Rapidgzip: Parallel Decompression and Seeking in Gzip Files Using Cache Prefetching

Maximilian Knespel
maximilian.knespel@tu-dresden.de
Technische Universität Dresden
Center for Information Services and High Performance Computing
Dresden, Saxony, Germany

Holger Brunst
holger.brunst@tu-dresden.de
Technische Universität Dresden
Center for Information Services and High Performance Computing
Dresden, Saxony, Germany

ABSTRACT

Gzip is a file compression format, which is ubiquitously used. Although a multitude of gzip implementations exist, only pugz can fully utilize current multi-core processor architectures for decompression. Yet, pugz cannot decompress arbitrary gzip files. It requires the decompressed stream to only contain byte values 9–126. In this work, we present a generalization of the parallelization scheme used by pugz that can be reliably applied to arbitrary gzip-compressed data without compromising performance. We show that the requirements on the file contents posed by pugz can be dropped by implementing an architecture based on a cache and a parallelized prefetcher. This architecture can safely handle faulty decompression results, which can appear when threads start decompressing in the middle of a gzip file by using trial and error. Using 128 cores, our implementation reaches 8.7 GB/s decompression bandwidth for gzip-compressed base64-encoded data, a speedup of 55 over the single-threaded GNU gzip, and 5.6 GB/s for the Silesia corpus, a speedup of 33 over GNU gzip.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; *Data compression*; • **Software and its engineering → Software performance**.

KEYWORDS

Gzip; Decompression; Parallel Algorithm; Performance; Random Access

ACM Reference Format:
Maximilian Knespel and Holger Brunst. 2023. Rapidgzip: Parallel Decompression and Seeking in Gzip Files Using Cache Prefetching. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3588195.3592992

1 INTRODUCTION

Gzip [12] and Deflate [11] are compression file formats that are ubiquitously used to compress single files and file agglomerates like TAR archives, databases, and network transmissions. A gzip file contains one or more gzip streams each containing a Deflate stream and additional metadata. Each Deflate stream contains one or more Deflate blocks. Three kinds of Deflate blocks exist: Non-Compressed Blocks, Dynamic Blocks, and Fixed Blocks. Dynamic Blocks, and Fixed Blocks are compressed using a combination of Huffman coding [17] and a variation of the Lempel–Ziv–Storer–Szymanski (LZSS) algorithm [32]. LZSS is a variation of LZ77 [34], which compresses substrings with backward pointers to the preceding stream.

Deflate streams can also be found in other file formats like ZIP [20], PNG [5] and other ZIP-based file formats like JAR [25], XLSX [14] and ODT [31].

1.1 Motivation

Gzip-compressed TAR files can grow to several terabytes, e.g., the updated 1.19 TB large gzip-compressed ImageNet archive [9]. The multiple petabytes large Common Crawl [8] dataset is also distributed as a set of gzip-compressed files. A pipeline for decompressing and preprocessing such data, e.g., for use in machine learning context, would benefit from being accelerated by multi-threaded decompression. Extracting a 1 TB large file with `gzip` takes several hours. Assuming a sufficiently fast file system and two 64-core processors, this can be reduced by a factor of 55 to several minutes when using our implementation.

Gzip files are only slowly getting replaced by newer compression formats like Zstandard [7] because of the wide-spread availability of ZLIB [15] and `gzip` for compressing and decompressing gzip streams. Parallel decompression of gzip becomes desirable when working with gzip-compressed data from external sources because the compression format cannot be chosen. If the compression format can be chosen and if decompression support is not an issue, then newer formats like Zstandard are more efficient. However, as we show in Section 4.9, decompression with our implementation using 128 cores can be twice as fast as the parallel Zstandard decompression tool `pzstd` given the same amount of cores.

1.2 Limitation of State-of-Art Approaches

Although a multitude of implementations for gzip exist, none of them can fully utilize current multi-core processor architectures for decompressing gzip-compressed files without restrictions on the contents. The Blocked GNU Zip Format (BGZF) [4] is a subset of gzip files that adds the encoded size of the gzip stream into the metadata of the gzip stream header. With this metadata, another thread can seek to the next gzip stream and start decoding it in parallel. The command line tool `bgzip` uses multi-threading to

HPDC '23, June 16–23, 2023, Orlando, FL, USA

decompress BGZF files. It cannot parallelize the decompression of gzip files that do not contain the metadata defined by BGZF.

Kerbiriou and Chikhi [21] show that gzip decompression can be split into two stages, each of which can be parallelized. For parallelization, the input data is divided into chunks, which are sent to decompression threads. In the first stage, each thread searches for the first Deflate block in its assigned chunk and starts first-stage decompression from there. Searching for Deflate blocks may result in false positives, i.e., positions in the stream that are indistinguishable from a valid Deflate block without parsing the whole preceding compressed stream. The output of the first-stage decompression may contain markers for data referenced by backward pointers that cannot be resolved without knowing the preceding decompressed data. These markers are resolved in the second stage.

There are some limitations to this approach:

- Parallelization works on a Deflate block granularity. Gzip streams containing fewer Deflate blocks than the degree of parallelization aimed for cannot be effectively parallelized.
- False positives for Deflate blocks cannot be prevented without knowing the preceding Deflate stream.
- There is overhead for the two stages, the intermediate format, which is twice the size of the result, and for finding new blocks. The larger the overhead for finding a block is, the larger the chunk size has to be chosen to achieve a speedup close to ideal linear scaling. This imposes a lower bound on the chunk size for reaching optimal computational performance.
- The memory usage is proportional to the degree of parallelism, the chunk size, and the compression ratio because the decompressed data of each chunk has to be stored in memory.

An implementation of this algorithm exists in the command line tool `pugz`. To reduce false positives when searching for Deflate blocks, it requires that the decompressed data stream only contains bytes with values 9–126 and decompresses to at least 1 KiB and up to 4 MiB. It is not able to decompress arbitrary gzip files.

Furthermore, chunks are distributed to the parallel threads in a fixed uniform manner and therefore can lead to workload imbalances caused by varying compression factors in each chunk. Because it is based on libdeflate [3], it also has the technical limitation that the output buffer size for each chunk has to be configured before decompression. The output buffer is set to 512 MiB of decompressed data for each of the 32 MiB chunks in the compressed data stream, meaning it will fail for files with a compression ratio larger than 16.

1.3 Key Insights and Contributions

Our implementation, `rapidgzip` , addresses the limitations of `pugz` by implementing a cache and a parallelized prefetcher. It is open-source software and available on Github[1]. Our architecture solves the following issues:

- False positives that are found when searching for Deflate blocks are inserted into the cache but are never used and will be evicted. This makes the architecture robust against false positives and enables us to generalize gzip decompression to arbitrary gzip files by removing the non-generalizing Deflate block finder checks.
- The prefetched chunks are pushed into a thread pool for workload balanced parallel decompression.
- Non-sequential access patterns are supported efficiently.
- Gzip files with more than one gzip stream are supported.
- Abstraction of file reading enables support for Python file-like objects.

Caching and prefetching are techniques commonly used in processors to optimize main memory access [30], file access [6], and other domains. We have successfully applied these techniques for chunks of decompressed data. This enables us to provide not only parallelized decompression but also constant-time random access into the decompressed stream given an index containing seek points. While a proof-of-concept for parallel decompression for specialized gzip-compressed data was available with `pugz` [21] and random access into decompressed gzip streams was available with `indexed_gzip` [24], we have successfully combined these two capabilities and made them usable for arbitrary gzip files.

Index for Seeking. An index containing seek points is built during decompression. Each seek point contains the compressed bit offset, the decompressed byte offset, and the last 32 KiB of decompressed data from the previous block. Decompression can be resumed at each seek point without decompressing anything before it. The range between the previous seek point and the requested offset is decompressed in order to reach offsets between seek points. The overhead for this is bounded because the seek point spacing can be configured to a maximum value.

Our implementation's seeking and decompression capabilities can also be used via a library interface to provide a light-weight layer to access the compressed file contents as done by ratarmount [22]. The seek point index can be exported and imported similarly to `indexed_gzip` [24] to avoid the decompression time for the initial decompression pass. Furthermore, decompression can be delegated to an optimized gzip implementation like zlib [15] when the index has been loaded. This is more than twice as fast as the two-stage decompression. Lastly, loading the index also improves load balancing and reduces memory usage because the seek points are equally spaced in the decompressed stream.

Performance Analysis. We also provide benchmark results for all components of our implementation in Section 4 in order to give a performance overview and determine bottlenecks. We improve one such bottleneck, the searching for Deflate blocks, by implementing a skip table. This block finder is 4 times faster than the one implemented in `pugz`. A faster block finder makes it possible to reduce the chunk size and, therefore, the memory requirements accordingly while retaining the same overall performance. The fast Deflate block finder also improves the speed for the recovery of corrupted gzip files.

1.4 Limitations of the Proposed Approach

The main limiting factor of our implementation is memory usage. Each chunk of decompressed data currently being prefetched and contained in the cache is held in memory. The prefetch cache

[1] https://github.com/mxmlnkn/indexed_bzip2

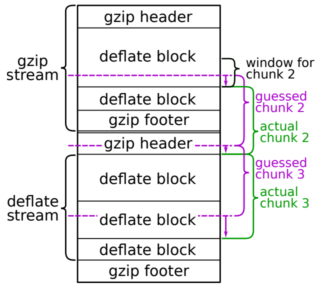

Figure 1: The structure of a gzip file. Also marked are the chunks as they are assigned to the decompression threads and the first Deflate block in each chunk, i.e, the actual chunk start as it will be written to the index.

holds twice as many chunks as the degree of parallelization. The compressed chunk size can be configured and is 4 MiB by default. During index creation, large chunks are split when necessary to ensure that the maximum decompressed chunk size is not larger than the configured chunk size. Therefore, the maximum memory requirement is only 8 MiB per thread if an existing index is used for decompression. The memory required for each chunk depends on the compression ratio of the file when decompressing without an existing index. Compression factors are often below 10, which translates to a memory requirement of 80 MiB per thread. In the worst case, for a file with the largest possible compression factor of 1032, the memory requirement for decompression would be 8.3 GiB per thread. This can be mitigated by implementing a fallback to sequential decompression for chunks with large compression ratios or by splitting the chunk sizes dynamically.

A second limitation is that the achievable parallelization is limited by the number of Non-Compressed Blocks and Dynamic Blocks in the file. If the file consists of a single gzip stream with a single Deflate block, then our implementation cannot parallelize decompression. This is a limitation of the two-stage decompression scheme. In contrast to pugz, the block finder in our implementation does not look for Deflate blocks with Fixed Huffman Codes. If a file consists only of such Deflate blocks, then the decompression will not be parallelized with our implementation. With default compression settings, such blocks occur only rarely, namely for very small files and at the end of a gzip stream. In Section 4.8, we show that most gzip compression tools and compression level settings result in gzip files that can be decompressed in parallel with our implementation.

2 THEORETICAL BACKGROUND

In this section, we summarize the Deflate block format [11] to explain the issue of backward pointers hindering parallelization and to explain our block finder optimizations. We also give a short description of the two-stage Deflate decompression.

2.1 The Deflate Stream Format

Even though the presented parallel decompressor works on gzip files, most implementation details are focused on the Deflate format. The gzip file format [12], shown in Figure 1, consists of one or more gzip streams. A gzip stream wraps a raw Deflate stream and adds

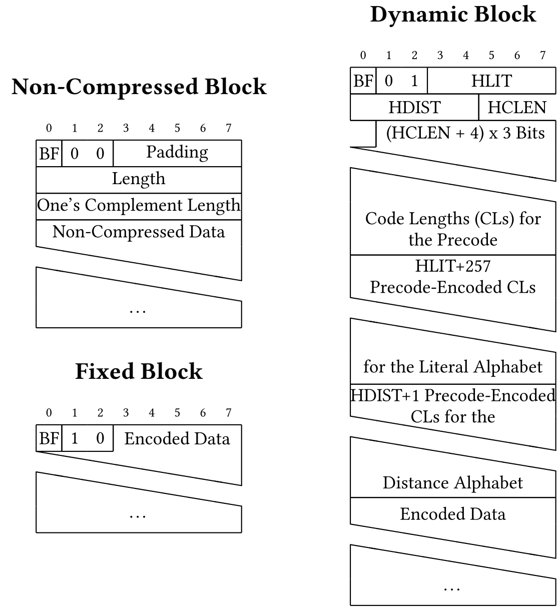

Figure 2: Deflate block formats. A Deflate block can start at any bit offset. Non-Compressed Blocks add padding bits after the block type to byte-align the length fields and the subsequent data. The block is the last one in the Deflate stream if the final-block (BF) bit is set. The block type value 11_2 is reserved.

metadata such as file format identification bytes, the original file name, and a checksum.

Figure 2 shows the three types of Deflate blocks:

- Non-Compressed: Contains a stream of original data. Used for incompressible data.
- Compression with fixed Huffman codes: Contains data compressed with a predefined Huffman code. Used for small data to save space by not storing a custom Huffman code.
- Compression with dynamic Huffman codes: Contains a Huffman code in its header followed by data compressed with it.

These blocks will be referred to in shortened forms as Non-Compressed Block, Fixed Block, and Dynamic Block. Deflate blocks are concatenated to a Deflate stream without any byte-alignment.

Each Dynamic Block uses three different Huffman codes (HC): one is for the Literal alphabet, the second one for the Distance alphabet, and the third one is to encode the code lengths (CL) for defining the first two Huffman codes themselves also referred to as Precode. The size of each HC is stored in HLIT, HDIST, and HCLEN respectively as shown in Figure 2.

The steps for decompressing a Dynamic Block are as follows:

(1) Read the lengths HLIT, HDIST, and HCLEN.
(2) Read the code lengths for the Precode and build a Huffman code from them.
(3) Use the Huffman code defined by the Precode to read the code lengths for the Distance alphabet and the Literal alphabet.
(4) Build a Huffman code from the Distance alphabet code lengths.

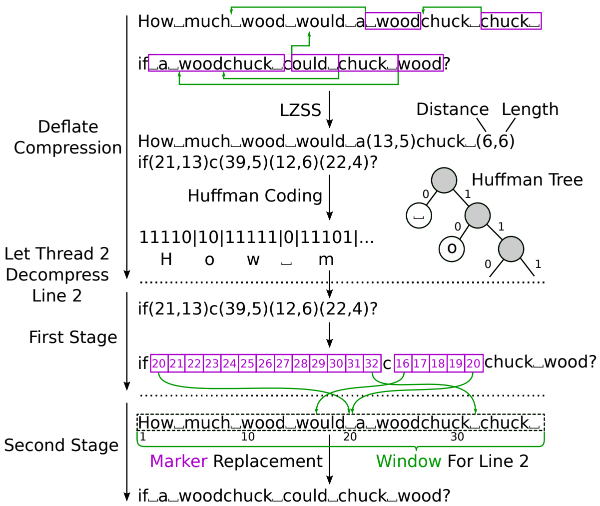

Figure 3: Example of Deflate compression and two-stage decompression.

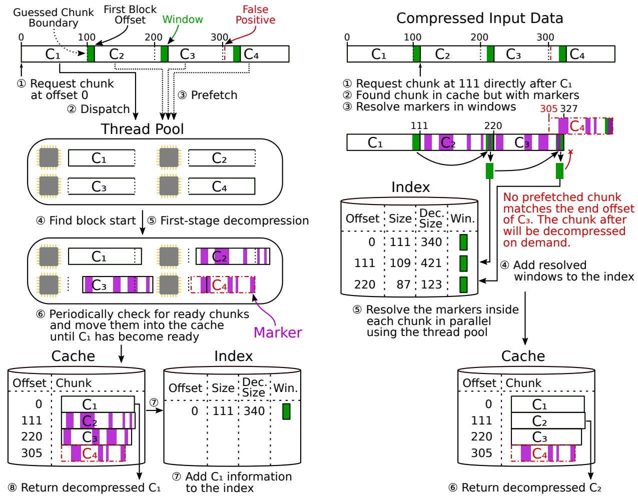

Figure 4: Example of parallel decompression with the cache-and-prefetch architecture. Accessing the first chunk will prefetch further chunks in parallel. On access to the next chunk, the cached result can be used after the markers have been replaced.

(5) Build a Huffman code from the Literal alphabet code lengths.
(6) Decode the Deflate data using the Literal and Distance alphabets.

The compressed data encoded with the Literal alphabet contains either raw literals or pointers to duplicated strings, i.e., instructions to copy a given length of data from a given distance. The distance is limited to 32 KiB of decompressed data, i.e., backward pointers are limited to a sliding window. Small lengths are encoded in the instructions and large lengths require reading more bits from the stream. A distance encoded using the Distance Huffman code follows all backward pointers.

The instructions to copy sequences from the window introduce data dependencies that complicate parallelization.

2.2 Two-Stage Deflate Decoding

The authors of pugz [21] show that it is feasible to parallelize decompression even if there are instructions to copy from an unknown window. This is done by resolving placeholders for unknown values in a second step after those values have become known. An example of this two-stage decompression is shown in Figure 3. The steps for a decompression thread starting at an arbitrary offset in the file are:

(1) Find the next Deflate block. Searching for Deflate blocks is a technique that has previously been used for reconstructing corrupted gzip files [26]. This can be implemented by trying to decompress at the given offset and going to the next offset on error.
(2) Start decoding by filling the as-of-yet unknown window with unique 15-bit wide markers corresponding to the offset in the buffer. The decompression routine has to be adjusted to output an intermediate format with 16-bit symbols to store the markers itself or a literal and an additional bit to signal whether it is a marker or a literal.
(3) After the window has become available, replace all marker symbols with data from the window to get the fully decompressed contents.

The benchmark results in Section 4 show that the marker replacement is a magnitude faster than Deflate decompression. This means that even if only the first stage has been parallelized and the marker replacement is applied serially, then decompression would be up to a magnitude faster. The windows can be propagated even faster because the marker replacement only has to be applied for the last 32 KiB of each chunk. The remaining marker symbols can be replaced in parallel such that each chunk is processed by a different thread. The propagation of the windows cannot be parallelized. Assuming a chunk size of 4 MiB, propagating only the last 32 KiB of each chunk would be 128 times faster than replacing the markers in the whole chunk. However, the non-parallelizable part takes up half the time when using 128 cores. Therefore, the maximum achievable speedup depends on the chunk size and has an upper bound according to Amdahl's law.

3 IMPLEMENTATION

Figure 5 shows the class hierarchy of `rapidgzip`. The main components are:

- A chunk fetcher that can opaquely prefetch and cache chunks in parallel
- A block finder that returns likely Deflate block offsets in a given data stream
- A gzip/deflate decoder that supports two-stage decoding
- A database containing Deflate block offsets and sliding windows to start decoding from

The `FileReader` interface abstracts file access to support not only file access to regular files but also access to Python file-like

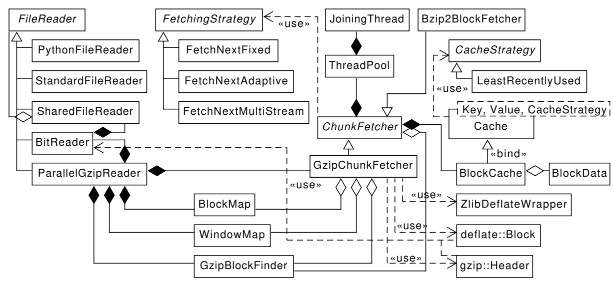

Figure 5: Class diagram of the main components of our implementation.

objects. A Python program can use this feature to implement recursive access to the contents of gzip-compressed gzip files. Our implementation fulfills the following design goals:

- parallel chunk decompression
- only requires an initial decompression pass until the requested offset
- fast concurrent access at two different offsets
- seeking is possible in constant time when the requested offset exists in the index
- building the index is not a preprocessing step, it is done on the fly
- robust against false positives returned by the block finder

Concurrent access at different offsets is common when providing access to gzip-compressed (TAR) files via a user-space filesystem, e.g., provided by `ratarmount` [22]. Therefore, our implementation not only generalizes the approach from `pugz` [21] to non-ASCII contents but also incorporates random access capabilities similar to `indexed_gzip` [24].

Robustness against false positives results from the cache acting as an intermediary with the offset as key. When a thread finds a false positive, then the wrong result will be inserted into the cache with a wrong offset as key. The main thread will request the next chunk based on the end offset of the previous chunk. If the prefetching of the next chunk found a false positive, then the main thread will not be able to find any matching chunk in the cache and has to dispatch a new decompression task for the previous chunk's end offset.

3.1 Seeking and Reading

A seek only updates the internal position and the end-of-file flag. Any further work for seeking will be done on the next read call.

Figure 4 shows an example of parallel decompression when reading consecutively from the start of the file. During a read call to the `ParallelGzipReader` class, the internal `GzipChunkFetcher` is requested to return the chunk that contains the given position. Access to a chunk triggers the prefetcher, even if the chunk has been found in the cache. If the chunk has not been cached, then a decompression task is dispatched to the thread pool on demand. Cached chunks can contain marker symbols. Marker replacement is also dispatched to the thread pool. At this point, all necessary information for a seek point in the index is available and will be inserted. A fully decompressed chunk is returned to the caller after the marker replacement task has finished.

3.2 Prefetching

Prefetching is applied according to a prefetch strategy and is computed on the chunk indexes, not the chunk offsets. The `ChunkFetcher` class combines a thread pool, a cache, a prefetch cache, a prefetching strategy, and a database for converting chunk offsets to and from chunk indexes. The prefetch cache is separate from the cache of accessed chunks to avoid cache pollution of the latter cache caused by the prefetching. For the common case of full file decompression, the access cache size is set to one and will only contain the last accessed chunk. Figure 4 displays the two caches as one for brevity.

The current default prefetching strategy is an ad-hoc algorithm that works for concurrent sequential access to one or more files in a gzip-compressed TAR archive. It is comparable to an exponentially incremented adaptive asynchronous multi-stream prefetcher [16]. The prefetch strategy returns the full degree of prefetch for the initial access so that decompression starts fully parallel. The prefetch strategy does not keep track of previously prefetched chunks. It returns a list of chunk indexes to prefetch based on the last accessed chunk indexes. The prefetcher has to filter out already cached chunks and chunks that are currently being prefetched.

3.3 Chunk Decompression

The `GzipChunkFetcher` class extends the `ChunkFetcher` class with the seek point index. The index contains compressed offsets, uncompressed offsets, and the window for each chunk. The `GzipChunkFetcher` also provides the implementation for the chunk decompression task that is dispatched to the thread pool. If a window exists in the index for a chunk offset, then the decompression task will delegate decompression to zlib. If no window exists, then the custom-written `rapidgzip` gzip decompressor will try to decompress at candidate offsets returned by the block finder until decompression succeeds. The decompressor will stop when a Dynamic Block or Non-Compressed Block that starts at or after the given stop offset has been encountered. This stop condition excludes Fixed Blocks because the block finder does not look for those. If the stop condition does not match the block finder search conditions, then performance will degrade because the prefetched subsequent chunk offset will not be found in the cache and therefore has to be decompressed again.

Rapidgzip implements full gzip parsing consisting of: gzip header parsing, deflate block parsing, and gzip footer parsing. For Deflate parsing, it supplies the `deflate::Block` class, which supports two-stage decompression as well as conventional decompression when a window is given. For Non-Compressed Blocks, it contains a fast path that simply copies the raw data into the result buffer. The Deflate decompressor also keeps track of the last non-resolved marker symbol. Decompression will fall back to the faster conventional decompression if there is no marker symbol in the window. This improves performance when a file contains large Non-Compressed Blocks and for data compressed with only few backward pointers.

3.4 Block Finder

The block finder returns the next Deflate block candidate offset when given an offset from which to start searching. It may return false positives and should but is not required to find all valid Deflate blocks.

False positives cannot be avoided when searching from an arbitrary offset. When compressing a non-compressible gzip-file with gzip, the resulting gzip stream will consist of Non-Compressed Blocks. The verbatim parts of those blocks contain valid Deflate blocks of the gzip file that has been compressed. These can be detected as false positives during parallel decompression.

The block finder is split into specialized Deflate block finders for Dynamic Blocks and Non-Compressed Blocks. These two are combined by finding candidates for both and subsequently returning the result with the lower offset.

3.4.1 Non-Compressed Blocks. The block finder for Non-Compressed Blocks looks for a pair of 16-bit length and 16-bit bit-wise negated length contained in the header as illustrated in Figure 2. The false positive rate for this check by itself is once every 2^{16}B = 64 KiB because the length is byte-aligned and can be of any length including 0, which leaves the 16-bit bit-wise negated length to be checked.

The false positive rate is further reduced by also checking the block type, requiring the final-block bit to be 0, and requiring the padding to achieve byte-alignment to be filled with zeros. Gzip compressed files with non-zero padding were not encountered. Applying the Non-Compressed Block finder on 12 samples of 1 GiB of random data, yields (2040 ± 90) false positives on average, i.e., once every (514 ± 23) KiB specified with one standard deviation.

In comparison to the Dynamic Block finder described in Section 3.4.2, the false positive rate is higher. But, this does not slow down overall performance because decompression of Non-Compressed Blocks consists of a fast `memcpy` followed by a check of the next block header, which is likely to fail for a false positive. Searching for Non-Compressed Blocks is 7× faster than the block finder for Dynamic Blocks, see Table 2 in Section 4.

Care has to be taken when matching the requested bit offset with the block found by the Non-Compressed Block finder. Bit offsets of Non-Compressed Blocks can be ambiguous because the length of the zero-padding is not known. The preceding bits for the block type and the non-final bit are also zero and thus indistinguishable from the zero-padding.

3.4.2 Compressed with Dynamic Huffman Codes. Finding Dynamic Blocks is a bottleneck of the decoder because most Deflate blocks contain dynamic Huffman codes. Checking these for correctness is more computationally intensive than for Non-Compressed Blocks. The steps for checking a Dynamic Block for correctness are in order:

(1) The final-block bit must be 0
(2) The two block-type bits must be 01_2
(3) The five bits containing the number of literal/length codes must not be 30 or 31
(4) The Precode bit triplets must represent a valid and efficient Huffman code
(5) The data decoded with the Precode must not contain invalid backward pointers
(6) The Distance Huffman code must be valid and efficient

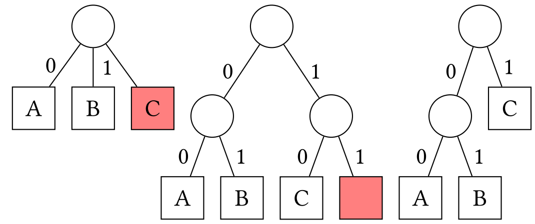

Figure 6: Three Huffman codes with invalid tree nodes highlighted in red. The code lengths for A, B, C are respectively given as (left) 1,1,1, (middle) 2,2,2, (right) 2,2,1. Left: The third symbol of bit length 1 is invalid because only two symbols can be encoded with a single bit. Middle: The code is inefficient because the Huffman code 11 is unused. Right: The code is valid. All available leaf nodes are used.

(7) The Literal Huffman code must be valid and efficient

If any of these steps fail for a given offset, then that offset is filtered and the next offset will be tested.

Huffman codes are stored as a list of code lengths per symbol. Huffman codes will be invalid if there are more symbols with a given code length than the binary tree permits. Huffman codes are not efficient if there are unused leaves in the binary tree. Figure 6 illustrates the requirements on the Huffman code.

Tested bit positions	1		$\times 10^{12}$
Invalid final block	(500 000.1	± 0.7)	$\times 10^6$
Invalid compression type	(375 000.0	± 0.4)	$\times 10^6$
Invalid Precode size	(7812.47	± 0.14)	$\times 10^6$
Invalid Precode code	(77 451.6	± 0.6)	$\times 10^6$
Non-optimal Precode code	(39 256.9	± 0.4)	$\times 10^6$
Invalid Precode-encoded data	(386.66	± 0.05)	$\times 10^6$
Invalid distance code	(14.291	± 0.006)	$\times 10^6$
Non-optimal distance code	(77.126	± 0.016)	$\times 10^6$
Invalid literal code	(340.6	± 1.0)	$\times 10^3$
Non-optimal literal code	(517.2	± 1.4)	$\times 10^3$
Valid Deflate headers	202	± 27	

Table 1: Empirical filter frequencies listed top-down in the order they are checked. To get these frequencies, the Dynamic Block finder has been applied to 1 Tbit + 2300 bit of data. It was chosen such that it can accommodate 1×10^{12} test positions that all have sufficient bits for the maximum possible Deflate header size. This simulation has been repeated 12 times. The uncertainties are specified with one standard deviation.

Table 1 shows empirical results for the amount of filtering for each of these checks. It shows the importance of filtering as early as possible.

A lookup cache is used to speed up the first three checks. Currently, this cache looks up a 14-bit value and returns an 8-bit number representing the offset of a potential Dynamic Block. If 0 is returned, the Precode bits will be checked next.

Seeking inside the bit stream is computationally expensive if it requires refilling the internal buffer. A 14-bit large buffer for the lookup table and a 61-bit large buffer for the Precode are maintained to reduce seeking. The Precode check itself has also been optimized to filter as soon as the symbol length frequencies have been calculated. It uses lookup tables and bit-level parallelism to speed up the computation of the frequency histograms.

Consider this example of adding two independent numbers, each packed into four bits, using only a single 8-bit addition instead of two separate 4-bit additions:

$$0101\ 1101_2 + 0001\ 0001_2 = 0110\ 1110_2$$

The addition of the two 4-bit numbers in the high bits on the left is independent of the low bits if and only if the sum of the two 4-bit numbers in the lower bits does not overflow the range that can be represented with 4 bits.

The frequency histogram uses 5 bits per frequency, which is sufficient to avoid overflows because there are only up to 19 Precode symbols. This is a total of 40 bits because there are 8 different code lengths from 0 to 7. This bit packing also enables the usage of another lookup table for testing the histogram validity. This table takes 20 consecutive bits corresponding to the frequencies for code lengths 1 to 5 as input.

This is followed by a short loop to filter all invalid or non-optimal Huffman codes, see Figure 6. All lookup tables are computed at compile-time by making frequent use of the extended C++17 `constexpr` keyword support. After this lookup table check is passed, the Huffman code structure for the Precode has to be created in a separate step. This is partly duplicated work but it is only done in the unlikely case that the checks succeed. We also found that only 1526 Precode frequency histograms belong to valid Huffman codes. We precalculated this list of valid histograms in an attempt to reduce the histogram validity check to a simple lookup. This did not improve performance.

The Huffman codes for the Deflate literals and distance codes are only initialized after both were found to be valid in order to speed up the preemptive filtering further. This also leads to some duplicate work if both Huffman codes are valid, which, as shown in Table 1, only happens once in 5 billion. See Section 4 for a comparison of the effect of these optimizations.

3.4.3 Compressed with Fixed Huffman Codes. The implemented block finder does not try to find Fixed Blocks because only the final-block bit and the two block type bits can be checked without starting the decompression. Further checks on the length or contents of the decompressed stream could be imposed but those would make assumptions that are not valid for all gzip-compressed files. Fortunately, these types of blocks are rare and in practice only used for very small files or end-of-stream blocks that contain only a few bytes of data. In the worst case of a gzip file only containing Fixed Blocks, this would result in only one thread decompressing the file while the others threads try and fail to find valid blocks.

3.4.4 Blocked GNU Zip Format. The Blocked GNU Zip Format (BGZF) [4] splits the data into fixed-size streams and uses the gzip extra field [12] to store the decompressed stream size. This makes it trivial to gather Deflate block offsets to start parallel decompression from. Furthermore, such start offsets do not require the decoded data

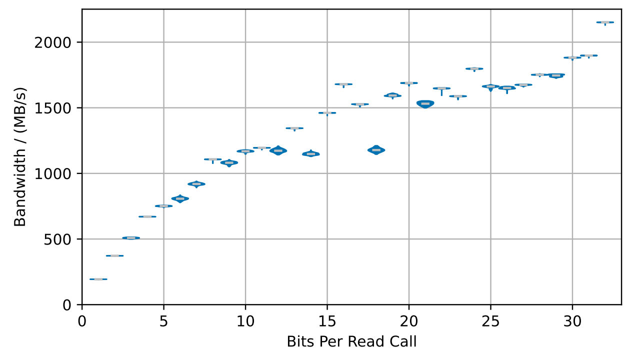

Figure 7: The `BitReader::read` method has been called with the given number of bits in a loop until the end of the data has been reached. The benchmark is single-threaded. The violin plot displays the sampled results from repeating the benchmarks 100 times. The data used for testing measures 2 MiB scaled by the bits-per-read to give approximately equal benchmark runtimes.

from the previous blocks. Because of this, the two-stage decoding can be skipped and parallel BGZF file decompression is trivial. The `GzipChunkFetcher` contains specialized fast paths if a BGZF file has been detected.

4 EVALUATION

In this section, we present single-core benchmarks for the components of our implementation followed by parallel decompression benchmarks. All benchmarks in this section were executed on a single exclusively allocated AMD Rome cluster node provided by the Technische Universität Dresden. Each cluster node contains 512 GiB of main memory and 2 AMD EPYC CPU 7702 processors with 64 cores each and with enabled simultaneous multithreading. This yields a total of 256 virtual cores with a base clock frequency of 2 GHz.

4.1 Bit Reader

Figure 7 shows that the performance of the `BitReader` class increases with the number of requested bits per call. Thus, the bit reader should be queried as rarely as possible with as many bits as possible for optimal performance. The number of requested bits depends on the Huffman decoder and the block finder implementation. One of the implemented Huffman decoders always requests the maximum Huffman code length, which is 15 bits for Deflate.

The parallelized gzip decompressor uses separate `BitReader` instances in each thread. This increases the aggregated bit reader bandwidth accordingly. Comparing the compressed bandwidth in Figure 7 and the decompressed bandwidth in Figure 9 shows that the bit reader is not the main bottleneck even for the worst case with a compression ratio of 1.

4.2 File Reader

Figure 8 shows results for multiple threads reading the same file residing in the in-memory file system `/dev/shm` in parallel. The file

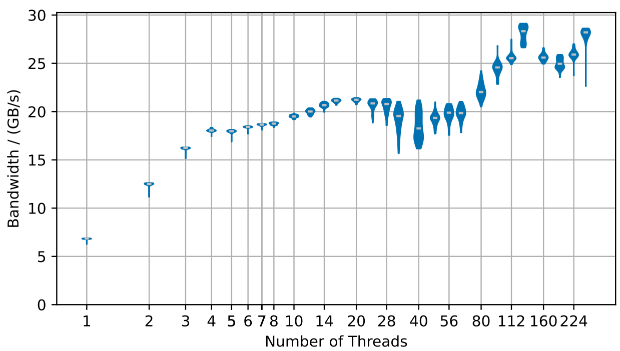

Figure 8: The `SharedFileReader` class is used from a set number of threads to read the file contents of a 1 GiB large file located in `/dev/shm` in a strided manner. The implementation uses POSIX pread to read in parallel from the file. Each thread reads a 128 KiB chunk, skips over the subsequent chunks read by other threads, and then reads the next chunk. The violin plot displays the sampled results from repeating the benchmarks 100 times.

was created by the main thread that has been pinned to core 0. The n-th reader thread is pinned to the n-th core. Each NUMA domain spans 16 physical cores. The first NUMA domain consists of cores 0–15 and their simultaneous multi-threading (SMT) counterparts, cores 128–143.

18 GB/s are reached reliably with 4 threads or more. This bandwidth on the compressed input data is close to the highest decompression bandwidths shown in Figure 9. It follows that file reading starts to become a bottleneck for more than 128 cores. There is a slight decline in the file reading bandwidth for more than 20 threads, which is caused by the fixed work distribution. The work distribution in `rapidgzip` is dynamic to avoid such load balance issues. The bandwidth increases with more than 64 threads because the additional threads reside on the second processor socket.

4.3 Block Finder

Table 2 displays benchmark results for all involved components. Several implementations of the Dynamic Block finder are included in the comparison in order to show the impact of optimizations. The block finder using zlib for a trial-and-error approach is the slowest of all block finders. A custom Deflate parser that returns early on errors is 28× faster. The lookup table implementation described in Section 3.4.2 is 6× faster than the custom Deflate parser. The block finder for Non-Compressed Blocks is 7× faster than the fastest Dynamic Block finder implementation. It is faster because it only needs to check the block type bits and compare the length with the one's complement length stored as a checksum.

The geometric mean of the Dynamic Block finder and the Non-Compressed Block finder bandwidths is 38 MiB/s. This value can be used as a stand-in for the combined block finder. It is 88 % as fast as the Dynamic Block finder and 3.3× faster than the block finder in pugz. A slow block finder can be counter-balanced by increasing the chunk size proportionally. This comes at the cost of increased memory usage. Furthermore, the minimum file size required for full parallelization increases proportionally to the chunk size.

Benchmark	Bandwidth / (MB/s)
DBF zlib	0.1234 ± 0.0003
DBF custom deflate	3.403 ± 0.007
Pugz block finder	11.3 ± 0.7
DBF skip-LUT	18.26 ± 0.03
DBF rapidgzip	43.1 ± 1.1
NBF	301.8 ± 0.5
Marker replacement	1254 ± 6
Write to /dev/shm/	3799 ± 4
Count newlines	9550 ± 5

Table 2: The decompression bandwidths for different implementations of the Dynamic Block finder (DBF) and other components such as the Non-Compressed Block finder (NBF) and marker replacement. The benchmarks have been repeated 100 times. Uncertainties are given with one standard deviation. "DBF custom Deflate" is a trial-and-error method that uses Rapidgzip's custom-written Deflate implementation instead of zlib's.

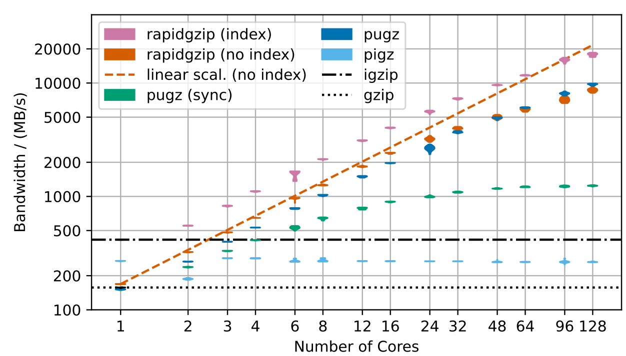

Figure 9: Benchmark results for gzip decompression. The results are written to `/dev/null` to avoid I/O write bottlenecks. It also makes the benchmarks comparable to those by Kerbiriou and Chikhi [21]. The degree of parallelism is enforced with the respective `-P` argument for `rapidgzip` and `-t` for `pugz`. Furthermore, the command line tool `taskset` is used to pin the process to a subset of the available cores. The file size for the base64 data was chosen to be 128 MiB per core for `pugz (sync)`, 512 MiB per core for all other benchmarks of `rapidgzip` and `pugz`, and 1 GiB for the `gzip`, `igzip`, and `pigz` benchmarks. The violin plot displays the sampled results from repeating the benchmarks 20 times.

4.4 Parallel Decompression of Base64-Encoded Random Data

Figure 9 shows the scaling behavior for parallel decompression for pugz and `rapidgzip` using base64-encoded random data compressed with `pigz`. `pigz` was used to speed up benchmark setup times but triggered race conditions in pugz. Such crashes and deadlocks were reduced by setting the `pigz` compression option `--blocksize` to 4 MiB per degree of parallelization. This option adjusts the workload assigned to each compression thread. It does not adjust the Deflate block size, which is 75 kB of compressed data on average. Setting this option presumably reduces crashes because it reduces the number of empty Deflate blocks that are used to byte-align the Deflate streams that have been independently compressed by each thread.

Discussion of the Base64-Encoded Random Test Data. The resulting file has a uniform data compression ratio of 1.315 and contains only a few backward pointers. The compression ratio is mostly achieved with the Huffman coding. The uniform compression ratio minimizes work distribution effects. The low frequency of backward pointers makes it possible to fully resolve the markers in a window after a dozen kilobytes of data. This enables the decoder to replace the two-stage method with single-stage decompression after a while. The custom Deflate decoder is still used but further optimizations might delegate decompression to zlib in this case. Therefore, this test data acts as a benchmark for all components except marker replacement.

Rapidgzip Performance and Influence of the Index. Figure 9 displays the benchmark results for `rapidgzip`. The benchmark is split into first-time decompression and decompression with an existing index file. Decompression with an index file is generally faster because of these reasons:

- marker replacement can be skipped
- the twice as large intermediate 16-bit format can be skipped
- decompression can be delegated to zlib
- the output buffer can be allocated beforehand in a single allocation because the decompressed size is known
- the workload is balanced because chunks in the index are chosen such that the decompressed chunk sizes are similar

For this case, `rapidgzip` is the fastest when using more than one thread. Using 128 cores, `rapidgzip` reaches 8.7 GB/s without an index and 17.8 GB/s with an index.

Comparing Rapidgzip with Pugz. The bandwidth of `rapidgzip` without a preexisting index is higher than `pugz` for fewer than 64 cores. Decompression with `rapidgzip` is assumed to be slower for 64 cores and more because it returns the results in the correct order. This behavior is the same as `pugz (sync)`, which is slower than `rapidgzip` in all cases. For this synchronized write mode, `pugz` does not scale to more than 32 cores. It achieves 1.2 GB/s decompression bandwidth for 48–128 cores. For 128 cores, `rapidgzip` without an index is 7× faster than `pugz (sync)`. The unsynchronized version of `pugz` writes the decompressed data to the output file in undefined order. This version has been benchmarked to reproduce the results in the work by Kerbiriou and Chikhi [21]. The unsynchronized `pugz` and `rapidgzip` do scale to 128 cores but the parallel efficiency decreases.

Comparing Rapidgzip with Single-Threaded Gzip Decompressors. Using 128 cores, `rapidgzip` achieves 8.7 GB/s, a speedup of 55 over decompression with gzip with 157 MB/s, and a speedup of 21 over `igzip`, which decompresses with 416 MB/s.

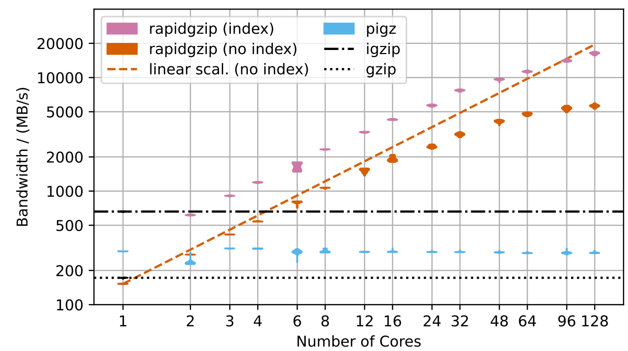

Figure 10: Benchmark results for gzip decompression of a compressed tarball of the Silesia dataset. The benchmark setup is the same as for Figure 9. The file size was scaled by concatenating 2 tarballs of Silesia per core. Compressing it with pigz yields 126 MB of compressed data per core. This is 424 MB of uncompressed data per core with an average compression ratio of 3.1.

When using a single thread, `rapidgzip` reaches 169 MB/s and is slightly faster than `gzip`. It is twice as slow as `igzip` and also slower than `pigz` with 270 MB/s. `igzip` is the fastest single-threaded gzip decompressor. The bandwidth of `pigz` decreases for two cores and then increases again. Pigz never outperforms `igzip`. It is not able to parallelize decompression but it offloads reading, writing, and checksum computation into separate threads.

4.5 Parallel Decompression of the Silesia Corpus

The Silesia corpus [10] is commonly used to benchmark file compression. Figure 10 contains benchmark results for parallel decompression using a gzip-compressed tarball of the Silesia corpus. The Silesia tarball was concatenated repeatedly to itself to scale the file size with the degree of parallelism. The comparison does not include `pugz` because it is not able to decompress data containing bytes outside of the permitted range of 9–126. It quits and returns an error when trying to do so.

`rapidgzip` with an index reaches 16.3 GB/s and 5.6 GB/s without an index. Compared to GNU gzip, which reaches 172 GB/s, this is a speedup of 95 and 33 respectively. In comparison to Figure 9, it stops scaling after ≈ 64 cores. This is because the Silesia corpus contains more duplicate strings than base64-encoded random data. The duplicate strings are compressed as backward pointers, which will be decompressed to markers in the first stage of decompression. If there are no markers after 32 KiB, the two-stage decompression can fall back to single-stage decompression. This is the case for base64-encoded random data but not so for the Silesia corpus. Between the two parallelized stages, the work distributor must resolve the markers in the last 32 KiB of each chunk sequentially. This limits efficient parallelization according to Amdahl's law. Increasing the chunk size can alleviate this theoretically but will also increase the cache pressure.

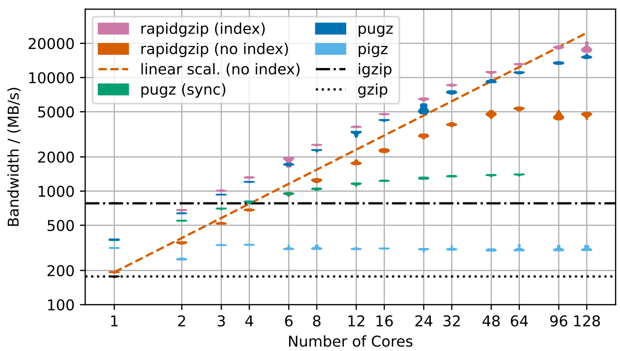

Figure 11: Benchmark results for gzip decompression of a compressed FASTQ file. The benchmark setup is the same as for Figure 9. The file size was scaled by concatenating two uncompressed FASTQ files per core. Compressing it with pigz yields 97 MB of compressed data per core. This is 362 MB of uncompressed data per core with an average compression ratio of 3.74.

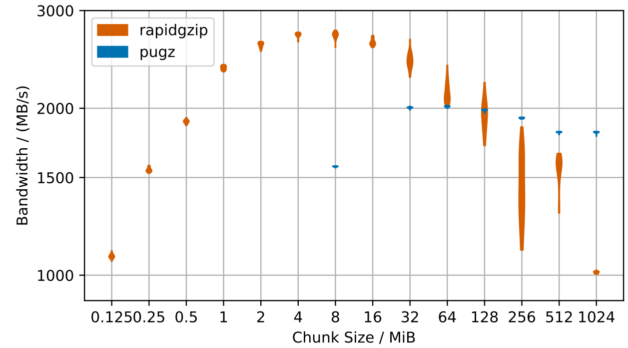

Figure 12: Benchmark results for gzip decompression unhindered by I/O write speeds using 16 cores and 8 GiB of base64-encoded random data resulting in a gzip-compressed file size of 6.08 GiB. The violin plot displays the sampled results from repeating the benchmarks 20 times.

The single-threaded decompression tools in Figure 10 are faster than in Figure 9 because the backward pointers generate decompressed data faster than undoing the expensive Huffman coding. In the best case, such a backward pointer can copy 255 bytes from the window to generate a part of the decompressed stream.

4.6 Parallel Decompression of FASTQ Files

Figure 11 contains benchmark results for parallel decompression on a FASTQ file[2]. FASTQ files are used for storing biological sequences and metadata. They were chosen for the benchmark because pugz was developed to work with them.

Both, rapidgzip with an existing index and pugz without output synchronization scale up to 128 cores while rapidgzip is slightly faster in all cases. Without an index, rapidgzip scales up to 48 cores and then stops scaling at 4.9 GB/s peak decompression bandwidth while pugz with output synchronization scales up to 16 cores reaching a peak decompression bandwidth of 1.4 GB/s. Benchmark results for pugz with synchronization for 96 and 128 cores are missing because pugz reproducibly threw errors when decompressing the test file with that many threads. Overall, the scaling behavior shown in Figure 11 is similar to that shown in Figure 10 with the notable exception that rapidgzip without an index stops scaling for more than 48 cores.

4.7 Influence of the Chunk Size

The chunk size, i.e., the amount of compressed data in a work package submitted to the thread pool is one of the parameters that can be optimized. Figure 12 shows that a very small chunk size leads to performance degradation because the overhead caused by the block finder becomes too large. A very large chunk size leads to performance loss because the work is not evenly distributed to the worker threads. For 512 MiB and more, the performance for pugz stays stable because the maximum chunk size is limited to support even work distribution. In these cases, each thread works on one chunk with a size of 389 MiB. This is not the case for rapidgzip and performance degrades even more because not all worker threads have chunks to work on. Assertion errors were observed with pugz for chunks sized 16 MiB and chunk sizes smaller than 8 MiB. The optimal chunk size for rapidgzip is 4 MiB and 32 MiB for pugz. This is 8× lower than pugz owing to the optimized block finder. A lower chunk size reduces memory consumption.

4.8 Influence of the Compressor

We have shown the weak-scaling behavior using pigz with default compression levels to compress the data in the previous sections. Table 3 shows that rapidgzip achieves efficient parallel decompression on the Silesia dataset for a wide variety of compression tools and compression levels. This means that it can be used for almost any gzip-compressed data making it useful for cases in which the compression options cannot be adjusted.

Table 3 also shows that there is significant variability across different compressors, e.g., decompression of pigz-compressed files is slower than for gzip-compressed files. One cause is the average Dynamic Block size, which can be chosen arbitrarily by the compressor. There is only one Huffman coding per Dynamic Block. The overhead required for reading and preprocessing that Huffman coding can be amortized better for longer Dynamic Blocks. Conversely, larger Dynamic Blocks result in increased overhead for rapidgzip to find the first Deflate block inside a chunk.

The dataset compressed with bgzip -0 can be decompressed the fastest because, in this case, bgzip skips compression altogether and stores the data as Non-Compressed Blocks. Thus, rapidgzip can fall back to a fast memory copy of the Deflate block contents.

The dataset created with igzip -0 contains all the compressed data in a single Dynamic Block. Therefore, the rapidgzip decompression threads cannot find any other Deflate block boundary to start decompression from. In this case, parallel decompression is not possible and the resulting decompression bandwidth is effectively a single-core decompression bandwidth.

[2]http://ftp.sra.ebi.ac.uk/vol1/fastq/SRR224/085/SRR22403185/SRR22403185_2.fastq.gz

Compressor	Compr. Ratio	Bandwidth / (GB/s)
bgzip -l -1	2.99	5.65 ± 0.15
bgzip -l 0	1.00	10.6 ± 0.4
bgzip -l 3	2.81	5.90 ± 0.21
bgzip -l 6	2.99	5.67 ± 0.17
bgzip -l 9	3.01	5.64 ± 0.17
gzip -1	2.74	6.05 ± 0.15
gzip -3	2.90	5.55 ± 0.20
gzip -6	3.11	5.17 ± 0.15
gzip -9	3.13	5.03 ± 0.16
igzip -0	2.42	0.1586 ± 0.0013
igzip -1	2.71	6.15 ± 0.19
igzip -2	2.77	6.42 ± 0.14
igzip -3	2.82	6.52 ± 0.20
pigz -1	2.75	3.82 ± 0.07
pigz -3	2.91	3.81 ± 0.07
pigz -6	3.11	3.76 ± 0.09
pigz -9	3.13	3.73 ± 0.07

Table 3: The decompression bandwidths for decompressing Silesia with `rapidgzip` using 128 cores. The benchmarks were repeated 20 times. Uncertainties are given with one standard deviation. The Silesia dataset was compressed with a variety of compression tools and compression levels as indicated in the first column. The Silesia dataset size was increased by concatenating 256 tarballs of the Silesia dataset. This creates a test file totaling 54.2 GB.

4.9 Comparison With Other Compression Formats

To conclude the benchmark section, we provide benchmark results for other compression formats in Table 4. Only default compression levels were included. For a more comprehensive comparison of compression formats benchmarked with one core, we refer to lzbench [29]. Multiple things are of note in Table 4.

As can be seen for parallelization on 16 cores, both, `bgzip` and `pzstd` require specially prepared gzip and zstd files, respectively, to achieve parallelized decompression. For `pzstd`, Zstandard files with more than one frame are required but `zstd` creates only Zstandard-compressed files with a single frame. Rapidgzip and `lbzip2` [13] can work with arbitrary gzip and bzip2 files respectively.

Single-threaded `rapidgzip` achieves 153 MB decompression bandwidth while `bgzip` is 1.9× as fast and `igzip` even 4.3× as fast. This shows further optimization potential for the custom-written Deflate implementation in `rapidgzip`.

`zstd` and `pzstd` are 5.4× faster than the single-core `rapidgzip` but only 1.2× faster than `igzip` . However, `pzstd` cannot effectively use many cores. The achieved speedup when using 16 cores instead of one core is 8.4. For 128 cores, the achieved speedup is 10.9. This ineffective parallelization closes the gap between `rapidgzip` and `pzstd` for larger core counts. For 128 cores, `rapidgzip` with an existing index becomes twice as fast as `pzstd`.

The command line tool `lbzip2` can be used for parallelized bzip2 decompression. While `pzstd` is 18× faster than `lbzip2` when using one core, it only is 2.1× faster when using 128 cores. Disabling checksum computation in `pzstd` using the `--no-check` option did not improve the decompression bandwidth.

The limited parallel scaling of `pzstd` shows performance potential for the parallel architecture. Alternatively, parallel Zstandard decompression could be implemented in the `rapidgzip` architecture shown in Figure 5 as has already been done for bzip2.

Com.	Rat.	Decompressor	P	Bandw. / (GB/s)
bzip2	3.88	lbzip2	1	0.04492 ± 0.00012
bgzip	2.99	bgzip	1	0.2977 ± 0.0023
gzip	3.11	bgzip	1	0.2965 ± 0.0010
gzip	3.11	rapidgzip	1	0.1527 ± 0.0010
gzip	3.11	rapidgzip (index)	1	0.1528 ± 0.0007
gzip	3.11	igzip	1	0.656 ± 0.009
zstd	3.18	zstd	1	0.820 ± 0.006
zstd	3.18	pzstd	1	0.816 ± 0.005
pzstd	3.17	pzstd	1	0.811 ± 0.003
lz4	2.10	lz4	1	1.337 ± 0.013
bzip2	3.88	lbzip2	16	0.667 ± 0.004
bgzip	2.99	bgzip	16	2.82 ± 0.07
gzip	3.11	bgzip	16	0.3017 ± 0.0007
gzip	3.11	rapidgzip	16	1.86 ± 0.12
gzip	3.11	rapidgzip (index)	16	4.25 ± 0.03
pzstd	3.17	pzstd	16	6.78 ± 0.14
zstd	3.18	pzstd	16	0.882 ± 0.006
bgzip	2.99	bgzip	128	5.5 ± 0.5
bzip2	3.88	lbzip2	128	4.105 ± 0.024
gzip	3.11	rapidgzip	128	5.13 ± 0.13
gzip	3.11	rapidgzip (index)	128	16.43 ± 0.27
pzstd	3.17	pzstd	128	8.8 ± 0.8
pzstd	3.17	pzstd (no check)	128	8.8 ± 0.6

Table 4: The decompression bandwidths for a variety of decompression tools executed with fixed parallelization P. Uncertainties are given with one standard deviation. The first column shows the tool used for creating the compressed file when using default compression levels. The second column shows the compression ratio of the file. The file size was scaled by concatenating 2 tarballs of Silesia per core. This fixes the uncompressed file sizes to 424 MB, 3.39 GB, and 27.13 GB for a parallelization of 1, 16, and 128 respectively.

Software	Version	Software	Version
bgzip	1.17	pigz	2.7
gzip	1.12	pugz	cc7c9942
igzip	9f2b68f0	pzstd	1.5.4
lbzip2	2.5	rapidgzip	d4aa8d4b
lz4	1.9.4	zstd	1.5.4

Table 5: Software versions or git commit hashes used for benchmarks.

5 RELATED WORK

While various parallel tools for compression gzip in parallel exist, no tool for parallel decompression of arbitrary gzip files is known to us.

Parallel Gzip Compression. The `bgzip` tool [4] divides the data to be compressed into chunks, compresses those in parallel as independent gzip streams, adds metadata, and concatenates the gzip streams into a Blocked GNU Zip Format (BGZF) file, which itself is a valid gzip file. Alternatively, `pigz` [2] compresses chunks in parallel as separate Deflate streams instead of gzip streams. It uses workarounds such as empty Non-Compressed Blocks to avoid having to bit-shift the results to fit the bit alignment of the previous Deflate stream when concatenating them. There also exists a multitude of hardware implementations to speed up gzip compression for example for compressing network data [1, 19, 23].

Modified Gzip File Formats Enabling Parallel Decompression. Parallelization of gzip decompression is more difficult because, firstly, the position of each block is only known after decompressing the previous block. Secondly, to decompress a Deflate block, the last 32 KiB of decompressed data of the previous Deflate block have to be known.

Some solutions like BGZF [4] work around these two issues by adjusting the compression to limit the dependency-introducing backward pointers, limiting Huffman code bit alignments, or storing additional Huffman code bit boundaries [28]. BGZF not only contains multiple independent gzip streams but those gzip streams also contain metadata storing the compressed size of each gzip stream to allow skipping over them. Some hardware [27] and GPU [33] implementations speculatively decode Huffman codes ahead in the stream. These implementations make use of the self-synchronizing nature of Huffman codes and reach decompressed data bandwidths of 2.6 GB/s [27].

Two-Stage Decoding. The closest work and the one that this work builds upon is `pugz` [21]. It uses a two-stage decompression scheme, which has been summarized in Section 2.2 and has been implemented in `rapidgzip`.

6 CONCLUSION

We have developed an architecture for parallel decompression and seeking in gzip files based on a cache-and-prefetch architecture. This architecture is robust against false positives returned by the block finder. This robustness enables us to extend the parallel gzip decompression algorithm presented by Kerbiriou and Chikhi [21], which was limited to files containing bytes in the range 9–126, to all kinds of gzip-compressed files. From this new implementation also emerge other improvements such as load balancing and support for multi-part gzip files. The dynamic work distribution also improves upon the observed slowdowns of `pugz` when writing the result to a file in the correct order.

We have implemented this architecture in the command line tool and library `rapidgzip`. We have achieved decompression bandwidths of 8.7 GB/s for base64-encoded random data, and 5.6 GB/s for the Silesia dataset when using 128 cores for parallelization. This is 21× and 8.5× faster, respectively, than `igzip` [18], the fastest single-threaded gzip decompression tool known to us. When compared to GNU gzip version 1.12, it is 55× and 33× faster, respectively.

In the future, we intend to add checksum computation and further optimizations to the custom Deflate implementation. We will also address the limitations mentioned in Section 1.2. In particular, the work splitting of chunks with very high decompression ratios will reduce the maximum memory requirements and further improve load balancing.

ACKNOWLEDGMENTS

The authors are grateful to the Center for Information Services and High Performance Computing [Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH)] at TU Dresden for providing its facilities for high throughput calculations.

REFERENCES

[1] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. 2014. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014* (Bristol, United Kingdom) *(IWOCL '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/2664666.2664670

[2] Mark Adler. 2022. *Pigz: A parallel implementation of gzip for modern multi-processor, multi-core machines.* https://github.com/madler/pigz

[3] Eric Biggers. 2022. *libdeflate: Heavily optimized library for DEFLATE/zlib/gzip compression and decompression.* https://github.com/ebiggers/libdeflate

[4] James K. Bonfield, John Marshall, Petr Danecek, Heng Li, Valeriu Ohan, Andrew Whitwham, Thomas Keane, and Robert M. Davies. 2021. HTSlib: C library for reading/writing high-throughput sequencing data. *GigaScience* 10, 2 (02 2021). https://doi.org/10.1093/gigascience/giab007 arXiv:https://academic.oup.com/gigascience/article-pdf/10/2/giab007/36332285/giab007.pdf giab007.

[5] Thomas Boutell et al. 1997. *PNG (Portable Network Graphics) Specification Version 1.0.* RFC 2083. RFC Editor. https://doi.org/10.17487/RFC2083

[6] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. 2008. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–12. https://doi.org/10.1109/SC.2008.5213604

[7] Yann Collet and Murray Kucherawy. 2021. *Zstandard Compression and the 'application/zstd' Media Type.* RFC 8878. RFC Editor. https://doi.org/10.17487/RFC8878

[8] Common Crawl. 2023. *Common Crawl.* Retrieved 2023-01-15 from https://commoncrawl.org/

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2021. ImageNet21K (Winter 2021 Release). Retrieved 2023-01-15 from https://www.image-net.org/

[10] Sebastian Deorowicz. 2003. *The Silesia corpus.* Retrieved 2023-01-15 from https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia

[11] Peter Deutsch. 1996. *DEFLATE Compressed Data Format Specification Version 1.3.* RFC 1951. RFC Editor. https://doi.org/10.17487/RFC1951

[12] Peter Deutsch. 1996. *GZIP file format specification version 4.3.* RFC 1952. RFC Editor. https://doi.org/10.17487/RFC1952

[13] Laszlo Ersek and Mikolaj Izdebski. 2023. *Parallel bzip2 utility.* https://github.com/kjn/lbzip2

[14] International Organization for Standardization. 2016. *Information technology – Document description and processing languages – Office Open XML File Formats– Part 1: Fundamentals and Markup Language Reference.* Standard. International Organization for Standardization, Geneva, CH.

[15] Jean-loup Gailly and Mark Adler. 2004. Zlib compression library. (2004).

[16] Binny S. Gill and Luis Angel D. Bathen. 2007. AMP: Adaptive Multi-stream Prefetching in a Shared Cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, Vol. 7. 185–198.

[17] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[18] Intel Corporation. 2022. *Intel(R) Intelligent Storage Acceleration Library.* https://github.com/intel/isa-l

[19] Intel Corporation. 2022. Intel(R) QAT: Performance, Scale, and Efficiency. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html

[20] Phillip Katz et al. 2022. APPNOTE.TXT - .ZIP File Format Specification. https://pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.3.10.TXT

[21] Maël Kerbiriou and Rayan Chikhi. 2019. Parallel Decompression of Gzip-Compressed Files and Random Access to DNA Sequences. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 209–217. https://doi.org/10.1109/IPDPSW.2019.00042

[22] Maximilian Knespel. 2022. *Ratarmount: Random Access Tar Mount.* https://github.com/mxmlnkn/ratarmount

[23] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. 2020. High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis. *IEEE Access* 8 (2020), 62207–62217. https://doi.org/10.1109/ACCESS.2020.2984191

[24] Paul McCarthy. 2022. *indexed_gzip: Fast random access of gzip files in Python.* https://github.com/pauldmccarthy/indexed_gzip

[25] Oracle. 2022. JAR File Specification. https://docs.oracle.com/en/java/javase/13/docs/specs/jar/jar.html

[26] Bora Park, Antonio Savoldi, Paolo Gubian, Jungheum Park, Seok Hee Lee, and Sangjin Lee. 2008. Data extraction from damage compressed file for computer forensic purposes. *International Journal of Hybrid Information Technology* 1, 4 (2008), 89–102.

[27] Sudhir Satpathy, Vikram Suresh, Raghavan Kumar, Vinodh Gopal, James Guilford, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Ram Krishnamurthy, Vivek De, and Sanu Mathew. 2019. A 1.4GHz 20.5Gbps GZIP decompression accelerator in 14nm CMOS featuring dual-path out-of-order speculative Huffman decoder and multi-write enabled register file array. In *2019 Symposium on VLSI Circuits*. C238–C239. https://doi.org/10.23919/VLSIC.2019.8777934

[28] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A. Ross. 2016. Massively-Parallel Lossless Data Decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*. 242–247. https://doi.org/10.1109/ICPP.2016.35

[29] Przemysław Skibiński. 2022. lzbench is an in-memory benchmark of open-source LZ77/LZSS/LZMA compressors. https://github.com/inikep/lzbench

[30] Alan Jay Smith. 1982. Cache memories. *ACM Computing Surveys (CSUR)* 14, 3 (1982), 473–530.

[31] OASIS Standard. 2021. Open Document Format for Office Applications (OpenDocument) Version 1.3. https://www.oasis-open.org/standards/

[32] James A. Storer and Thomas G. Szymanski. 1982. Data Compression via Textual Substitution. *J. ACM* 29, 4 (oct 1982), 928–951. https://doi.org/10.1145/322344.322346

[33] André Weißenberger and Bertil Schmidt. 2018. Massively Parallel Huffman Decoding on GPUs. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/3225058.3225076

[34] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714