Theory/Practice Transfer Paper

Matriculation number:	8240
Accepted topic:	Effective cold-storage of CI pipeline artifacts
Bachelor's programme, centuria:	Angewandte Informatik, A17a

Contents

1 Introduction

Over the years, the prices for computer storage have decreased [1]. At the same time, consumption has grown significantly [2] [3] [4]. In the meantime, software development has seen a trend towards automated and continuous testing [5] [6] [7].

By looking at an example from PPI AG, it becomes clear that pipelines produce vast amounts of data. A two-hour test suite outputs approximately 5GB of logs and debugging information. The team of roughly 40 employees using these runs about 2.200 pipelines a month which amounts to just under 360GB a day. However, this data is not only produced for static analysis. It should be accessible for extended periods so that potential issues and test failures can be debugged. Given the example, a 4TB drive lasts for less than two weeks (ignoring redundancy and storage speeds requirements).

This presents a significant scalability challenge as multiple teams might require potentially hundreds of terabytes a month, depending on the company's size. For this reason, optimising the storage usage and thus making the best use of the available storage is paramount to achieve efficient scalability. The arguably simplest method to increase storage density is by the use of compression algorithms. Based on that, we can formulate the primary research question of this paper:

> Which compression algorithms are suited for archival of pipeline artefacts?

Furthermore, we shall consider the nature of the data at hand. It may be considered semi-volatile as it is written once and accessed a few times over a period of less than a week[1]. For this reason, the data may be considered read-only until it is eventually deleted. The data access may not be considered sequential as the developer is expected to query HTML reports and log files depending on the type of issue encountered. It is thus randomly accessed. Given that the data is compressed at rest, we may formulate our secondary research question:

> How to ensure random read-only access to compressed data?

To answer these questions, the paper will be split into two parts. In section 2, we will be evaluating the CPU resource usage and compression ratio of different compression algorithms based on a data set collected from the previously mentioned development team. Later on, we will be looking at possible methods to allow random access to compressed data.

[1] Further research regarding the actual storage period is being conducted in parallel.

2 Compression performance

To answer the first research question, we will be outlining a number of different compression algorithms and their relationship with others. Furthermore, we will be running compression performance tests for every algorithm with a context-specific input data set.

2.1 Algorithms

Before we talk about specific algorithms, it is crucial to define some terminology clearly. When talking about a compression tool in general, one may refer to a multitude of terms. Each of these terms represents a layer in the compression stack. At the very bottom is the fundamental compression method or algorithm. While algorithms themselves may be comprised of multiple stages (e.g. dictionary compression and entropy-based encoding), they usually output a single continuous binary blob. On the next layer up, this binary blob may be stored in a container format. Such a container format may define framing and allow additional metadata to be embedded in an archive file. However, it should be noted that compression algorithms do not inherently require a container format. It is very well possible to store the raw compressed data in a file, and some of the tools we will look at are doing precisely this. At the top of the stack is the CLI tool which provides a user-interface to either the container format or algorithm.

We will now be outlining a number of algorithms. Each algorithm will be used through its respective CLI tool, and it may or may not use a container format (which will be noted).

Brotli: The first algorithm is called Brotli. It has been published as RFC 7392 by Jyrki Alakuijala, Zoltán Szabadka and relies on a combination of LZ77 based dictionary compression and Huffman coding [8] [9]. We will be using version 1.0.9 of the CLI tool published by Google Inc. on GitHub. The RFC defines a container format, and thus the resulting data is framed.

Bzip2: In contrast to Brotli, Bzip2 is not based on the fundamental principles of LZ77. Instead, it relies on a Burrows-Wheeler transform and a secondary Huffman coding stage. Unfortunately, few specification documents are available, and most information available is based on a reverse-engineered specification by Joe Tsai [10]. However, for evaluation, we will be using the official CLI with version 1.0.6, which also outputs a container format.

Gzip: Next up is Gzip which relies on the DEFLATE algorithm defined in RFC 1951 [11]. This algorithm relies on LZSS, which itself is a derivate of LZ77 [12], and uses an additional Huffman encoding. Gzip defines a specific container format in RFC 1952, which is used with the CLI tool [13]. We are using Apple gzip 321.40.3.

LZ4: This algorithm, called LZ4, is a close derivate of LZ77 with no additional entropy stage. A

streaming container format is available and used by the CLI. The version used for evaluation is 1.9.3 by Yann Collet.

LZMA: Another algorithm that closely follows LZ77 is LZMA [14]. It should be noted that a different version called LZMA2 is available. However, the underlying compression is the same. The extension adds support for uncompressed sub-chunks. This is used by the xz container format, but for our evaluations, we will use the algorithm directly through the liblzma 5.2.5 library.

LZOP: Yet another derivate of LZ77 is Lempel–Ziv–Oberhumer. It provides both a CLI and container format, and we are using version 1.04 for testing.

PIXZ: A closely related algorithm to LZMA. It is a high-level wrapper of the XZ container format and by extent LZMA2. It allows multi-threaded compression and decompression as well as random access[2]. According to the specification, it is the same container format as XZ but stores additional metadata in trailing blocks which are ignored by the normal XZ decompressor. We are using version 1.0.7 of the CLI tool.

Snappy: Derived from the LZ77 dictionary compression, this algorithm is optimised for speed over compression ratio. According to public documentation, it has been developed with the intent of compressing web traffic in-flight. We are using version 1.0.0 of the szip CLI tool.

ZIP: This ubiquitous CLI tool uses the same DEFLATE algorithm as Gzip but defines a different framing format. We are using version 3.0.

Zstd: Defined by RFC 8478, this algorithm is based on LZ77 like many others [15]. However, it extends it by a Huffman coding for literals and finite-state entropy stage for sequences [16].

2.2 Evaluation

The algorithms have been evaluated on a 16" MacBook Pro[3] with 16GB of RAM and an Intel Core i9-9880H running macOS 11.2 (20D5029f). The results of a synthetic benchmark on the machine along with additional hardware specifications can be found under the following URL: https://browser.geekbench.com/v5/cpu/5846065. Each algorithm was restricted to a single core using appropriate command-line flags where relevant. The exact commands and data evaluation tools can be found in the Appendix and accompanying GitHub repository[4] respectively.

Another relevant variable is the input data. Since this paper explicitly focuses on the topic of CI

[2]Using the same method, we will analyse in more detail further down the line.
[3]The exact identifier is MacBookPro16,1
[4]To maintain author anonymity, the link has been excluded in this version of the document

pipelines, an example taken from such a pipeline will be used[5]. The artefact directory comes from a project at PPI AG and is approximately 226 Megabytes in size[6]. It contains 6721 files of varying sizes with the file type frequencies listed in [1].

Table 1: Mime types of files in input data.

Count	Mime type
3795	text/plain
1715	application/json
336	inode/x-empty[7]
355	text/xml
290	application/octet-stream
215	text/html
6	application/csv
3	application/vnd.ms-fontobject
3	font/sfnt
3	image/svg+xml

Since not every algorithm evaluated can combine multiple files into a single compressed archive, all files have been archived using the TAR format before compression. The archive is 232 Megabytes in size due to the added file metadata[8].

[5]The exact data set is not provided to allow unrestricted publication of this paper. If you require access, please contact the author.

[6]Note that this is the raw data size not taking block alignment and losses due to the storage medium into account

[7]Empty inodes are used for lock files and for files that act as status flags (in addition to ones that are empty by coincidence)

[8]Each file in a TAR archive has an associated 512-byte header

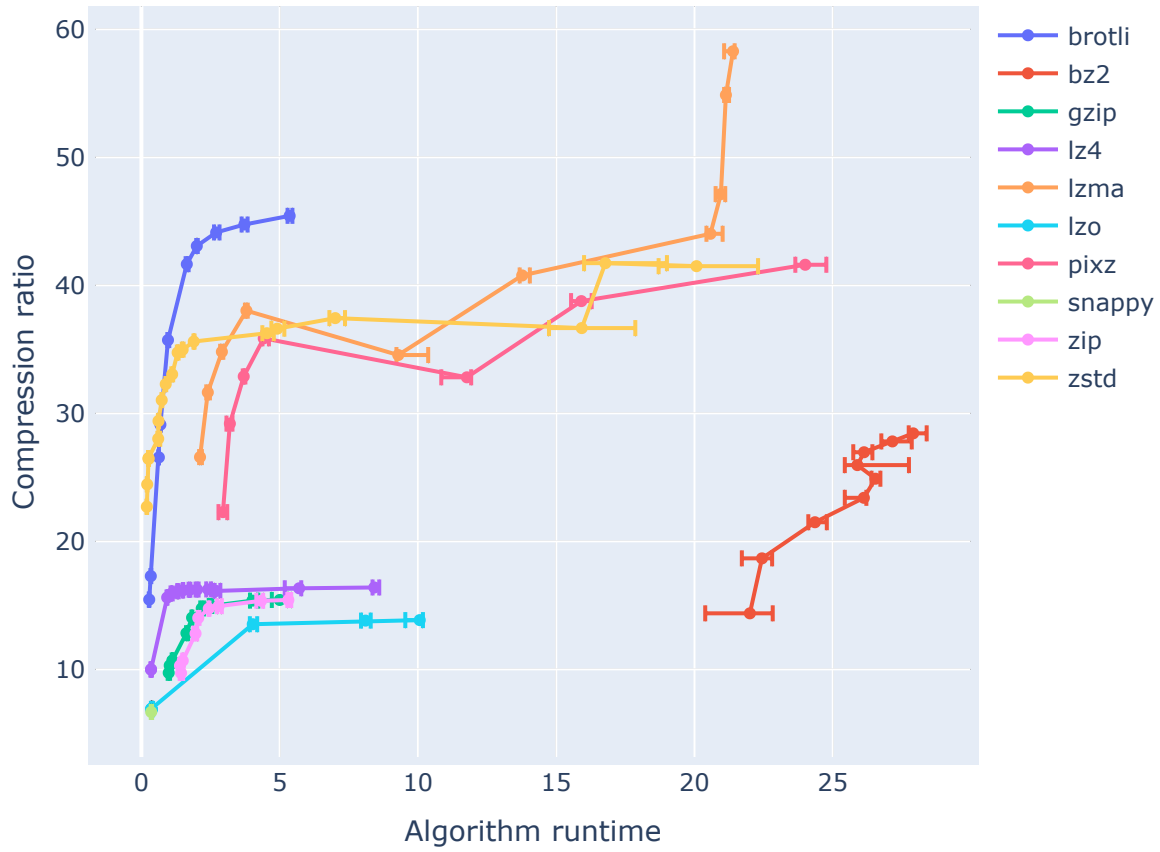

Figure 1: Algorithm efficiency

Each algorithm is plotted in figure 1 by its configuration levels with the runtime plotted on the X-axis (lower is better) and the compression ratio of the resultant archive on the Y-axis (higher is better). Since an increase of compression level uniformly caused a rise in runtime, the data points are not labelled further[9]. Each data point represents the median of all runs with minimum and maximum measurements expressed through error bars. All algorithms behaved deterministically, and thus no vertical error bars are present.

Snappy: Since this algorithm did not have configurable compression levels, it is only represented by a single data point in the bottom left of the graph. As claimed by the algorithm's description, it performs exceptionally fast at the expense of compression ratio with the lowest of all at approximately 6x compression. While this is not useful for archival of pipelines, it is favourable for transport compression, and thus the algorithm achieves its intended goal. However, it should be noted that other algorithms like LZ4, Brotli, or Zstd achieve significantly higher compression ratios at approximately the same computational expense.

LZO: At its lowest level, this algorithm operates comparably to Snappy[10]. At higher levels, it becomes comparable to Deflate, albeit using more CPU resources and not quite reaching the maximum

[9]Lowest algorithm runtime equals lowest compression level and the data points are connected in ascending order

[10]It should be noted that decompression speeds are not evaluated and it is possible that their respective decompression speeds diverge

compression ratio of Deflate.

Deflate: Both gzip and zip use the Deflate algorithm (see section **??**) and thus, as expected, they perform roughly equal with gzip being marginally faster. While these two exhibit relatively low compression ratios, they perform reasonably fast even at high compression levels. Although the gain beyond compression level 6 is negligible.

LZ4: Even though LZ4 does not use an entropy coding stage[11], it outperforms Deflate slightly in terms of runtime and compression ratio at all levels.

Brotli: Of all algorithms, Brotli has the best scaling in relation to the configured compression level. The lower levels provide significant gains in compression ratio at marginal runtime increase while the upper levels asymptotically approach a maximum compression level at the increasing cost of computing resources.

LZMA: LZMA and pixz[12] start with higher computational complexity than Zstd or Brotli but at the same time provide a higher compression ratio at low levels. At levels higher than 4, the increase in compression ratio becomes marginal in relation to the runtime. However, LZMA ends off with a sharp rise in compression ratio at the four highest levels. Ultimately, it surpasses even Brotli and provides the highest compression ratio of all tested algorithms. In comparison, pixz always lacks behind LZMA in terms of compression ratio[13] and does not exhibit the same sharp rise at the extreme compression levels.

Zstandard: At low levels, Zstd compresses faster and with a higher compression ratio than Brotli. However, at levels higher than seven it falls behind and uses significantly more computing resources. Notably, it exhibits a drop in compression ratio at level 14 for unknown reasons.

All algorithms we covered so far are based on a dictionary compression stage which evolved from the work of Lempel and Ziv [17] and some combined it with additional entropy matching (mostly Huffman or FSE). However, one algorithm is left, and that is BZip2. In contrast to the others, it does not use the dictionary method and instead relies on a combination of Run-length encoding, and Burrow-Wheeler transforms along with Huffman encoding [10].

BZip2: The different approach to compression that this algorithm takes is clearly reflected in the graph. It takes a considerably larger amount of time than all other algorithms (only pixz at the highest level is comparable to the lowest level of BZip). At the time of release (1996), it achieved significantly higher compression ratios than other LZ77 based algorithms from that time (namely LZO, LZ4, and Deflate). However, shortly after in 1998, LZMA was published, which achieved significantly higher ratios at lower ratios (as seen in figure 1) [14].

[11] This could potentially be the reason why it performs better on this particular input — further research is required

[12] pixz uses LZMA under the hood

[13] This is likely due to the added container metadata required and the block-based compression. See section 3.2 for more detail.

3 Random access

While general compression may improve the efficiency of data at rest, it complicates access to this data. With most common compression algorithms, all preceding bytes in a file up to a given index we want to access have to be decompressed. Especially in a continuous integration environment, this is problematic as artefacts are usually bundled per pipeline. Since pipelines usually generate HTML based reports with external resources like JavaScript or CSS, multiple accesses to a single archive are the norm. In this scenario, the ability to randomly access compressed data is beneficial. It allows for efficient storage on-disk while still permitting fast access.

In the following sections, we will be going over the fundamentals of how LZ77 based compression algorithms work, evaluate potential methods of enabling random access and analyse the drawbacks concerning compression ratio.

3.1 Fundamentals of LZ77

Before we begin with any evaluation of random access abilities, it is mandatory to talk about the possible ways random access can be implemented. For that, we have to look at the underlying concept of most compression algorithms. Out of all algorithms listed earlier, only two are not relying on the fundamental concept presented by Lempel and Ziv in 1977. For this reason, we will argue with the basic concepts of the LZ77 algorithm proposed in 1977. It is expected that the concepts hold for all other algorithms that are based on it.

The core principle of LZ77 is reference lookups. In its most basic form, the algorithm walks over a string and compares what it encounters with what it has encountered previously. To effectively do so, it keeps a buffer of the previously seen characters. With the original and most derivations, this lookup buffer size can be configured or even dynamically altered on the fly[14]. Regardless, it is only a setting that is relevant during compression. When decompressing a file compressed with LZ77, one only needs to resolve the references inserted earlier and instead insert the referenced bytes. [17]

To gain an easier understanding of the concept, we will consider an example. Note that all examples have been taken from a compression algorithm writeup by Phillip Cutter. While it is not a scientific source in and of itself (for that one should reference the original papers for the relevant algorithms like source 17), it is a highly recommended read for a further introduction into the topic [18].

> Original: Hello everyone! Hello world!
>
> Encoded: Hello everyone! <16,6>world!

[14] It should be noted that this buffer is almost always constrained in size. An unbounded buffer not only consumes large amounts of memory but also slows down the compression as the full buffer is searched through for each byte in the file.

In the example, the algorithm encountered the first character `H` and stored it in the lookup buffer. It did the same for every other character until it encountered the second `H`. At this point, it found the first one at the very beginning of the buffer[15] and carries on comparing the following characters in the string with the following bytes in the buffer to find the longest matching substring. After the space, the substring match ends, and a reference is inserted. This reference can be interpreted as "go back 16 characters, copy six characters".

At a fundamental level, that is all there is to compression with LZ77. While there are more optimisations described in the paper and added on by newer algorithms like Zstd, the core stays the same.

3.2 Block based compression

Now that we know the basic operational principles of LZ77 based compression, we may take a look at the possibility of random access. At first, we will consider a purely theoretical approach. Then, we will look at a more practical implementation and evaluate its performance with the same input data we used previously.

Since we know that LZ77 decompression relies solely on index-based lookups within the file, it should be viable to start decompression of a file at a random offset. Doing so requires frequent seeks for the first set of lookups since we have not yet encountered the referenced text. However, everything that comes after the reference buffer size used to compress the file will be decompressed as usual. This approach comes with one significant drawback: there is no reference of the uncompressed vs compressed byte offsets. Since the data we skipped by jumping ahead has an unknown number of references with unknown sizes, we can not know the byte offset. Without additional metadata, it is impossible to determine the decoded byte offset for a given input byte offset. In our scenario, this may be remedied by storing the byte offsets of each file we compress in an index stored either in-band at the beginning of the file, or out-of-band elsewhere.

Even though this method would work for a rudimentary implementation of LZ77 or LZSS, it is not that trivial when looking at modern derivates. Most APIs do not provide sufficiently low-level access to the decompressor to allow for this decompression method. While it may be considered a long-term goal to integrate this kind of random-access decompression into the public API, it is not feasible short-term.

For this reason, we will be looking at a more practical strategy which uses the same idea from before but in a way that is implementable with current APIs. Instead of decompressing a continuous stream of data starting at an offset, we will be using a block-based method. Instead of encoding the whole input as one long string, it will be cut into blocks of a fixed length. These blocks will then be

[15] The ∈ also causes a lookup, but as it only ever matches a 1-character long string it will not be replaced since the reference is larger than the replaced sequence. An optimisation that strictly speaking is not part of the original algorithm, but LZSS, an extension by Storer et al. [12]

compressed independently and their respective compressed and uncompressed offsets stored in an in-band metadata block. Compared to the previous approach, this is theoretically possible with any compression algorithm and does not require special low-level API features.

As previously mentioned, LZ77 keeps a fixed size, sliding window reference buffer (for simplicity, we will assume it is fixed). This means that at any given point in time during the compression, the algorithm will only ever insert references to data that is at most the reference buffers size away. Based on this knowledge, we can assert that compressing individual blocks instead of one consecutive file will only result in losses at the leading block edges where a previous block's content can not be referenced. For this reason, we expect a direct correlation between the block size and compression ratio with larger blocks, especially those larger than the sliding window size, having a higher ratio. Additionally, it is expected that the effectiveness is highly dependent on the block alignment and repetition of the input data. Future research may focus on analysing the input data to choose the best block size — potentially even variable block sizes — based on the input data. When compressing a set of files, it might even be possible to change the order in which they are compressed to put similar files into the same block. However, this kind of analysis is out-of-scope for this paper.

An additional benefit of a block-based compression is that the process of compressing the data may be parallelised. As each block is compressed independently, it is trivial to scale to as many threads as desired. Compared to regular algorithms, this might pose a significant advantage since many systems are equipped with multiple cores. While the CPU time required to compress a given file stays the same, the wall time required can be reduced by large factors — depending on the exact CPU core configuration.

For this paper, a reference implementation using the Zstd compression algorithm will be provided. This implementation uses the Rust bindings of the algorithm and is available in the accompanying source code repository[16]. It takes a single file as input and outputs a file that contains concatenated independently compressed blocks. To keep it simple, the implementation does not add inline metadata or any framing for the blocks. However, as the block size is fixed and thus the number of blocks can be derived from the input data, we may statically determine the number of bytes this metadata would take if it were stored inline. For our exemplary input file, the number of blocks is calculated in eq. 1 using a block size of 16MB. For simplicity, we will assume that each block requires two 32-bit unsigned integers for the uncompressed and compressed size in bytes[17]. In all upcoming figures, the corresponding block metadata size is calculated and added.

[16] Link omitted in this version of the document

[17] For simplicity we assume that we compress a single file and that multiple files will be combined before compression using another container format.

$$
\begin{aligned}
b_{count} &= \lceil i_{size} / b_{size} \rceil \\
b_{count} &= \lceil 4.916.560.384 / 16.777.216 \rceil \\
b_{count} &= 294
\end{aligned}
\tag{1}
$$

Before getting into the evaluation, we should look at a feature provided by the Zstd API. In addition to the rolling reference buffer based on the original LZ77 specification, it provides the possibility to specify a global dictionary which can be used for lookups. In theory, this should increase the compression ratio — especially in the block-based compression mode where we might otherwise lose some lookups at block borders. However, this mode comes with the drawback that the dictionary has to be known by both the compressor and decompressor. We will include this in the upcoming evaluation.

3.3 Performance evaluation

We will now evaluate the compression ratio of different block sizes, with and without an external dictionary, and at different compression levels. Since we already evaluated the algorithm's general runtime performance, it will not be evaluated this time, and instead, we focus solely on the compression ratio. For evaluation, we consider three algorithm variations:

- **Regular:** Default CLI compression
- **Blocked:** Block based compression
- **Dictionary:** Block based compression with dedicated dictionary

It should be noted that the block metadata size is added to the compressed file size before calculation of the compression ratio. For the dictionary variant, the dictionary size is added as well since it is required for decompression. A line represents each variant in figure 2[18]. The X-Axis shows the compression ratio. The Y-Axis is split into multiple categories for each block size, where each category contains values for each available compression level.

In general, the block-based compression is performing worse than the sequential compression, as expected. However, with increasing block sizes, the margin is getting smaller and smaller. At a size of 4MB, the block-based compression performs similarly at low compression levels. With increasing block size, it approaches the regular compression even at high compression levels. However, dictionary-based compression always lacks behind block-based compression. This may indicate that either the dictionary is not well suited for the data or the overhead of storing the dictionary inline. The latter can potentially be solved by generating a general dictionary which is embedded into the clients and used

[18] Since the number of compression levels is a prime number, it is impossible to cleanly divide the number of labels on the X-Axis and thus increase the font-size.

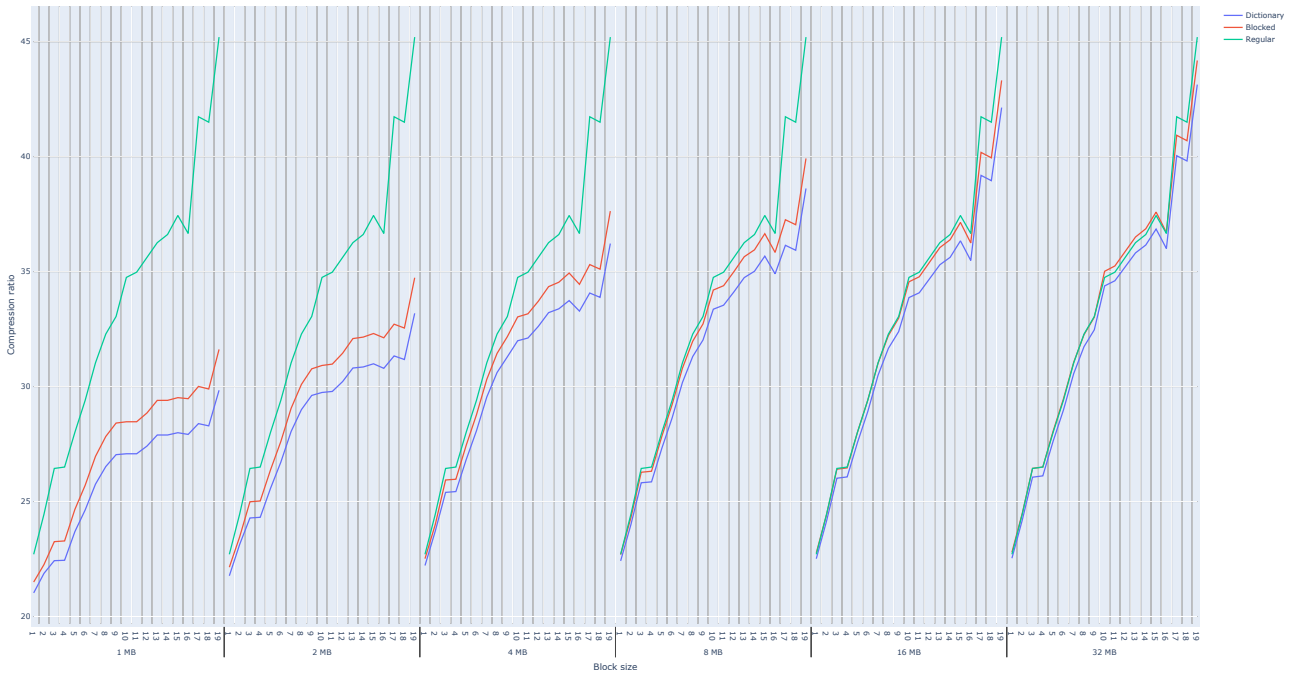

Figure 2: Block size comparison

for all archives. However, it remains questionable whether the dictionary approach is well suited for the data at hand as it performs worse in all observed scenarios. It may be easier to rely on block-based compression without a dictionary as it simplifies the container and decompressor.

Overall, it appears that a block size of 8 MB at compression levels below ten and a block size of 16 MB at compression levels above that is well suited. It provides a reasonable trade-off between the access granularity, and by extent the amount of data that has to be decompressed even though we are not interested in it, and the compression loss.

4 Conclusion

In this research paper, we evaluated the performance of a number of compression algorithms. The results showed that at low compression levels, Zstd is favourable. However, at higher compression levels, Brotli is more efficient and achieves a higher overall compression ratio. For the highest compression rate of all observed algorithms, LZMA is optimal. It should be noted that these results are heavily dependent on the input data used. Caution should be applied when using them for anything other than CI test pipeline artefacts[19].

In addition to the performance analysis, we performed a theoretical evaluation of possible methods to enable random access to compressed data. One of these theories has been turned into a proof-

[19]Even then the contents may differ sufficiently to change the balance of different algorithms. Re-evaluation using the tools provided in the accompanying source code repository is recommended.

of-concept and tested against existing compression algorithms. Surprisingly, only small losses in terms of compression ratio have been encountered, suggesting that efficient random access and thus compression of data at rest is viable.

During the evaluation, a number of variables have not been controlled for various reasons. First, the CPU core frequency was not pinned, and it is possible that it fluctuated. Additionally, the thermal system has caused thermal throttling. While we attempted to minimise the effects by doing warmup runs for each algorithm until the system reached a thermal steady-state, it is impossible to rule out any interference. In the future, it is recommended to run the tests on a system which does not throttle under heavy load[20]. Furthermore, it might be possible that some algorithms have compiled with CPU specific optimisations as each tool was compiled independently by the corresponding distributors. Finally, it should be noted that all compression tests have operated on-disk as opposed to in-memory. While the SSD used has a bandwidth of approximately 3 GB/s each way, which presumably exceeds the compression algorithm speed, it is an uncontrolled variable. Ideally, the data should be loaded into memory first for future tests.

Future research may focus on the effect that different inputs have on the compression algorithms as this has not been covered in this paper. Additionally, we encountered a potentially interesting phenomenon where algorithms with an entropy-based coding stage outperformed those without. Analysing this further may provide more detailed, valuable insights. Another factor which we did not evaluate is the decompression speed of different algorithms. While it is not of significant relevance for this particular scenario, it might serve as a tiebreaker for similarly performing algorithms.

[20] Unfortunately, no such hardware was available for this paper.

References

[1] KOSIS, "Producer price index (ppi) for computer storage devices in south korea from 2010 to 2019," *Statista*, Dec. 2020.

[2] TrendFocus and StorageNewsletter, "Solid-state disk drive (SSD) shipments worldwide from 2013 to 2019, by quarter (in million units)," *Statista*, Jan. 2021.

[3] StorageNewsletter, TrendFocus, The Register, and Forbes, "Worldwide unit shipments of hard disk drives (HDD) from 1976 to 2025 (in millions)," *Statista*, May 2020.

[4] StorageNewsletter, "Hard disk drive (HDD) unit shipments worldwide by market from 2003 to 2025 (in millions)," *Statista*, May 2019.

[5] SauceLabs and Dimensional Research, "Testing trends in 2018: A survey of software professionals," Feb. 2018.

[6] SauceLabs and Dimensional Research, "Testing trends in 2017: A survey of software professionals," Jan. 2017.

[7] Stack Overflow, "2020 developer survey," Feb-2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020.

[8] J. Alakuijala and Z. Szabadka, "Brotli Compressed Data Format," Internet Requests for Comments; Internet Engineering Task Force, RFC 7932, Jul. 2016.

[9] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[10] J. Tsai, "BZip2 Format Specification," 17-Mar-2016. [Online]. Available: https://github.com/dsnet/compress/blob/master/doc/bzip2-format.pdf.

[11] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," Internet Requests for Comments; Internet Engineering Task Force, RFC 1951, May 1996.

[12] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.

[13] P. Deutsch, "GZIP file format specification version 4.3," Internet Requests for Comments; Internet Engineering Task Force, RFC 1952, May 1996.

[14] K. I. Iwata, M. Arimura, and Y. Shima, "An Improvement in Lossless Data Compression via Substring Enumeration," in *2011 10th IEEE/ACIS International Conference on Computer and Information Science*, 2011.

[15] Y. Collet and M. Kucherawy, "Zstandard Compression and the application/zstd Media Type," Internet Requests for Comments; Internet Engineering Task Force, RFC 8478, Oct. 2018.

[16] J. Duda, "Asymmetric numeral systems as close to capacity low state entropy coders," *CoRR*, vol. abs/1311.2540, 2013.

[17] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[18] P. Cutter, "The Hitchhiker's Guide to Compression," Oct-2020. [Online]. Available: https://go-compression.github.io.