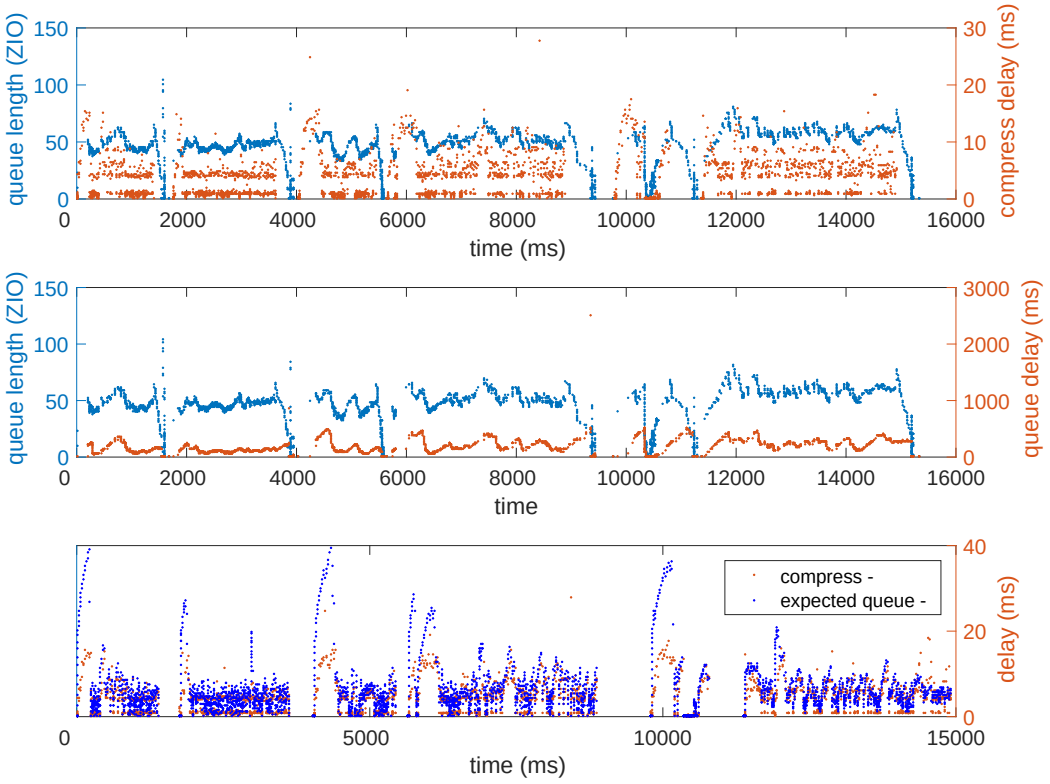


Figure 6.2: Auto compression between lz4fast-[100,20,10,5,1] and gzip[1-9]

With more algorithms available there is a high chance of oscillation as demonstrated in Figure 6.2. The queue is kept to a constant size but there is high switching between multiple levels of lz4fast and gzip. In bottom Figure 6.2 / 3 a high overlapping between compress delay and expected queue delay can be seen.

High performance setup with lz4 and gzip-(1-9)

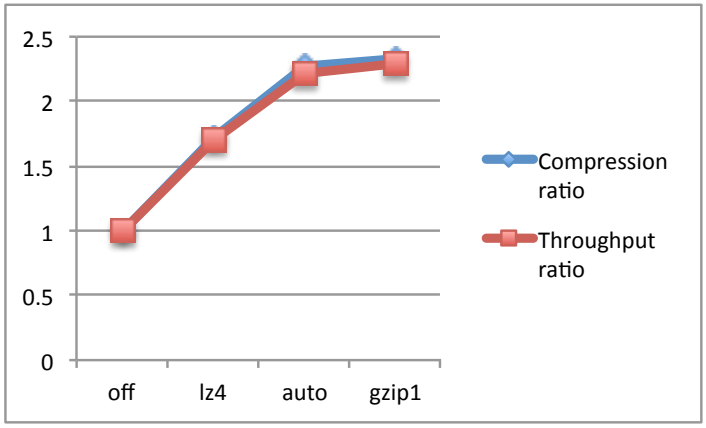


Figure 6.3: Fast CPU

Figure 6.3 shows the results for high performance CPU measurement. The best write throughput could be achieved by gzip-1 with 255 MB/s as it also achieves the best compression ratio. The throughput gain is equal to the compression ratio.

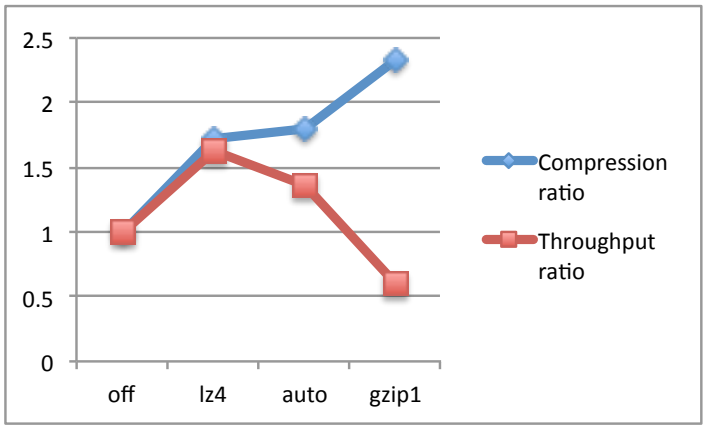


Figure 6.4: Slow CPU

With only one core active gzip compression now performs worse than with compression turned off. Lz4 is the best possible choice. The auto compression throughput is close to lz4 and the compression ratio slightly better.

Low performance setup with lz4 and GZIP-1 for different filetypes

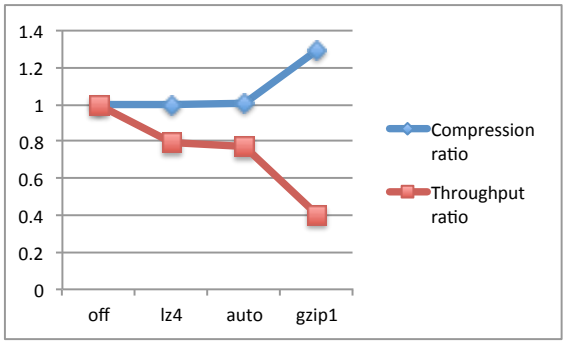


Figure 6.5: Low compressible data

On low compressible data all compression algorithms perform worse than no compression. Auto compression choses the algorithm with highest throughput.

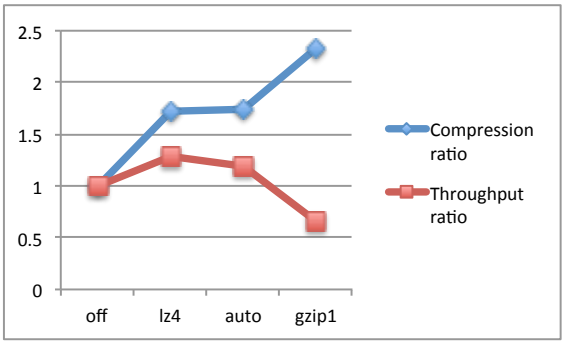


Figure 6.6: Medium compressible data

On medium compressible data lz4 performs better than off and gzip worse. Auto compression is in between off and lz4.

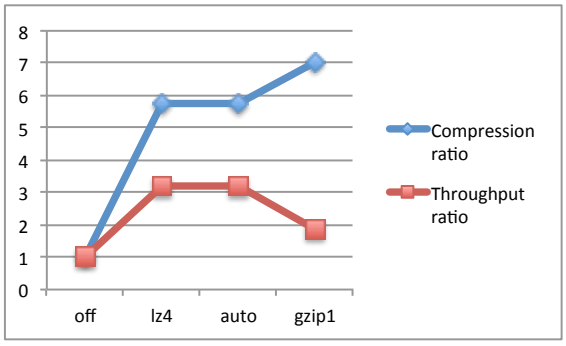


Figure 6.7: High compressible data

On high compressible data lz4 performs best. Auto compression also choses lz4 as there is no faster algorithm available for this configuration.

High compressible data with lz4fast(20,10,5,1) and gzip(1-9)

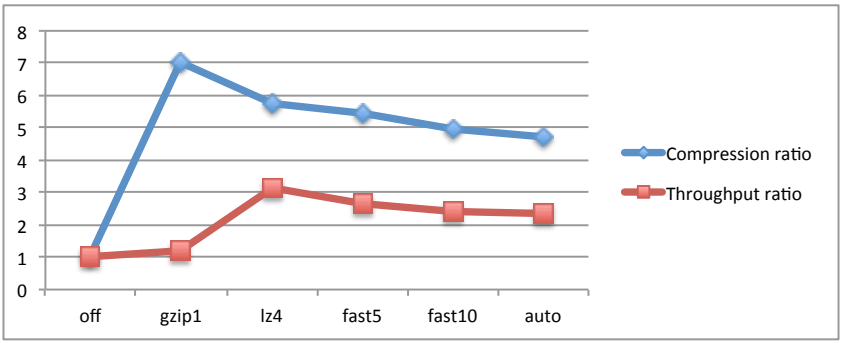


Figure 6.8: Low performance setup 2 cores

Auto compression compresses the data less than lz4 but performs worse. Also Lz4fast-5 and lz4fast-10 also cause a reduce in throughput.

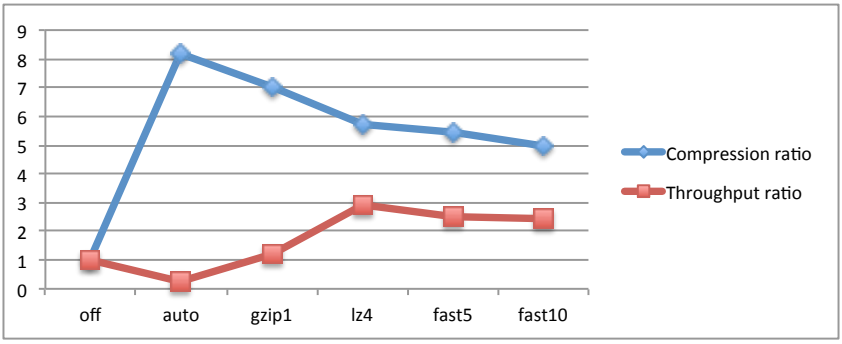


Figure 6.9: Low performance setup 1 cores

If only one CPU is active auto compression choses the slowest compression algorithm. In this case gzip-9. In this the result is even worse than compression turned "OFF".

6.4 QoS compression

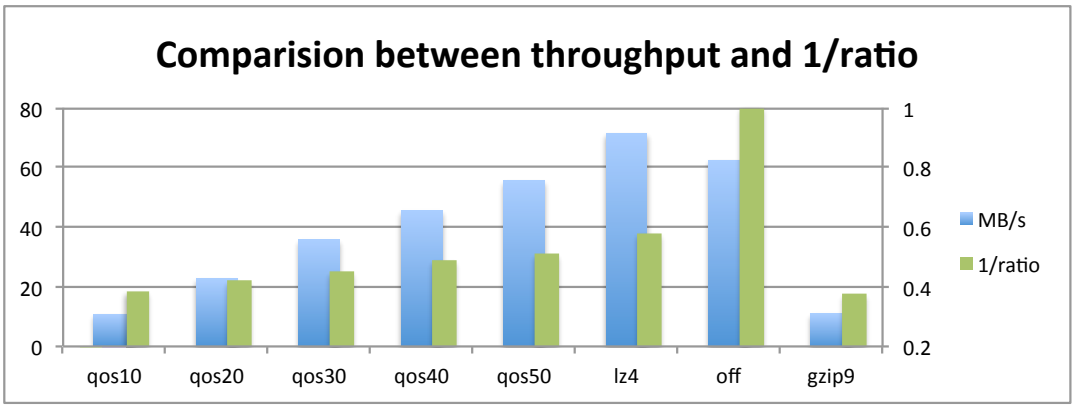


Figure 6.10: QoS achievements for the single dataset

For the single dataset the selected QoS troughput is achieved. Low throughput results in better compression ratio.

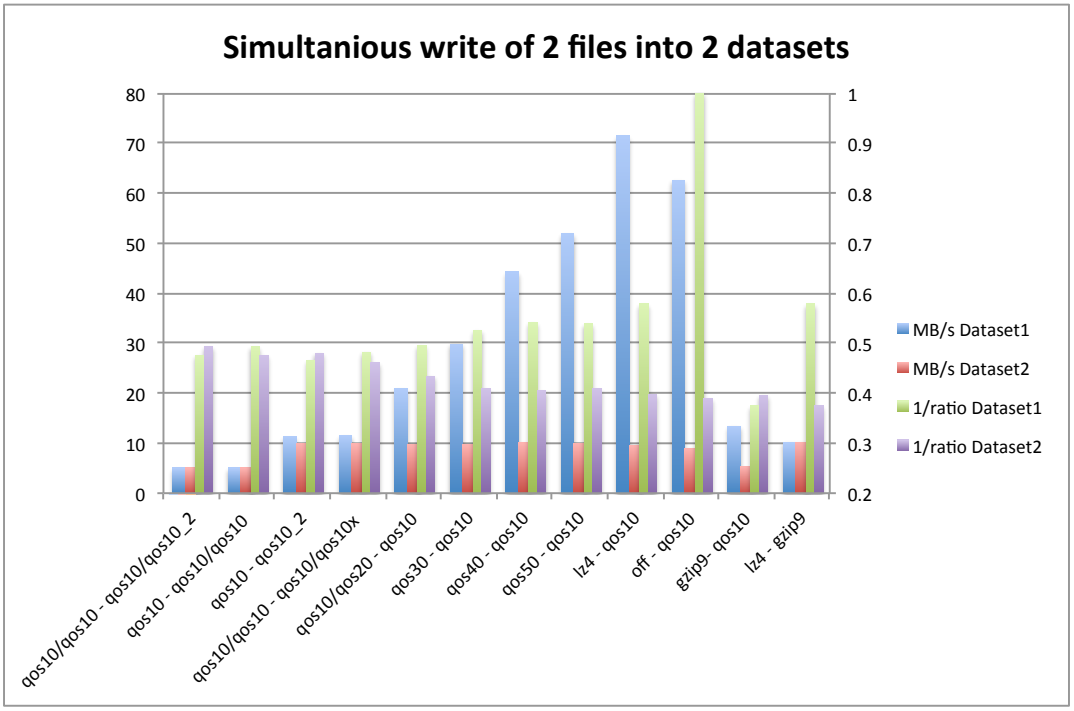


Figure 6.11: QoS achievements for two datasets

Figure 6.11 shows the results of several simultaneous writes into two dataset with different compression settings. The first two tests demonstrate bandwidth sharing between parent and child. The selected 10 MB/s are equally divided into 5 MB/s for the child and 5 MB/s to the other child (test 1) or the parent (test 2). The fourth

test demonstrates the child with overwritten property value. The throughput is then reached independently. Most interesting is the comparison between the last (lz4-gzip9) and "qos10 - qos50". The average compression ratio and average throughput of QoS are both higher than for non QoS.

7 Conclusion & future work

7.1 Conclusions

The use of the right compression algorithm can improve writes in ZFS. By automatically adapting compression to either optimal load distribution or chosen bandwidth, throughput increases and storage space is saved.

The support of more compression algorithms enables more flexibility to the configuration setup. The use of lz4 is most effective for fast storage devices while gzip can improve writes on slow storage hardware and fast CPUs. Lz4fast can slightly can increase throughput for fast storage devices while requiring less compression time (=less CPU resources).

Unfortually, the auto compression did not perform as well on low performance CPU tests. Fast compression algorithms require less CPU and perform well, while the autocompression is designed for high CPU utilization. For low computational power it may be concluded that compression has influence on the measured transfer speed in ZFS therefore the here proposed method does not consider enough parameters to compensate this deviation. The error of compression effecting other pipeline stages in front of the queue should be compensated by the queue offset.

For the use of lz4fast the efficency has to be considered if computational resources are short. For high compressible data the auto algorithm recognizes the reduction in queue size for lz4 and chooses the next available lz4fast. Yet the overall throughput is reduced. The gain in speed is not possible because lz4fast does not achieve higher throughput for this particular testing environment.

The use of many compression algorithms to chose from may cause the implemented controller to oscillate over a greater distance. This has negative influence on performance. An example with values from the lzbenchmark [lzb] demonstrates the problem. Gzip-1 can perform 66 MB/s and achieve a ratio of 2.74. If instead a combination of lz4 with 452 MB/s and gzip-6 with 20 MB/s is used they would achieve a lower throughput at 38 MB/s and a lower ratio at 2.6 as calculated in equation 7.1. This reduces throughput on both sides of the queue and the control algorithm reduces the overall thoughput. Only supporting two algorithms can achieve better results but then there is a smaller range available to adapt to.

$$\begin{aligned} throughput &= \frac{2}{\frac{1}{452} + \frac{1}{20}} = \quad 38 \\ ratio &= \frac{2.1 + 3.10}{2} = 2.6 \end{aligned} \tag{7.1}$$

In the test case of high available computational power about the same performance and compression ratio could be achieved as the best single choice algorithm (gzip-1).

When only choosing between lz4 and gzip the auto compress feature could always increase throughput compared to compression turned off if lz4 was able to increased throughput by itself.

QoS compression proposed a possible implementation to distribute storage bandwidth to multiple datasets. The use of compression is fitted to the transaction based architecture of ZFS. The throughput requirements that are chosen per dataset can be successfully achieved. Prioritizing fast compression datasets over low compression datasets could increase the average throughput and the average ratio because (same as in auto compression) the usage of CPU and storage resources are spent more wisely.

7.2 Future work

The functionality of the features has only been shown for asynchronous writes. Synchronous writes are handled differently in ZFS. More investigation on this end is required.

Other QoS scheduling mechanisms are possible in future implementations. In special cases, for example file download, where the incoming data arrives slower than the possible write speed of the file system, the throughput could be adapted to the current demand. This would act like an extended version of auto compression in consideration of network resources to maximize storage saving without reduction in throughput.

The auto and qos feature could be improved by adding more compression algorithms to fill the gap between lz4 and gzip. These algorithms are available but have to be integrated in ZFS. For example zstd could replace gzip as it is faster in compression and decompression and achieves a better ratio[lzb]. However, an interesting aspect is compression in hardware. Intel's new processors implement the gzip compression algorithm in hardware. This offload reduces CPU consumption [qat]. Further throughput benchmarks are required here to compare to lz4. The fact that even if compression would take longer than with lz4 less CPU resources are needed. The optimal compression algorithm then has to be reevaluated.

Without the compression offload the CPU load of other applications should be monitored and taken into account for the compression selection procedure. Another aspect in saving CPU resources occurs when trying to compress incompressible data. Even if a lot of time is available for compression it should not be spent useless. A hint to ascending blocks not to use compression or at least only use lz4 could be beneficial.

Besides compression, the decompression speed can play an important role for applications. The same aspects as for compression are important. Decompression can be faster than uncompressed reads but the computational power might be taken away from other applications. Reads can be any number of times and the time point is uncertain. Therefore the benefit of compression from this point of view is pure speculation. To stay on the safe side a configuration of algorithms optimized for decompression could be used. Besides LZ4, the LZ4HC is a possible candidate. The compression ratio achieved by this algorithm is not as good as gzip at same throughput but the decompression

speed is as fast as lz4 with about 2200 MB/s. Data reads will always be the same speed and less dependent on high system load. In cases where the system load can be predicted to be low in a later point in time when the data is read algorithms like zstd could perform better. Compression speeds at 242 MB/s are above simple HDD write speed and decompression at 636 MB/s is generally faster than read capabilities but both needs to be checked to be sufficient. Using gzip as it is already implemented in ZFS can still improve reads and writes if an oversupply of computational resources is available.

Besides acceleration for lz4 there has been another addition in the newest version. The streaming API [lz4d] enables compression of multiple adjacent contiguous blocks. However, the block order is important for both compression and decompression. The benefit emerges in a better compression ratio as more than one block is taken into account. A possible application could be storing of encrypted data and other files that are read and modified as a whole.

Bibliography

[dat] About the test data. http://mattmahoney.net/dc/textdata.html. Accessed: 2017-05-01.

[dno] Dnodesync. http://open-zfs.org/wiki/Documentation/DnodeSync. Accessed: 2017-05-01.

[Ehm15] Florian Ehmke. Adaptive Compression for the Zettabyte File System. Master's thesis, Universität Hamburg, 02 2015.

[gzi] Compression algorithm. http://www.gzip.org/algorithm.txt. Accessed: 2017-05-01.

[HLCY07] I-Hsuan Huang, Chun-Shou Lin, Ching-Sung Chen, and Cheng-Zen Yang. Design of a qos gateway with real-time packet compression. In *TENCON 2007 - 2007 IEEE Region 10 Conference*, pages 1–4, Oct 2007.

[ind] Increase indirect block size. http://wiki.lustre.org/images/4/49/Beijing-2010.2-ZFS_overview_3.1_Dilger.pdf. Accessed: 2017-05-01.

[KKL16] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Data Compression for Climate Data. *Supercomputing Frontiers and Innovations*, pages 75–94, 06 2016.

[lz4a] Add lz4hc compression type: Conversation. https://github.com/zfsonlinux/zfs/pull/3908#r42811565. Accessed: 2017-05-01.

[lz4b] Add lz4hc compression type: Files changed. https://github.com/zfsonlinux/zfs/pull/3908/files#diff-97fc713996107a80df7a10a9e9fa7c50. Accessed: 2017-05-01.

[lz4c] Lz4 explained. https://fastcompression.blogspot.de/2011/05/lz4-explained.html. Accessed: 2017-05-01.

[lz4d] Lz4 streaming api basics. https://github.com/lz4/lz4/wiki/LZ4-Streaming-API-Basics. Accessed: 2017-05-01.

[lz4e] Sampling, or a faster lz4. https://fastcompression.blogspot.de/2015/04/sampling-or-faster-lz4.html. Accessed: 2017-05-01.

[lzb] lzbench is an in-memory benchmark of open-source lz77/lzss/lzma compressors. https://github.com/inikep/lzbench. Accessed: 2017-05-01.

[MNNW14] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2014.

[OOOY13] M. Omote, K. Ootsu, T. Ohkawa, and T. Yokota. Efficient data communication using dynamic switching of compression method. In *2013 First International Symposium on Computing and Networking*, pages 607–611, Dec 2013.

[OYO16] K. Ootsu, T. Yokota, and T. Ohkawa. A consideration on compression level control for dynamic compressed data transfer method. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 637–640, Dec 2016.

[pco] P-regler. `http://rn-wissen.de/wiki/index.php/Regelungstechnik#P-Regler`. Accessed: 2017-05-01.

[qat] Gzip compression offloading with qat accelerator. `https://github.com/zfsonlinux/zfs/pull/5846`. Accessed: 2017-05-01.

[sat] Serial ata (sata): Playing an important role in the storage ecosystem through cost, power and performance. `https://www.sata-io.org/technical-overview`. Accessed: 2017-05-01.

[sil] Silesia compression corpus. `http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`. Accessed: 2017-05-01.

[txg] Zfs transaction groups. `https://github.com/zfsonlinux/zfs/blob/master/module/zfs/txg.c`. Accessed: 2017-05-01.

[vde] Zfs i/o scheduler. `https://github.com/zfsonlinux/zfs/blob/master/module/zfs/vdev_queue.c`. Accessed: 2017-05-01.

[vid] Matt ahrens - lecture on openzfs read and write code paths. `http://open-zfs.org/wiki/Documentation/Read_Write_Lecture`. Accessed: 2017-05-01.

[WWZ16] R. Wang, C. Wang, and L. Zha. Pacm: A prediction-based auto-adaptive compression model for hdfs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1617–1626, May 2016.

[zfsa] Feature flags. `http://open-zfs.org/wiki/Feature_Flags`. Accessed: 2017-05-01.

[zfsb] System administration commands. `https://github.com/zfsonlinux/zfs/blob/master/man/man8/zfs.8`. Accessed: 2017-05-01.

[zio] Zio pipeline. https://github.com/zfsonlinux/zfs/blob/master/module/zfs/zio.c. Accessed: 2017-05-01.

[zpo] Zfs pool feature descriptions. https://github.com/zfsonlinux/zfs/blob/master/man/man5/zpool-features.5. Accessed: 2017-05-01.

Appendices

A Source code

A.1 Auto compression controller

```
1  uint64_t compress_auto_min_queue_delay(vdev_t *vd, uint64_t
     ↪ size) {
2
3    uint64_t min_time = 0;
4
5    if (!vd->vdev_children) { // is leaf
6      uint64_t vd_queued_size_write =
         ↪ vd->vdev_queue.vq_class[ZIO_PRIORITY_ASYNC_WRITE]
         ↪ .vqc_queued_size;
7      uint64_t vd_writespeed =
         ↪ vd->vdev_stat_ex.vsx_diskBps[ZIO_TYPE_WRITE];
8
9      uint32_t max_queue_depth =
         ↪ zfs_vdev_async_write_max_active *
         ↪ zfs_vdev_queue_depth_pct / 100;
10
11     uint64_t buffer = size * (max_queue_depth / 4);
12     if (vd_queued_size_write >= buffer) {
13       vd_queued_size_write -= buffer;
14     } else {
15       vd_queued_size_write = 0;
16     }
17     if (vd_writespeed) {
18       uint64_t trans = 1000 * 1000 * 1000;
19       return ((vd_queued_size_write * trans) /
           ↪ vd_writespeed);
20     }
21     return (0);
22   } else {
23     int i;
24     for (i = 0; i < vd->vdev_children; i++) {
25       uint64_t time = compress_auto_min_queue_delay(
26       vd->vdev_child[i], size);
27       if (time) {
```

```
28          if (min_time == 0) {
29            min_time = time;
30          } else if (time < min_time) {
31            min_time = time;
32          }
33        }
34      }
35    }
36    return (min_time);
37  }
```

Listing A.1: Queue delay extimation

```
1  size_t compress_auto(zio_t *zio, enum zio_compress *c,
     ↪ abd_t *src, void *dst,
2  size_t s_len) {
3    size_t psize;
4    spa_t *spa = zio->io_spa;
5    vdev_t *rvd = spa->spa_root_vdev;
6
7    zio_t *pio = zio_unique_parent(zio);
8    uint64_t exp_queue_delay =
       ↪ compress_auto_min_queue_delay(rvd,
9    zio->io_lsize);
10
11   zio->io_temp_parent = pio;
12
13   *c = ac_compress[0];
14
15   if (pio != NULL) {
16     int level = pio->io_compress_level;
17
18     if (pio->io_compress_auto_Bps[level] != 0) {
19       uint64_t trans = 1000 * 1000 * 1000;
20
21       if ((zio->io_lsize * trans) /
           ↪ pio->io_compress_auto_Bps[level] <
22       exp_queue_delay) {
23         if (level < COMPRESS_AUTO_LEVELS - 1) {
24           if (pio->
25           io_compress_auto_Bps[level + 1] !=
26           0) {
27             if ((zio->io_lsize * trans)
28             / pio->
```

```
29              io_compress_auto_Bps[level + 1] <
                  ↪ exp_queue_delay) {
30                level++;
31              } // else stay on level
32            } else if (pio->
33            io_compress_auto_exploring == B_FALSE) {
34              pio->io_compress_auto_exploring = B_TRUE;
35              zio-> io_compress_auto_exploring = B_TRUE;
36              level++;
37            } // else stay on level
38          }
39        } else {
40          while ((zio->io_lsize * trans) /
                ↪ pio->io_compress_auto_Bps[level] >
                ↪ exp_queue_delay) {
41            if (level > 0) {
42              level--;
43            } else {
44              break;
45            }
46          }
47        } // else stay on level
48      }
49
50      *c = ac_compress[level];
51      zio->io_compress_level = level;
52    }
53    hrtime_t ac_compress_begin = gethrtime();
54    psize = zio_compress_data(*c, src, dst, s_len);
55    zio->io_compress_auto_delay = gethrtime() -
        ↪ ac_compress_begin;
56    compress_auto_update(zio);
57    return (psize);
58  }
```

Listing A.2: Auto compression controller

List of Figures

2.1 Write behavior of compression turned off 9
2.2 Write behavior of lz4 10
2.3 Write behavior of gzip-1 11
2.4 Write behavior of gzip-9 12
2.5 Write behavior of lz4 with throttle 13

3.1 Compression throughput and ratio as function of the acceleration factor n 15
a Compression throughput in MB/s 15
b Compression ratio 15
3.2 Throughput behavior for lz4fast as function of acceleration factor n . . . 16
a Sequential data throughput 16
b Data reduction throughput 16
3.3 Parallel throughput in MB/s on the y axis as function of the acceleration factor n 18
3.4 Auto feature control system 22

4.1 Queue based schematic setup 26

5.1 Write compress call stack 36
5.2 Decision tree for the controller 41
5.3 Hierarchal implementation 45

6.1 Auto compression between lz4 and gzip 49
6.2 Auto compression between lz4fast-[100,20,10,5,1] and gzip[1-9] 50
6.3 Fast CPU 51
6.4 Slow CPU 51
6.5 Low compressible data 52
6.6 Medium compressible data 52
6.7 High compressible data 52
6.8 Low performance setup 2 cores 53
6.9 Low performance setup 1 cores 53
6.10 QoS achievements for the single dataset 54
6.11 QoS achievements for two datasets 54

List of Listings

5.1 module/zcommon/zfs_prop.c : Adding a new property value 28
5.2 module/zfs/zfs_ioctl.c: List of features 29
5.3 module/zfs/zfeature_common.c 29
5.4 module/zfs/zfs_ioctl.c: Refuse property 30
5.5 module/zfs/zfs_ioctl.c: Aktivate feature 31
5.6 How to activate lz4fast compression 33
5.7 include/sys/zio_compress.h: New compression value 33
5.8 include/sys/zio.c: Compression pipeline stage 34
5.9 include/sys/zio_compress.c: Compression table 34
5.10 How to activate auto compression 35
5.11 module/zfs/vdev.c: Disk throughput estimation 36
5.12 module/zfs/compress_auto.c: Queue delay 37
5.13 module/zfs/compress_auto.c: Queue delay part1 37
5.14 module/zfs/compress_auto.c: Queue delay part2 38
5.15 module/zfs/compress_auto.c: Queue delay part3 38
5.16 module/zfs/zio.h: Feedback information 39
5.17 Sorted array of compression algorithms 39
5.18 Compare function of estimated compression and estimated queue delay . 40
5.19 module/zfs/compress_auto.c: Feedback 42
5.20 How to activate qos compression 43
5.21 module/zfs/zio.c: Compression wrapper 43
5.22 module/zfs/compress_qos.c: QoS controller part1 43
5.23 module/zfs/dmu_objset.c: Dataset parent ZIO 44
5.24 module/zfs/zio.c: QoS garanties reference point for single datasets compression 44
5.25 module/zfs/compress_qos.c: QoS controller part2 45
5.26 module/zfs/zfs_vnops.c: Prioritisation through delaying 46
5.27 module/zfs/dmu_objset.c: QoS garanties reference point for multiple datasets 46

A.1 Queue delay extimation 63
A.2 Auto compression controller 64

List of Tables

2.1 Comparison of compression algorithms [lzb] 7

3.1 Calculated data reduction throughput based on Table 2.1 16
3.2 User space compressed file with lz4 and gzip 17
3.3 Chosen compression algorithms 20
3.4 Chosen QoS compression algorithms 23

6.1 Lz4fast results for write throughput on different hardware 48

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