SFS: A Tool for Large Scale Analysis of Compression Characteristics

Julian Kunkel

German Climate Computing Center (DKRZ), kunkel@dkrz.de

Abstract. Data centers manage Petabytes of storage. Identifying the a fast lossless compression algorithm that is enabled on the storage system that potentially reduce data by additional 10% is significant. However, it is not trivial to evaluate algorithms on huge data pools as this evaluation requires running the algorithms and, thus, is costly, too. Therefore, there is the need for tools to optimize such an analysis. In this paper, the open source tool SFS is described that perform these scans efficiently. While based on an existing open source tool, SFS builds on a proven method to scan huge quantities of data using sampling from statistic. Additionally, we present results of 162 variants of various algorithms conducted on three data pools with scientific data and one more general purpose data pool. Based on this analysis promising classes of algorithms are identified.

1 Introduction

The tremendous growth of data has lead to an increase in data traffic and storage needs: It was estimated that at the end of 2016, 1.1 Zettabytes per year are transferred via the Internet[1]. In 2011, it was estimated that the available storage of the world would exceed 295 Exabyte of perfectly compressed information [2]. Lossless compression schemes pack the information content of data efficiently, to reduce the costs to store and transfer data. The achievable compression ratio[2] depends on the ability of a compression algorithm to optimally pack the given byte array of data. Depending on the file format and actual data, the Shannon entropy of these bytes may be low, yielding a high data reduction or low. Some compressors are written having certain types of data in mind. For example, Google researchers published several algorithms (Brotli, Zopfli) to compress web-data for transfer. Since data is read many times but needs to be compressed only once, the algorithms aim to optimize compression ratio and decompression speed, but at the cost of compression time.

In High-Performance Computing (HPC), data centers manage Petabytes of storage. Identifying promising compression algorithms is not trivial as the evaluation may be costly, too. Quantifying the compression ratio and the compression speed for a certain algorithm basically requires to analyze these features on the

[1] http://www.livescience.com/54094-how-big-is-the-internet.html

[2] We define the compression ratio as $r = \frac{\text{size compressed}}{\text{size original}}$; inverse is compr. factor.

large data pool. Applying a command line tool such as bzip2 to compress and decompress every file is problematic for data centers, as they host files in the order of several Terabytes. Running slow algorithm may already take several weeks on a single file. An initial case study that has been conducted by the author on the data pool at DKRZ using this strategy took several weeks but only resulted in the scan of 70 (bigger) files out of 300 million (details are described in the evaluation).

Therefore, tools are necessary to optimize the runtime of this analysis but still provide sufficient accuracy. Examples for such characteristics are: the proportion of storage capacity is utilized by a certain file format; the overall compression ratio expected for the storage system when using a particular algorithm. Throughout the paper, the term compression characteristics refers to compression ratio/factor and speed for compression and decompression.

In [5], we introduced a generic methodology to scan file features, i.e., certain file properties, and to infer them for the complete data set using statistical sampling based on weights. In this paper, we describe the open source tool SFS to perform these scans and enables even more efficient scans for determining compression characteristics.

Contributions of this paper are: 1) The introduction of the statistical-file-scanner (SFS); 2) A evaluation of 150+ compression schemes on different data pools. This paper is structured as follows: We give a review of related work in Section 2. The design of SFS is described in Section 3. In Section 4, the convergence behavior of the method will be analyzed (the section is based on the analysis conducted in the paper [5] but presents some additional results). The analysis of data pools using the tool is presented in Section 5. Finally, Section 6 provides a summary and description of future work.

2 Related Work

The related work is structured into compression studies and tools/methods to analyze data. The compression of text data has been studied in various works starting from small studies of a few algorithms on a few files, e.g., in [8] to the evaluation of many algorithms like in Mahoney [7]. Mahoney hosts benchmark results for compressing the XML dump of the English Wikipedia. Everybody can contribute by submitting results, so the page now hosts results for more than 500 compression algorithms.

To optimize the occupied space for backups, deduplication and compression strategies have been evaluated. For example, DARE applies delta compression on top of deduplication [9]. For scientific data, some studies have tested a handful of lossless compression algorithms, e.g., [4], [5], [3]. These studies, however, operate on a rather small data pool and/or required significant compute time. For floating point data, lossy data compression schemes like SZ [1] and ZFP [6] promise much higher compression ratio at the expense of a loss of precision.

The tool LZbench[3] is dedicated to evaluate various lossless compression schemes. Shipping with many state-of-the-art algorithms and even experimental derivatives, it also provides a framework for conducting evaluations and assessing the results. However, it lacks a few features to efficiently scan huge pools of data.

In [5], a statistical method is introduced to predict characteristics of file characteristics for large data sets based on representative samples. This method allows to estimate file types and, e.g., compression ratio by scanning a fraction of the files, thus reducing costs. However, the methodology was demonstrated on a subset of data and by performing costly full file scans.

3 Statistical File Scanner

The Statistical File Scanner (SFS) utilizes the statistical method to estimate the value for a data characteristics of large data sets without actually requiring to scan the full data set [4] Goal is to determine the characteristics of files relatively to the occupied storage space, e.g., 10% of space is occupied by images, or space could be reduced to 50% using compression algorithm X. This is different from just computing the mean statistics across all files as that approach would weight small files equally to large files. The paper in [5] shows the approach to determine this value correctly based on samples. Particularly, one must estimate the mean weighted by the size of the files. The more samples are taken, the smaller are the confidence intervals in the result. The file scanner used for the previous paper is now refurbished and geared towards the needs for conducting large scale studies on compression. SFS follow the methodology of this paper and is implemented as a set of scripts built on top of an extended LZbench.

3.1 Scanning Methodology

The steps for determining the characteristics of files works as follows:

1. Determine the complete list of files together with their file sizes. This scanning the file tree is costly but far less than compressing the full data; it takes a couple of hours to scan 300 Million files on DKRZ on one node.
2. Create task lists for a number of threads and random samples. Firstly, read the list of available files assign a probability to each file based on its size $p(\text{size}) = \text{size}/\text{size all files}$. Independently draw files for each thread and task. Indeed (large) files may be selected multiple times in this process!
3. Scan the files. Therefore, threads operate independently on their task list, scan a file and produce their output with the desired properties.
4. Integrate results. All outputs are parsed, cleaned and input into a database.

[3] `https://github.com/inikep/lzbench`

[4] The tool is publicly available: `https://github.com/JulianKunkel/statistical-file-scanner`.

5. Analyze results using external tools. The mean of a characteristics such as compression ratio is simply computed by the mean characteristics across all scanned samples (this includes duplicates of large files).

There are two variants of the scanning process: The **full mode** scans a file and when computing the mean, it is weighted according to the number of times it is drawn in Step 2. For economical reasons, in this case, one file scan is sufficient but the number of selections must be remembered. The **partial mode** analyzes a random chunk of each chosen file. SFS was originally designed having full file scans in mind, however, for large files and slow compression algorithms this approach is not feasible. In a test on DKRZ's supercomputer Mistral, scanning 3 TByte of data took 6 node weeks on 36 cores.

The partial mode is useful for compression: each time a (huge) file is selected, it picks a random fixed size chunk from the file and determines its compression characteristics. Using this strategy, one could define to use, e.g., 10,000 random samples of 10 MiB from the files on a storage pool of arbitrary size and compute the mean across all these chunks[5]. This allows to estimate and limit the runtime. Intermediate results can be used for preliminary analysis and observing convergence of the method.

Analyzing intermediate results is supported in both modes, but in full mode with caution as the compression time depends on the file size (and selected memory limit). In a pathetic case, one has 100 small files and 10 large files; starting 10 threads and stopping them quickly leads to the situation where some small files are scanned but no large file.

3.2 Architecture of SFS

SFS uses Bash and Python scripts for the scanning and LZbench as file scanner. Time consuming steps of the processing are parallelized[6]:

1. Scan directories. This Bash script starts from a work directory and runs one `ls -R` in parallel for each top level directory found – up to a chosen number of processes. Each top-level directory is stored as project name for the analysis.
2. Select Files for a user-defined number of threads and random samples. This Python script parses the scanner output and creates one task file for each thread. In full mode, it detects if a file is repeatedly assigned to a thread and marks this file to prevent repeated parsing.
3. Run scanner. This Bash script runs a number of worker threads on the current node – each working on their task list (see below).
4. Create DB – parses output; checks for some inconsistencies, integrates the results into an SQL-lite database. Allows incremental import.
5. Analyze results of the database, e.g., using R.

[5] Small chunks are weighted equally to large chunks but are drawn less likely.
[6] Supporting single node parallelism, the actual scanning process can be distributed across multiple nodes.

Scanning thread: A thread reads the task list sequentially. In each line a file to scan with its size is specified. It first checks if the file's size is still the same when creating the file list. Otherwise it ignores this file as the file was modified in the mean time and, thus, the probability to select the file changed[7]. It then first runs the `file` and `cdo`[8] commands to determine the file types to allow an analysis based on the file format. Next the modified LZbench is run and output is stored into a thread-local log file.

LZbench scans a single file by reading a fixed amount of data[9] and outputs for each algorithm the name, the compression and decompresison time[10] and the original and compressed size of the chunk. It ensures that even for small files a compression/decompression phase is repeated to run for at least 1 second. The algorithms to test can be configured in the script, by default up to 160 variants of algorithms are tested. LZbench does not measure time for I/O. To make LZbench work on the large scale, several modifications to LZbench have been done[11]. Noteworthy changes are: 1) restricting the available main memory [12]; 2) to randomize the data retrieval from huge files, i.e., by randomly sampling a fixed amount of data from huge files. While the first extension enables scanning of huge files in chunks and reports the statistics for each chunk, the latter allows to randomly pick a position in a large file and scan a determined amount of data from this position.

The scanning thread distinguishes between the full and partial mode: In full mode, one scan of a file is sufficient, therefore, repeated scans must be prevented. The Python script will already check if a file is selected again by the same thread and, thus, has been scanned already by this thread when sequentially processing the file list. However, a file may have been selected for different threads. The first thread that reaches the file in the list must process it[13]. Since we do not know which thread will process the file first, we keep a SQLlite database that records all completely scanned files.

The database is also used for a lightweight restart mechanism. Before starting to scan the next file, each thread will update its position. Upon start, it checks the task list and database to identify where to restart. In partial mode, a file is scanned repeatedly but (typically) on different regions. Therefore, multiple scans of the file are allowed and handled correctly.

---

[7] We assume the files on the large file system do not change rapidly between scanning of the directories and running LZbench.

[8] The Climate Data Operator tool `https://code.zmaw.de/projects/cdo` supports many scientific file types and is more robust to detect them correctly.

[9] In full mode, this step is repeated in chunks of the specified memory size.

[10] If an algorithm is unable to decompress the data, then the decompression time is recorded as 0.

[11] Most extensions are now merged back in the public development branch of LZbench.

[12] Previously LZbench required to load the complete file into memory prior analysis which is prohibitive for huge files.

[13] This is necessary for analyzing intermediate results.

3.3 Limitations of the Scanning Process

The determined performance characteristics of algorithms depend on the CPU performance and memory throughput. However, depending on the evaluated algorithms and the number of concurrent threads, the problem becomes either bound by compute or memory. For example, when using fast compressing algorithms, they are memory bound while slow ones are CPU bound. Modern many-core architectures need to run more than one memory intensive thread to saturate the memory bus anyway, so running a few threads even with memcopy does not harm the individual performance much. At some point, the memory bus is saturated. In that sense, the selection of algorithms and the number of threads to run influence the reported performance.

LZbench performs the operations in the two phases: read data, then apply all algorithms. If only a single fast algorithm is selected and I/O is much slower than memory throughput, the ensemble of threads is typically reading the input file; even with a number of threads equal to the available cores, it is expected that the ensemble is not memory bound. With fast I/O and when using only fast algorithms and many threads, the ensemble may become memory bound. Due to these considerations, in typical setups (evaluate different algorithms, use a number of threads up to the number of physical cores), it is expected that memory congestion is low and results are comparable to single core performance.

It might be the goal to measure the performance of a compression algorithms when using all existing cores at the same time. In particular, this might be relevant for bulk synchronous HPC applications. This is not yet possible with SFS and the single threaded LZbench.

4 Convergence of the Sampling Strategy

The main requirement of the used sampling strategy is to allow to estimate the compression characteristics for the pool of data. That means, for example, that the determined compression ratio should describe the saved storage space when using a compression algorithm. Note that this is not the arithmetic or harmonic mean ratio across all files. Instead, we want to compute the mean compression ratio according to Equation (1).

$$ratio_{size} = \frac{\sum_f \text{compr.size}(f)}{\sum_f \text{file size}(f)} \tag{1}$$

To compute the $ratio_{size}$ with sampling, we apply the following strategy: Pick a random sample from the file list based on the probability defined by filesize/totalsize. Draws from the list is done with replacement, i.e., we never remove any picked file. For each chosen file, the compression characteristics (including $ratio_{size}$) are computed. Then the arithmetic mean can be computed across the invidual results to obtain the correct characteristics. Thus large files are more likely to be picked but each time their characteristics are weighted identically as

for small files. In [5], it is demonstrated that this is the correct approach and a simply random sampling strategy does not provide converging results.

To evaluate the convergence for several characteristics, we use the data from [5]. We simulate the convergence, by first capturing the characteristics of a large data pool (380k files and 53.1 TiB capacity). Then we randomly draw a certain number of samples using our sampling strategy and compute the mean. This process is repeated 100 times each for an increasing number of samples. Box plots about these characteristics are shown in Figure 1. The figure shows the compression ratio for several algorithms, the determined proportion a given file type occupies on the storage and compression and decompression rate. It can be seen that with an increase of samples, the number of outliers is reduced, and the interquartile range becomes shorter. For example, by drawing 1024 samples multiple times, the ratio for gzip is between 0.58 and 0.6 without outliers, thus all 100 tries of the experiment come to nearly the same conclusion of the mean compression ratio.

While this experiment used results where characteristics are measured on individual file level (full mode in SFS), the overall convergence behavior applies also to the partial mode of SFS. The reason is that logically partial mode is similar to randomly select used data blocks on the file system and measure their compression characteristics.

5 Evaluation

SFS is run on two cluster systems: The WR cluster is a small research cluster with 50 nodes. The supercomputer Mistral at DKRZ is equipped with 3000 nodes and 52 Petabyte of storage[14]. As the evaluation needs root privileges to allow scanning of all files, the access to Mistral was limited. In the case of DKRZ, the full file scan strategy was applied, while for WR the partial mode is explored. LZbench is configured to run all supported compressors with variants of compression levels. This lead to a configuration with 162 algorithms. We will focus on the results on the WR cluster, then compare them to DKRZ results.

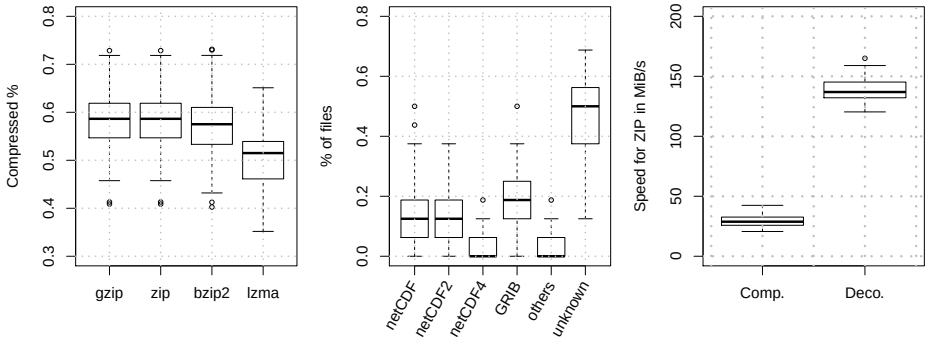
5.1 Test Environment

In both cases a single compute nodes is used for SFS: WR: A quad socket AMD Opteron 6344 with 2.6 GHz and 12 cores, Gigabit Ethernet. 12 threads are run with the scanner using the partial mode and a chunk size of 10 MB. During a three day run, 4403 random samples have been drawn from 3118 files. The samples cover a volume of 38.1 GByte out of the 1.1 TByte, where the selected files occupy 500 GByte.

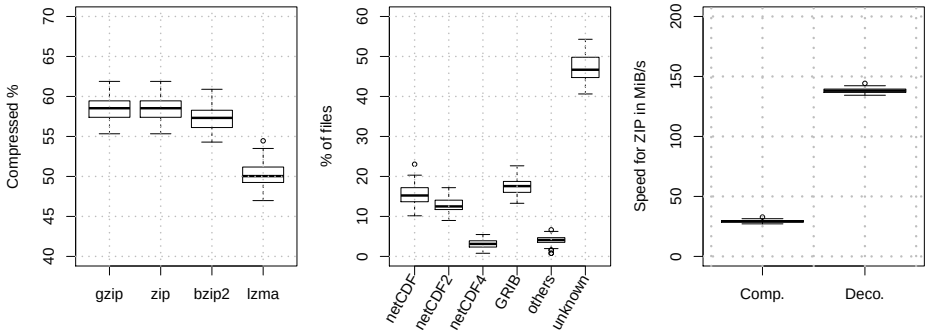
DKRZ: A dual socket Intel Xeon E5-2695V4 with 2.1 Ghz and 18 cores, Infinband FDR. 36 threads are run for the scanner. During a 5 week run, 70 files with a total size of 3 TB have been completely scanned using the described methodology. As we will see the results are still quite comparable.

[14] See `https://www.vi4io.org/hpsl`

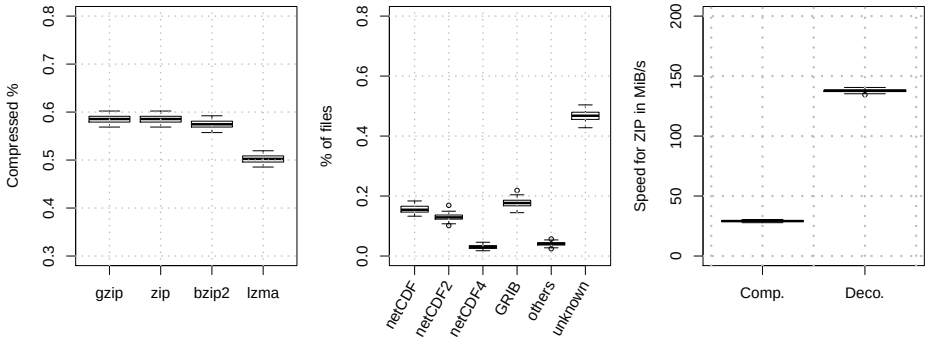
(a) Drawing 16 samples with replacement



(b) Drawing 256 samples with replacement



(c) Drawing 1024 samples with replacement



(d) Drawing 16384 samples with replacement

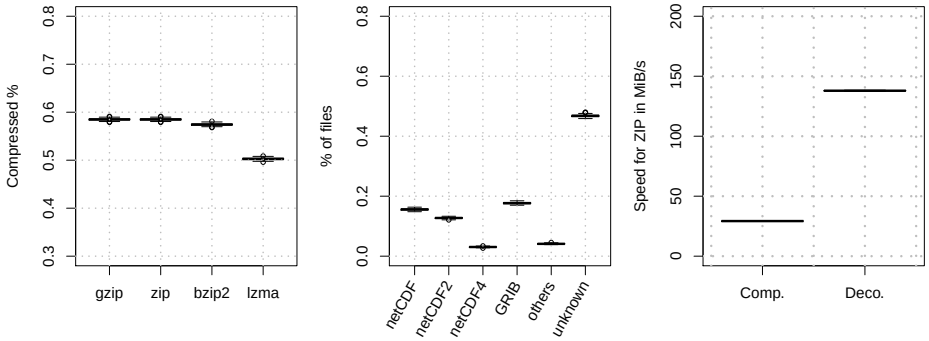


Fig. 1: Simulated convergence behavior of various characteristics

Data pools: The scientific computing (WR) group provides a data pool for several data sets with currently 1.1 TB of data. It consists of three different types of data: 1) A variety of relevant data from the German Electron Synchrotron (DESY) consisting of images and HDF5 files; 2) Directories of data prepared for the student cluster competition of ISC consisting of system files, various scientific applications and their input and output files; 3) Data for the ECOHAM5 ecosystem model. The DKRZ file system offers a space of 52 Petabytes occupied by 300 million files. 80% of the files are small (some KiB), but the capacity is mainly used by scientific files in NetCDF, GRIB and HDF5 format.

5.2 Study of the WR data pool

The characteristics of the algorithms for the complete WR data pool are plotted in Figure 2 [15]. Each point represents the mean characteristics for a single algorithm; the axes show the ratio and compression speed of it and the color encodes the decompression speed as a third dimension.

There are roughly two diagonal lines between ratio 0.5 and ratio 0.6 and between 0.6 to 1.0. As the figure suggests, there is a correlation between ratio and the decompression speeds: The Pearson correlation coefficient between the ratio and the logarithmic compression and decompression speed is 0.72 and 0.76, respectively. That means that starting with the algorithm of the best compression ratio, the performance increases exponentially with a linear decrease in compression ratio. Between both speeds the correlation is weaker (0.61), since some compressors are designed for fast decompression. Similar correlations are obtained when subsetting the data for ratios < 0.6 and > 0.6, indicating that algorithms in these sections follow different strategies.

As the data pool of WR can be easily split into several use cases, we analyze the compression characteristics of each subpool individually. Table 1 and Table 2

[15] Some algorithms failed to compress a fraction of chunks correctly: blosclz, density, lzmat, lz5hc15-1, pithy & tornado06a. Tornado06a-1 and density failed for 37% and 8% of chunks, respectively; others failed below 0.2% of chunks. In the computation of means, erroneous chunks are excluded.

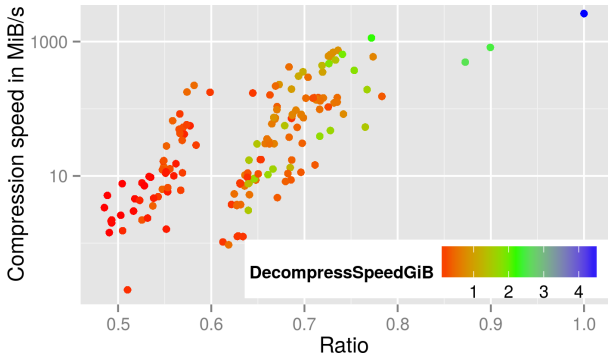


Fig. 2: Comparison of algorithms for the full WR pool

show the characteristics of all algorithms as a heatmap. The algorithm encodes the name, e.g., blosclz, then, separated by dashes, the date of the code version may be added and finally comes the compression level – the higher the level, the more CPU time is spend to increase the compression ratio. Sometimes instead of the date of the code a code version is given. Green is the best value and red the worst. Several interesting aspects of the compressors can be identified: Firstly, the SCC pool behaves differently in respect to compression ratio than the scientific data in DESY and ECO5. DESY and ECO5 show a similar compression ratio. The compression and decompression speed between pools are typically between 0.5 to 2x between the pools. Memcopy serves here as a reference for the achievable throughput with 4-5 GiB/s.

5.3 Identifying Useful Algorithms

Based on the analysis, one could judge that a general purpose algorithm such as LZ4fast is applicable across all subpools. To identify candidate algorithms for global usage, e.g., within file systems, we treat the subpools of the WR data as a single pool.

To reduce the search space for humans, the analysis scripts supports to filter algorithms inferior to another algorithm. Therefore, it checks the combination of all algorithms: An algorithm A is inferior to algorithm B iff A.ratio $\geq$ B.ratio and A.comprSpeed $<$ B.comprSpeed and A.decoSpeed $<$ B.decoSpeed. Applying the script to the complete data set, 70 algorithms remain from the 162 algorithms.

We can identify interesting classes of algorithms: balanced algorithms achieve similar compression and decompression speeds with a acceptable compression ratio, read-optimized algorithms compress slowly but decompress fast and the best algorithms in a category. A selection of these algorithms is shown in Table 3. CSC35-5 and ZLIB17 yield the highest ratio but very low compression ratio. LZ5HCR and LZSSSE2 are read-optimized algorithms with a good ratio while BLOSC yields much worse ratio.

For the DKRZ data, the results have been filtered similarly and are also included in the table. While the mean compression ratio is slightly different to the WR data pool, the ordering of algorithms is still quite similar. Performance is in many cases comparable but at the upper end, LZ4fast and Pithy achieve now a much better performance. The performance of LZ4fast is even higher than the memory throughput (therefore, memcpy is purged by the algorithm). It could actually accelerate computation of memory-bound workloads!

To verify this result under a memory-bound load, one big (full mode) test of 40k files has been conducted on DKRZ running only the Pithy and LZ4fast-17 algorithms. This test revealed a throughput for memcpy of 735 MiB/s and 980 and 1290 MiB/s for LZ4fast (compress/decompress), respectively. In this test, many of the 36 started threads compete for memory bandwidth. Here, the mean ratio for LZ4fast is 0.645, a bit worse compared to the small scan of 70 files.

6 Summary

In this paper, the SFS tool for identifying compression characteristics has been introduced. While relying on an extended LZBench as the vehicle to determine

Algorithm	Ratio			Compr. in MiB/s			Decompr. in MiB/s		
	SCC	DESY	ECO5	SCC	DESY	ECO5	SCC	DESY	ECO5
blosclz2015-11-10-1	0.797	0.978	0.962	824.4	783.7	993.7	2466.6	2531.0	2467.3
blosclz2015-11-10-3	0.755	0.964	0.927	501.5	462.9	652.0	2470.4	2751.3	2610.1
blosclz2015-11-10-6	0.577	0.917	0.850	214.0	164.7	306.0	1220.3	2236.3	2098.0
blosclz2015-11-10-9	0.538	0.848	0.816	160.7	109.6	237.3	782.7	548.1	1457.5
brieflz110	0.523	0.806	0.863	83.9	59.4	114.9	144.6	100.2	193.1
brotli052-0	0.475	0.689	0.745	226.6	138.6	261.2	191.6	128.8	188.2
brotli052-2	0.440	0.657	0.724	101.4	68.3	121.9	180.0	132.4	174.9
brotli052-5	0.419	0.643	0.696	20.8	13.3	22.8	183.4	133.3	197.1
brotli052-8	0.412	0.638	0.694	6.4	4.0	5.9	188.2	135.8	205.9
brotli052-11	0.391	0.596	0.656	0.3	0.2	0.1	143.3	87.5	154.2
crush10-0	0.497	0.740	0.822	7.9	10.2	19.6	149.8	108.1	203.7
crush10-1	0.481	0.721	0.822	5.0	2.7	11.7	158.0	111.4	227.3
crush10-2	0.475	0.709	0.817	1.5	0.7	2.7	160.2	109.2	231.9
csc33-1	0.402	0.577	0.653	9.5	6.3	10.0	22.5	14.2	23.4
csc33-3	0.393	0.553	0.650	6.1	4.2	8.4	21.8	12.9	23.2
csc33-5	0.388	0.552	0.648	3.7	2.9	5.4	22.1	13.0	23.3
density0125beta-1	0.720	0.813	0.863	594.8	580.8	651.5	755.9	687.8	761.7
density0125beta-2	0.580	0.765	0.809	426.1	381.6	629.9	490.7	480.1	608.2
density0125beta-3	0.532	0.725	0.788	168.0	164.3	235.2	148.8	152.7	190.3
fastlz01-1	0.547	0.848	0.821	150.1	110.7	188.5	527.2	469.5	994.4
fastlz01-2	0.539	0.848	0.817	166.4	119.8	208.0	540.7	437.9	1009.5
gipfeli2016-07-13	0.521	0.779	0.807	271.6	171.7	376.6	541.0	364.1	987.5
libdeflate16-08-29-1	0.451	0.663	0.715	65.8	48.9	85.9	149.6	207.9	420.4
libdeflate16-08-29-3	0.445	0.661	0.713	61.1	44.2	84.0	336.0	211.0	424.2
libdeflate16-08-29-6	0.440	0.658	0.711	49.6	35.3	75.0	343.5	214.8	432.2
libdeflate16-08-29-9	0.433	0.645	0.705	12.5	12.0	20.2	338.5	208.7	434.9
libdeflate16-08-29-12	0.430	0.642	0.704	6.3	7.2	6.4	342.1	209.7	448.4
lz4fastr131-3	0.558	0.884	0.823	685.2	578.0	916.9	1843.1	2056.0	2528.9
lz4fastr131-17	0.601	0.907	0.840	1019.5	1172.2	1561.2	2075.8	2381.6	2589.3
lz4hcr131-1	0.514	0.804	0.795	66.1	46.7	80.3	1472.8	1294.8	2294.0
lz4hcr131-4	0.492	0.766	0.788	37.5	23.7	48.7	1553.5	1307.4	2388.7
lz4hcr131-9	0.485	0.756	0.786	19.6	15.0	20.4	1586.1	1335.4	2496.3
lz4hcr131-12	0.485	0.755	0.786	10.1	9.4	7.1	1595.8	1341.0	2519.3
lz4hcr131-16	0.485	0.755	0.785	2.8	3.6	2.4	1600.8	1341.6	2532.1
lz4r131	0.543	0.870	0.814	539.5	403.8	602.9	1765.8	1911.7	2462.1
lz515	0.505	0.801	0.792	292.3	182.9	319.8	1017.8	681.2	1862.7
lz5hc15-1	0.573	0.897	0.816	403.2	321.9	621.6	1606.1	2020.3	2558.9
lz5hc15-4	0.502	0.793	0.783	91.2	58.9	124.6	990.1	729.2	1849.0
lz5hc15-9	0.477	0.765	0.778	3.6	7.0	14.9	810.8	364.8	1307.7
lz5hc15-12	0.461	0.747	0.769	6.2	4.3	12.7	786.8	351.1	1268.6
lz5hc15-15	0.457	0.739	0.769	0.6	1.9	0.8	835.5	331.9	1413.9
lzf36-0	0.561	0.871	0.818	168.2	123.6	223.0	478.2	445.0	871.4
lzf36-1	0.540	0.825	0.817	164.2	120.6	223.8	485.9	437.1	867.5
lzfse2016-08-16	0.445	0.658	0.716	36.3	30.2	47.3	330.0	261.7	463.0
lzg108-1	0.549	0.837	0.812	10.1	20.3	27.5	346.2	307.9	478.9
lzg108-4	0.528	0.826	0.797	9.0	12.7	23.7	351.2	299.6	481.9
lzg108-6	0.516	0.816	0.794	8.0	8.3	20.2	355.2	298.4	486.1
lzg108-8	0.504	0.798	0.790	5.5	3.8	11.6	363.0	315.1	492.6
lzham10-d26-0	0.415	0.601	0.660	4.8	3.9	5.3	100.4	62.8	131.5
lzham10-d26-1	0.398	0.580	0.654	1.8	1.3	1.9	107.9	67.4	133.0
lzjb2010	0.612	0.915	0.892	169.4	132.4	218.5	351.8	306.9	460.1
lzlib17-0	0.421	0.619	0.651	11.0	8.1	13.9	18.7	13.0	23.1
lzlib17-3	0.396	0.579	0.638	3.4	2.0	4.6	19.7	13.8	23.9
lzlib17-6	0.384	0.571	0.632	2.3	1.6	3.4	20.1	14.2	24.3
lzlib17-9	0.383	0.568	0.630	1.5	1.5	1.1	20.1	14.1	24.6
lzma938-0	0.418	0.616	0.674	11.5	8.1	14.8	26.1	17.9	32.8
lzma938-2	0.408	0.609	0.675	9.5	6.4	12.4	27.2	19.2	33.4
lzma938-4	0.403	0.600	0.675	5.6	3.7	7.5	27.8	20.2	34.1
lzma938-5	0.384	0.571	0.632	2.3	1.9	4.3	28.1	20.3	34.5
lzmat101	0.478	0.759	0.798	14.5	8.3	23.9	308.0	282.1	424.2
lzo1209-1	0.545	0.849	0.811	114.5	82.4	137.4	547.6	442.8	1123.0
lzo1209-99	0.520	0.808	0.803	43.8	31.4	57.4	442.5	399.0	721.4
lzo1a209-1	0.538	0.808	0.797	112.0	79.6	135.8	692.9	623.1	1353.3
lzo1a209-99	0.513	0.770	0.788	41.4	29.7	54.2	527.8	537.6	867.5
lzo1b209-1	0.529	0.825	0.799	98.1	65.5	118.3	641.3	581.4	1557.2
lzo1b209-3	0.524	0.811	0.798	94.7	65.8	114.0	640.7	602.8	1531.2
lzo1b209-6	0.515	0.788	0.791	114.5	80.1	144.1	612.6	582.3	1356.1
lzo1b209-9	0.512	0.787	0.790	81.5	59.4	98.0	562.5	542.8	1073.8
lzo1b209-99	0.504	0.782	0.789	39.2	26.3	44.6	566.4	496.3	1215.3
lzo1b209-999	0.480	0.752	0.782	7.5	6.7	7.1	592.7	464.0	1372.7
lzo1c209-1	0.532	0.823	0.800	97.7	68.0	124.9	667.1	658.3	1564.7
lzo1c209-3	0.528	0.809	0.799	96.3	68.2	126.8	664.4	672.0	1524.4
lzo1c209-6	0.517	0.783	0.791	84.4	61.7	107.3	619.9	632.6	1342.3
lzo1c209-9	0.513	0.779	0.790	68.5	50.8	80.4	572.2	594.6	1068.6
lzo1c209-99	0.505	0.772	0.789	35.7	25.1	43.6	579.4	562.4	1222.5
lzo1c209-999	0.485	0.748	0.781	10.0	11.1	8.2	601.2	520.0	1399.0
lzo1f209-1	0.534	0.826	0.801	88.1	59.6	114.6	572.6	522.5	1408.2
lzo1f209-999	0.488	0.768	0.787	8.7	9.1	7.1	496.2	361.2	1101.0
lzo1x209-1	0.546	0.865	0.832	619.3	544.9	1038.2	671.9	903.2	1976.9
lzo1x209-11	0.555	0.876	0.835	725.1	689.7	1281.8	686.9	949.8	2008.8
lzo1x209-12	0.550	0.871	0.833	687.5	629.6	1195.2	676.9	921.5	1986.6
lzo1x209-15	0.547	0.867	0.833	646.2	573.4	1102.5	672.8	908.5	1981.3
lzo1x209-999	0.475	0.741	0.776	4.2	3.1	6.3	500.8	370.2	1227.6

Table 1: Characteristics for the different file pools

Algorithm	Ratio			Compr. in MiB/s			Decompr. in MiB/s		
	SCC	DESY	ECO5	SCC	DESY	ECO5	SCC	DESY	ECO5
lzo1y209-1	0.547	0.868	0.832	616.1	541.8	1035.7	670.8	919.4	1970.8
lzo1y209-999	0.476	0.747	0.777	4.3	3.1	6.3	504.0	388.2	1242.1
lzo1z209-999	0.473	0.742	0.774	4.1	3.0	6.2	486.6	361.1	1195.6
lzo2a209-999	0.507	0.755	0.812	10.4	11.2	9.9	323.9	239.2	598.6
lzrw15-Jul-1991-1	0.593	0.842	0.849	143.1	106.3	182.6	533.6	554.8	694.2
lzrw15-Jul-1991-2	0.587	0.841	0.847	139.8	103.7	178.3	560.5	594.1	673.4
lzrw15-Jul-1991-3	0.574	0.830	0.843	164.7	123.2	211.1	579.0	557.5	718.1
lzrw15-Jul-1991-4	0.566	0.816	0.839	163.7	121.9	209.9	471.3	390.4	529.5
lzrw15-Jul-1991-5	0.548	0.799	0.836	60.0	45.8	68.4	440.8	357.7	484.2
lzsse22016-05-14-1	0.530	0.799	0.838	13.9	12.3	17.1	1504.2	1068.5	1624.7
lzsse22016-05-14-6	0.495	0.743	0.826	7.9	6.8	14.1	1606.9	1125.4	1671.9
lzsse22016-05-14-12	0.495	0.743	0.826	7.7	7.0	14.2	1605.5	1125.5	1665.3
lzsse22016-05-14-16	0.495	0.743	0.826	7.7	7.0	14.2	1613.6	1125.6	1667.0
lzsse42016-05-14-1	0.516	0.775	0.835	13.3	11.4	17.9	1847.4	1452.2	2261.1
lzsse42016-05-14-6	0.498	0.754	0.830	8.9	8.5	14.9	1937.5	1508.3	2248.1
lzsse42016-05-14-12	0.498	0.754	0.830	8.7	8.8	15.1	1918.9	1508.1	2242.3
lzsse42016-05-14-16	0.498	0.754	0.830	8.7	8.8	15.0	1945.5	1506.1	2243.5
lzsse82016-05-14-1	0.514	0.768	0.826	11.0	9.4	14.8	1839.5	1546.1	2520.1
lzsse82016-05-14-6	0.494	0.746	0.820	7.9	7.3	12.9	1922.4	1610.0	2435.1
lzsse82016-05-14-12	0.494	0.746	0.820	7.7	7.6	13.1	1925.2	1609.6	2432.3
lzsse82016-05-14-16	0.494	0.746	0.820	7.7	7.6	13.0	1938.0	1608.3	2426.3
lzvn2016-08-16	0.495	0.789	0.800	33.5	25.9	42.3	556.0	446.4	820.4
memcpy	1.000	1.000	1.000	3840.6	5041.5	4889.2	4192.7	4963.2	4864.5
pithy2011-12-24-0	0.551	0.875	0.828	595.1	438.1	1102.2	1443.1	1323.2	2461.6
pithy2011-12-24-3	0.540	0.857	0.826	537.4	341.6	1026.8	1342.1	1132.9	2305.6
pithy2011-12-24-6	0.527	0.829	0.822	466.4	264.8	857.5	1312.8	972.4	2230.6
pithy2011-12-24-9	0.524	0.823	0.821	406.2	226.4	747.2	1289.6	928.9	2186.2
quicklz150-1	0.536	0.832	0.826	307.1	268.3	398.4	560.6	469.4	586.7
quicklz150-2	0.510	0.790	0.819	116.0	96.8	134.7	463.3	308.2	528.6
quicklz150-3	0.508	0.787	0.817	34.3	26.6	33.7	769.2	588.7	973.0
shrinker01	0.522	0.812	0.794	236.1	157.0	348.6	863.8	966.5	1890.3
slz-zlib100-1	0.536	0.854	0.813	164.0	125.9	212.3	272.1	258.8	330.1
slz-zlib100-2	0.532	0.850	0.809	164.3	124.1	216.2	270.6	249.6	329.0
slz-zlib100-3	0.530	0.849	0.809	163.7	123.4	217.0	270.3	247.3	328.7
snappy113	0.551	0.850	0.817	409.3	284.8	662.5	1045.6	950.4	1701.9
tornado06a-1	0.433	0.885	0.783	209.4	114.5	272.5	305.5	197.2	484.3
tornado06a-2	0.547	0.860	0.873	125.9	86.6	172.6	248.7	186.3	376.3
tornado06a-3	0.454	0.668	0.700	66.0	45.8	86.0	85.4	62.4	114.9
tornado06a-4	0.448	0.661	0.700	49.4	34.7	61.8	86.6	63.9	116.5
tornado06a-5	0.433	0.657	0.696	18.6	12.3	23.0	66.0	44.9	81.2
tornado06a-6	0.431	0.655	0.695	12.3	8.1	14.4	66.5	45.2	81.5
tornado06a-7	0.423	0.649	0.692	7.0	4.8	8.5	67.7	46.1	81.4
tornado06a-10	0.422	0.647	0.691	2.0	1.3	2.5	68.2	46.2	83.0
tornado06a-13	0.412	0.629	0.687	5.2	3.9	8.6	69.2	47.5	82.8
tornado06a-16	0.410	0.619	0.687	2.8	2.1	2.6	70.0	50.1	84.4
ucl-nrv2b103-1	0.506	0.762	0.801	20.3	14.7	25.0	178.4	112.0	237.6
ucl-nrv2b103-6	0.485	0.737	0.791	10.8	5.5	17.3	187.9	112.7	255.2
ucl-nrv2b103-9	0.480	0.747	0.800	2.0	0.8	4.0	184.0	105.9	238.2
ucl-nrv2d103-1	0.506	0.761	0.804	20.4	14.7	25.1	182.4	113.7	254.8
ucl-nrv2d103-6	0.486	0.736	0.795	11.0	5.7	18.0	192.0	113.7	270.9
ucl-nrv2d103-9	0.479	0.737	0.799	2.1	0.8	4.2	186.3	105.2	246.0
ucl-nrv2e103-1	0.506	0.762	0.804	20.4	14.7	25.0	183.5	116.5	255.2
ucl-nrv2e103-6	0.486	0.738	0.795	11.0	5.6	17.8	192.6	116.0	269.1
ucl-nrv2e103-9	0.478	0.739	0.798	2.1	0.9	4.1	187.7	107.1	245.2
wflz2015-09-16	0.571	0.877	0.811	93.1	69.5	155.2	838.4	700.5	1244.4
xpack2016-06-02-1	0.441	0.657	0.704	57.8	41.4	71.6	287.8	204.3	485.9
xpack2016-06-02-6	0.423	0.641	0.698	20.7	13.3	31.4	329.1	224.8	522.9
xpack2016-06-02-9	0.421	0.639	0.697	15.4	9.5	26.1	333.3	227.7	545.1
xz522-0	0.415	0.609	0.674	8.8	5.8	11.1	23.3	16.4	29.3
xz522-3	0.401	0.598	0.674	4.3	2.2	6.0	24.5	17.9	30.0
xz522-6	0.383	0.571	0.631	2.5	1.7	4.1	24.5	17.7	29.6
xz522-9	0.383	0.571	0.631	2.3	1.8	4.3	24.2	17.6	29.3
yalz772015-09-19-1	0.523	0.833	0.798	36.6	25.1	46.0	386.0	382.6	502.7
yalz772015-09-19-4	0.510	0.822	0.794	20.0	14.5	25.5	398.2	350.0	614.2
yalz772015-09-19-8	0.507	0.818	0.792	12.6	9.1	15.8	401.5	340.8	620.8
yalz772015-09-19-12	0.505	0.814	0.791	9.5	7.0	12.0	402.0	334.4	634.2
yappy2014-03-22-1	0.628	0.869	0.868	59.7	45.9	76.3	1385.3	1539.1	2117.0
yappy2014-03-22-10	0.566	0.852	0.840	49.6	43.4	62.5	1675.2	1614.3	2606.0
yappy2014-03-22-100	0.547	0.846	0.835	37.7	38.9	45.9	1792.3	1642.8	2796.3
zlib128-1	0.465	0.671	0.716	34.9	23.4	42.5	200.8	138.5	251.9
zlib128-6	0.443	0.662	0.708	14.9	8.3	25.1	201.6	146.8	256.9
zlib128-9	0.440	0.660	0.707	6.0	5.6	10.6	204.7	147.9	266.1
zling2016-01-10-0	0.438	0.635	0.711	12.0	11.3	19.8	52.1	47.4	88.5
zling2016-01-10-1	0.436	0.634	0.711	11.6	10.8	18.9	52.3	47.5	88.8
zling2016-01-10-2	0.436	0.634	0.710	11.4	10.6	18.7	52.4	47.5	88.4
zling2016-01-10-3	0.435	0.634	0.710	11.2	10.2	18.4	52.4	47.4	88.3
zling2016-01-10-4	0.435	0.633	0.710	11.1	10.0	18.0	52.4	47.4	88.2
zstd100-1	0.464	0.667	0.727	250.3	197.6	263.6	539.6	420.3	693.1
zstd100-2	0.452	0.662	0.720	214.5	149.3	213.5	517.0	386.3	674.7
zstd100-5	0.437	0.646	0.711	84.6	54.6	75.2	503.4	309.5	638.1
zstd100-8	0.428	0.641	0.706	41.9	20.5	40.9	531.6	309.5	650.7
zstd100-11	0.425	0.638	0.704	23.6	9.4	24.3	540.8	296.9	648.2
zstd100-15	0.424	0.636	0.703	8.1	4.9	11.9	553.8	304.6	685.9
zstd100-18	0.415	0.624	0.701	4.3	2.8	8.1	544.8	290.8	652.0
zstd100-22	0.407	0.609	0.694	2.6	1.9	2.6	477.4	203.3	578.8

Table 2: Characteristics for the different file pools (2)

Algorithm	Ratio	Compr MiB/s	Decom. MiB/s
csc33-5	0.485	3.4	16.7
lzlib17-9	0.491	1.4	17.0
xz522-9	0.493	2.1	20.8
lzma938-5	0.493	2.2	24.2
brotli052-11	0.510	0.2	110.6
lzma938-2	0.526	7.9	23.1
zstd100-22	0.526	2.2	294.3
xpack2016-06-02-9	0.548	12.3	282.9
brotli052-5	0.549	16.5	156.6
xpack2016-06-02-6	0.549	16.9	278.9
zstd100-11	0.549	13.8	394.0
zstd100-2	0.574	177.6	455.3
lz4hcr131-16	0.640	3.1	1522.2
lzsse22016-05-14-16	0.640	7.7	1341.6
lz4hcr131-12	0.640	9.4	1519.5
lz4hcr131-9	0.640	17.2	1511.5
lz4hcr131-4	0.649	30.0	1477.8
lz515	0.673	229.2	858.6
density0125beta-2	0.683	419.4	496.5
pithy2011-12-24-9	0.694	305.9	1131.4
lzo1x209-1	0.726	606.7	833.7
lz4r131	0.726	469.8	1893.1
lz4fastr131-3	0.741	646.1	2001.1
lz4fastr131-17	0.772	1132.7	2263.1
blosclz2015-11-10-3	0.872	494.4	2612.6
blosclz2015-11-10-1	0.900	819.4	2496.9
memcpy	1.000	4449.1	4602.0

(a) WR data

Algorithm	Ratio	Compr MiB/s	Decom. MiB/s
lzlib17-9	0.426	1.5	22.0
xz522-9	0.427	2.2	24.3
lzma938-5	0.431	2.9	29.1
lzham10-d26-1	0.445	1.4	113.3
csc33-3	0.445	6.5	23.3
brotli052-11	0.451	0.3	124.5
lzma938-0	0.473	13.0	28.2
zstd080-22	0.476	1.1	260.7
brotli052-5	0.489	18.4	165.6
zstd080-18	0.496	3.9	434.4
xpack2016-06-02-9	0.498	19.3	386.8
xpack2016-06-02-1	0.504	53.5	362.0
zstd080-5	0.511	69.4	560.8
brotli052-2	0.512	126.6	168.7
zstd080-2	0.518	220.9	594.0
zstd080-1	0.523	355.0	633.9
lzo1c209-999	0.566	13.5	939.5
lz5hc15-4	0.574	126.3	1410.1
lz515	0.576	326.9	1934.9
lz4hcr131-16	0.577	3.1	2720.6
lz4hcr131-12	0.577	12.4	2700.8
lz4hcr131-9	0.577	28.4	2670.3
lzo1b209-6	0.578	143.3	992.5
lz4r131	0.599	951.4	3037.4
lz4fastr131-3	0.603	1272.6	3215.6
pithy2011-12-24-3	0.613	1787.5	3535.2
lz4fastr131-17	0.614	1904.8	3610.3

(b) DKRZ data

Table 3: Selected algorithms with good properties (sorted by ratio)

characteristics per file chunks, the contribution of this paper is in the methodology to infer characteristics of the data pool from data samples – a strategy that has been previously analyzed and is briefly recapitulated and demonstrated for compression characteristics in this paper. By analyzing random chunks of large files, SFS is able to scan huge data pools. Showing the features of 162 compression algorithms, some relevant algorithms could be identified. It turned out the compressors characteristics across different data pools of scientific data and systems are comparable. LZ4fast bears a great potential to optimize not only storage capacity but also improve memory throughput for memory bound workloads. Future work is to extend the LZbench tool to coordinate the scanner emulating memory congestion of bulk synchronous HPC applications.

Acknowledgements

I would like to thank the people providing the data used for the study. In particular, the scientists at DESY that provided data from various DESY Photon Science Experiments at PETRA III (`http://petra3.desy.de/`) and DORIS, A. Rothkirch and Beamlines A2, G3, P02.1, P02.2, P03, P06 and P11.

References

1. Sheng Di and Franck Cappello. Fast error-bounded lossy HPC data compression with SZ. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 730–739. IEEE, 2016.
2. Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *science*, 332(6025):60–65, 2011.

3. Nathanel Hübbe and Julian Kunkel. Reducing the HPC-Datastorage Footprint with MAFISC – Multidimensional Adaptive Filtering Improved Scientific data Compression. *Computer Science - Research and Development*, pages 231–239, 05 2013.
4. Michael Kuhn, Konstantinos Chasapis, Manuel Dolz, and Thomas Ludwig. Compression By Default - Reducing Total Cost of Ownership of Storage Systems. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, number 8488 in Lecture Notes in Computer Science, Berlin, Heidelberg, 06 2014. Springer International Publishing.
5. Julian Kunkel. Analyzing Data Properties using Statistical Sampling Techniques – Illustrated on Scientific File Formats and Compression Features. In Michaela Taufer, Bernd Mohr, and Julian Kunkel, editors, *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P3MA, VHPC, WOPSSS*, number 9945 2016 in Lecture Notes in Computer Science, pages 130–141. Springer, 06 2016.
6. Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.
7. Matt Mahoney. Large text compression benchmark. URL: `http://www.mattmahoney.net/text/text.html`, 2009.
8. Senthil Shanmugasundaram and Robert Lourdusamy. A comparative study of text compression algorithms. *International Journal of Wisdom Based Computing*, 1(3):68–76, 2011.
9. Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. In *Data Compression Conference (DCC), 2014*, pages 203–212. IEEE, 2014.