TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Maximilian Knespel, Holger Brunst
Technische Universität Dresden

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

HPDC' 23

ZIH
Center for Information Services & High Performance Computing

Motivation

Accessing huge datasets, e.g., from academictorrents.com:

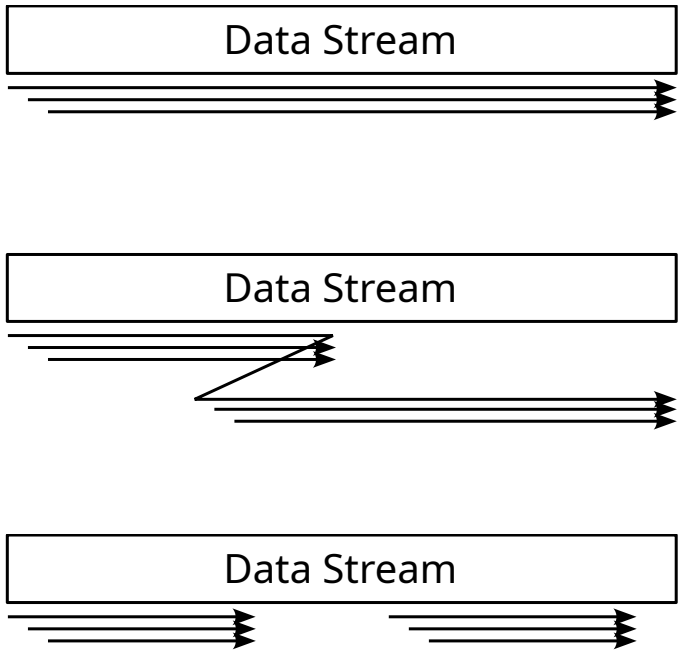
- **wikidata-20220103-all.json.gz:** gzip-compressed JSON, 109 GB, 1.4 TB uncompressed
- **ImageNet21K:** gzip-compressed TAR archive, 1.2 TB, 14 million images averaging 9 KiB.

Solutions:

- ratarmount : **R**andom **a**ccess **TAR mount**.
  Make (huge) archives' contents available via FUSE.
- rapidgzip, indexed_bzip2: Backends for ratarmount for parallel decompression and fast seeking inside compressed gzip and bzip2 files.
  They also offer command line tools for parallelized decompression.

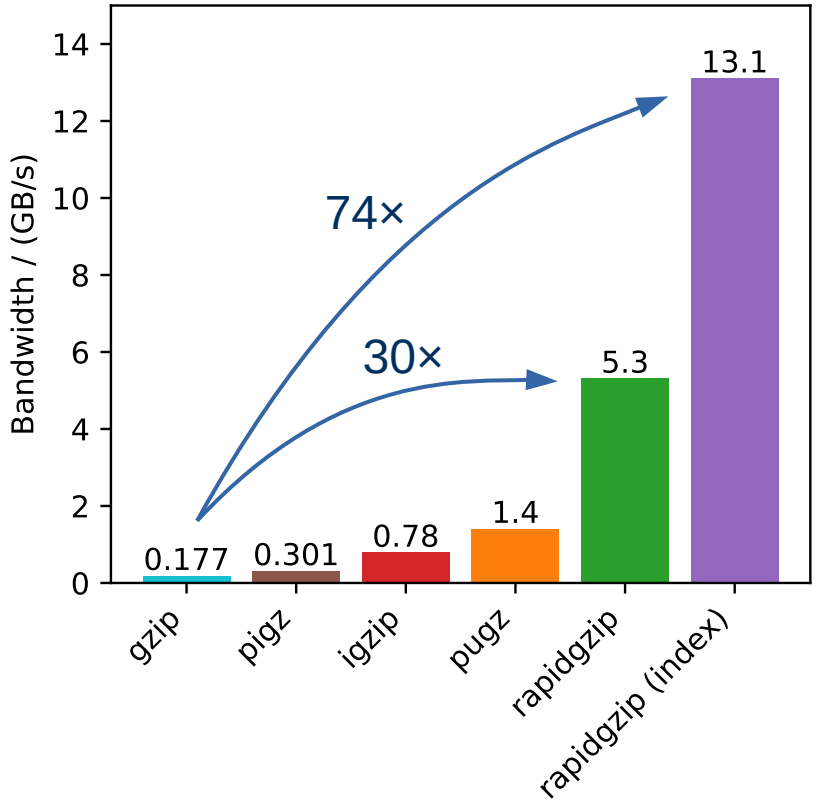
Requirements

- Parallelize gzip decompression
  - Without additional metadata
  - After any seeking
  - For concurrent accesses at two offsets
- Decompress all kinds of gzip files
- Enable fast backward and forward seeking
  - After the index has been created
  - While the index has only been partially created
- Usable as a (Python-)library



Introducing rapidgzip

- rapidgzip: **R**andom **a**ccess **p**arallel (**i**ndexed) **d**ecompression for **gzip** files
- Parallel decompression of gzip files
- Decompression is faster and less memory intensive with an existing index
- The index enables seeking without having to start decompression from the file beginning
- Header-only C++ library with Python bindings:
  `> pip install rapidgzip`
- Also a has a command line interface that can be used as a drop-in replacement for decompression:
  `gzip -d → rapidgzip -d`
- Not for compression
- https://github.com/mxmlnkn/rapidgzip



Decompression benchmarks on a 12 GB FASTQ file using 64 cores of an AMD EPYC 7702 @ 2.0 GHz processor.

Tools Overview

- 1992: `gzip` by Jean-loup Gailly and Mark Adler 

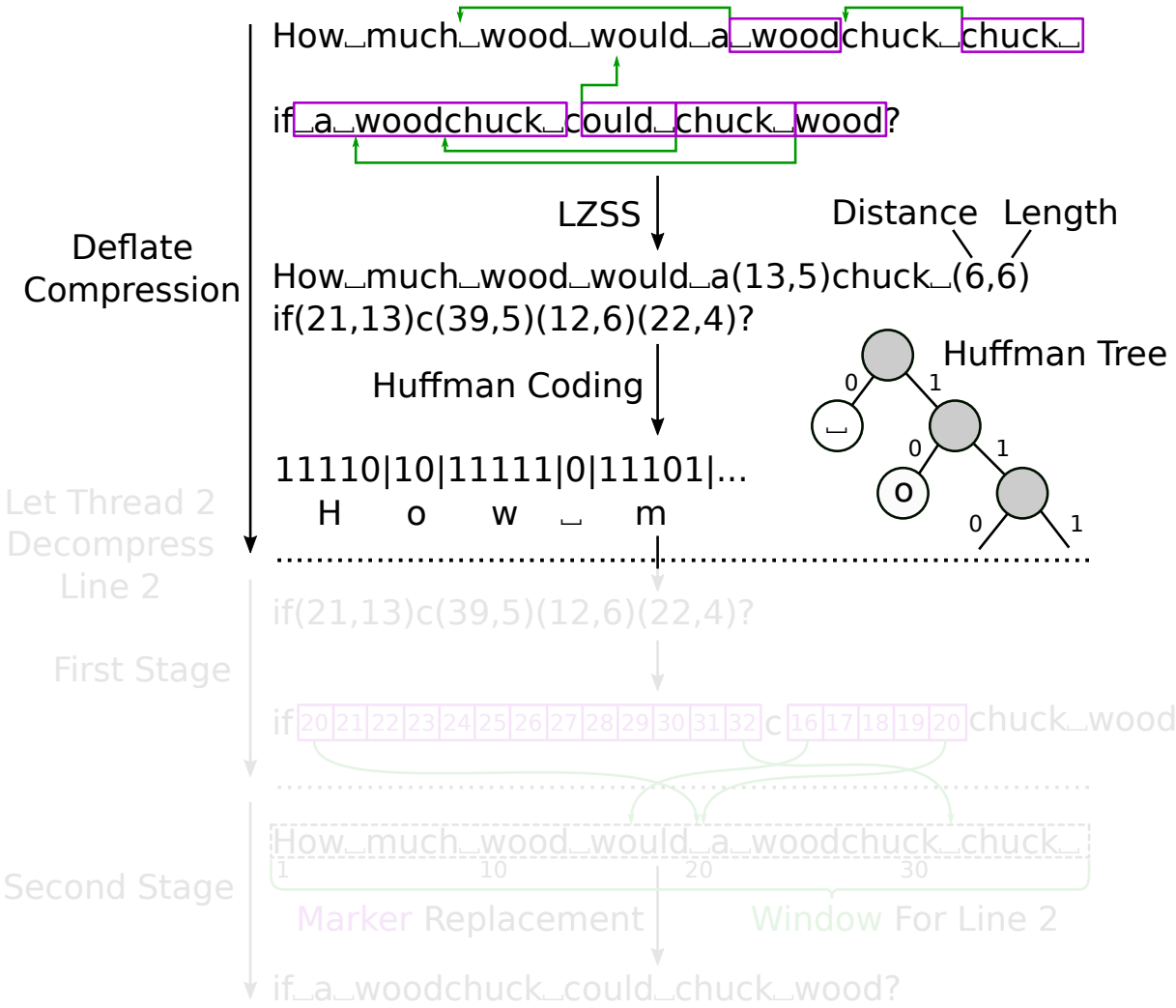
- 2005: zlib/examples/zran.c example by Mark Adler 

  - Shows how to resume decompression in the middle of a gzip stream
- 2007: `pigz` (**p**arallel **i**mplementation of **gz**ip) by Mark Adler 

  - Compresses in parallel
- 2008: Blocked GNU Zip Format (BGZF) and the command line tool `bgzip`, part of HTSlib
  - Compresses in parallel to gzip files with additional metadata
  - Can decompress files containing such metadata in parallel
  - James K. Bonfield and others, "HTSlib: C library for reading/writing high-throughput sequencing data", *GigaScience*, Volume 10, Issue 2, February 2021
- 2016: indexed_gzip: Python module for random access based on zran.c
- 2019: `pugz`
  - Can decompress gzip-compressed files in parallel if it only contains characters 9–126
  - Kerbiriou, Maël, and Rayan Chikhi. "Parallel decompression of gzip-compressed files and random access to DNA sequences." 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2019.

→ What makes parallel decompression so difficult?

Deflate Compression



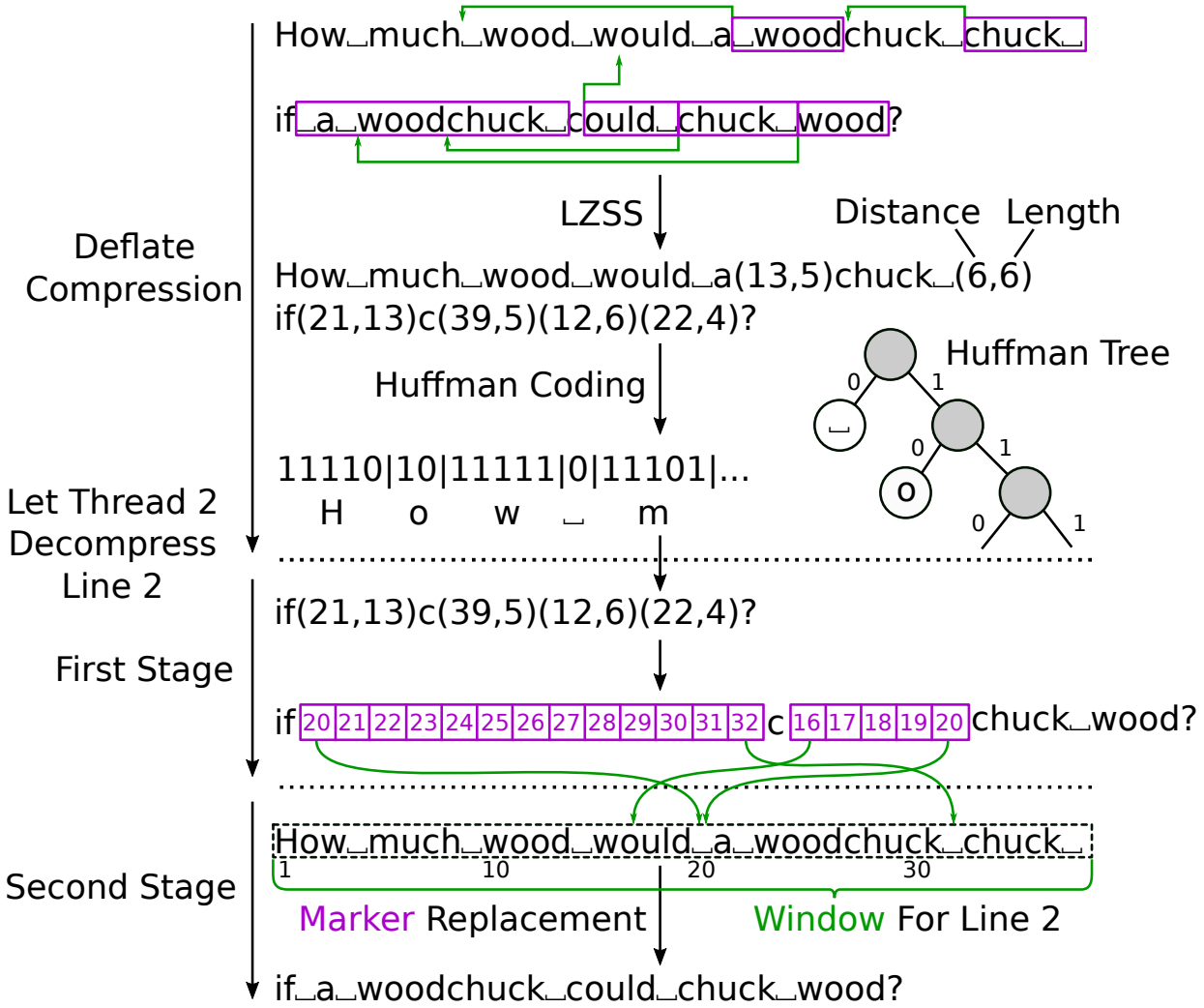
Challenges for Parallel Decompression:

- Find deflate block start offsets in bit stream
- Handling references to unknown data

→ Two-Staged Decompression as introduced by Kerbiriou and Chikhi (2019)

- Non-resolvable references result in markers that get resolved in the second stage after the 32 KiB window has become known

Parallelized Deflate Decompression



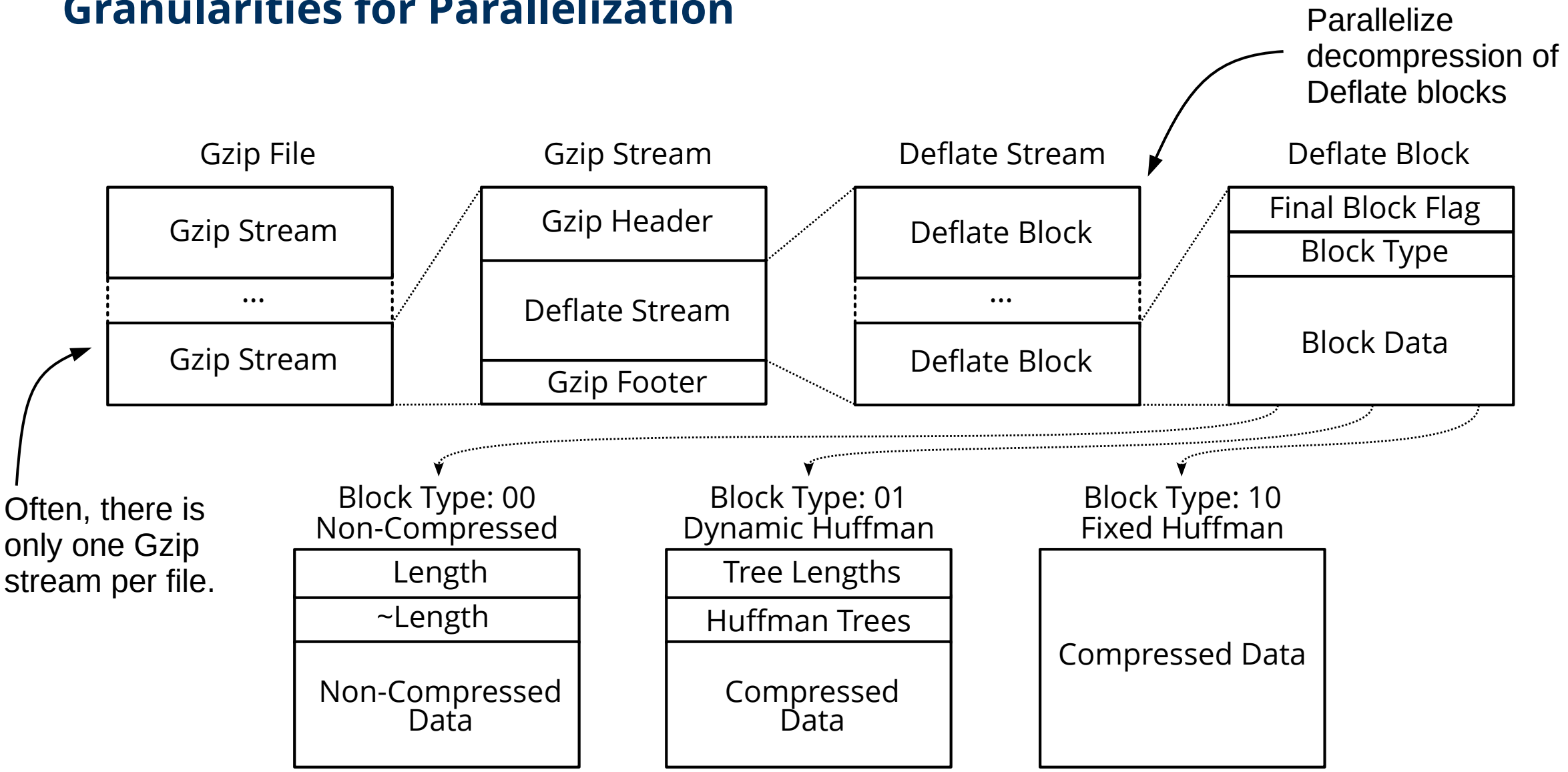
Challenges for Parallel Decompression:

- Find offsets in bit stream to start decompression from
- **Handling references to unknown data**

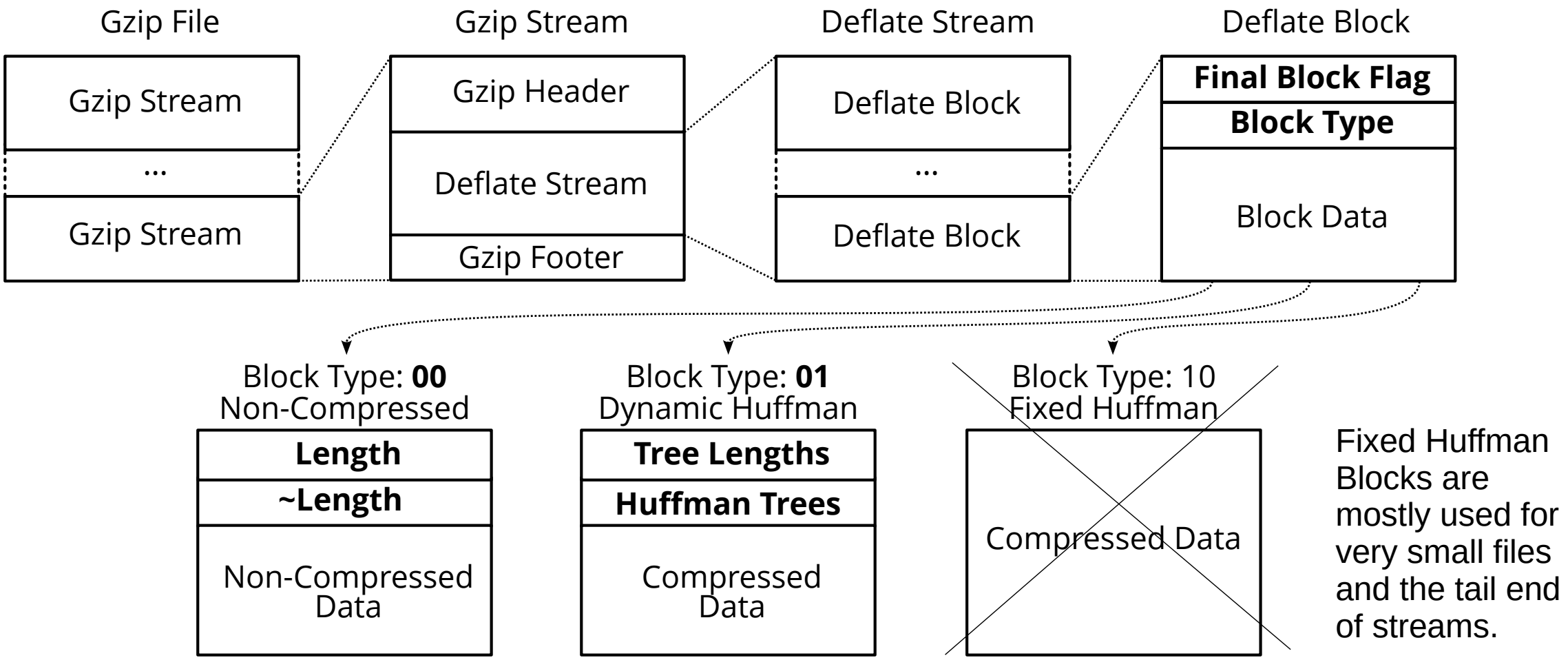
→ Two-Staged Decompression as introduced by Kerbiriou and Chikhi (2019)

- Non-resolvable references result in markers that get resolved in the second stage after the 32 KiB window has become known

Granularities for Parallelization



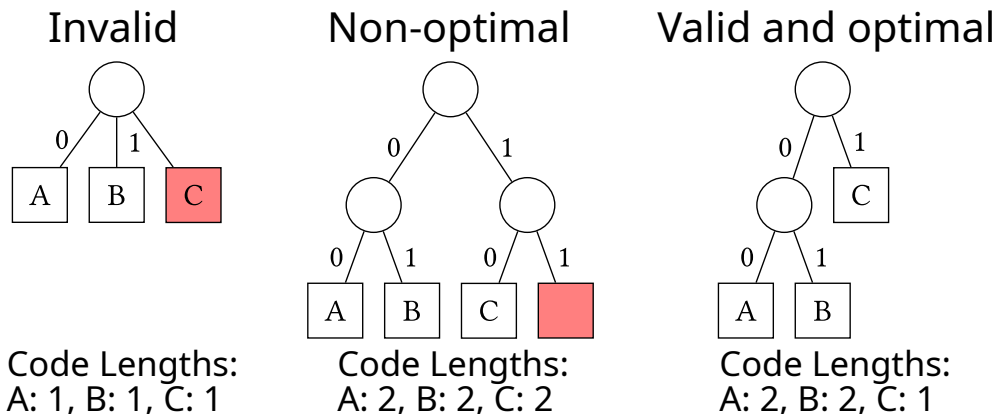
Redundancies that Help in Finding Deflate Blocks



Fixed Huffman Blocks are mostly used for very small files and the tail end of streams.

How Often Will Valid-Looking Block Headers be Found in Random Data?

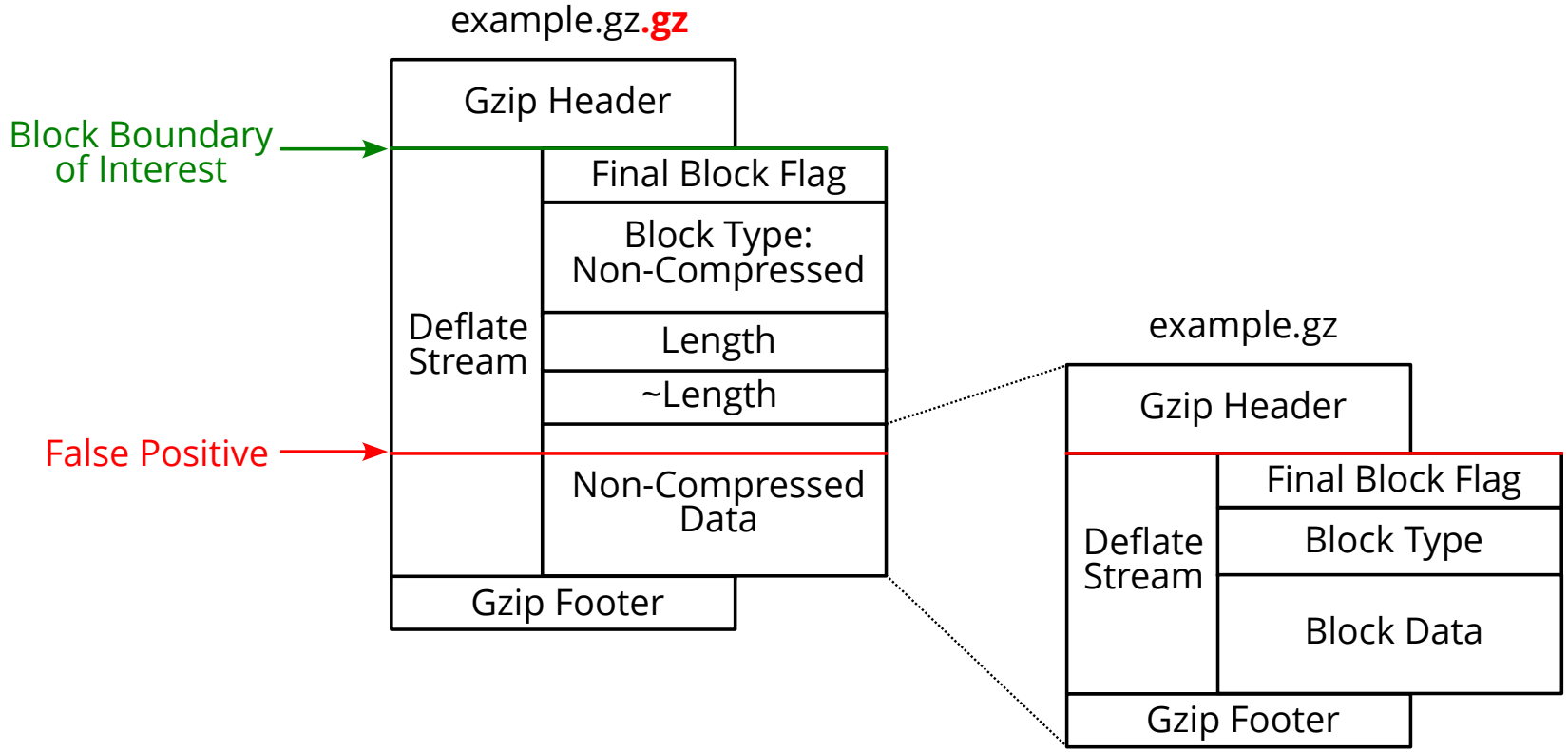
- **Deflate Blocks with Fixed Huffman codings:** Ignore because they are rare
- **Non-Compressed Deflate blocks:** Look for 16-bit lengths and their one's complement
  → *1 false positive per 525 kB*
- **Deflate Blocks with Dynamic Huffman codings:**
  Look for valid Deflate block headers and valid and optimal Huffman codings.
  ~200 offsets pass this test given 1 Tbits of random data → *1 false positive per 625 MB*



- **Pugz** reduces false positives further by checking that the decompressed data only contains characters in the range 9-126, an assumption made for FASTQ files

False Positives Can Be Crafted Using Non-Compressed Blocks

Example: Consider a gzip file that is compressed a second time.

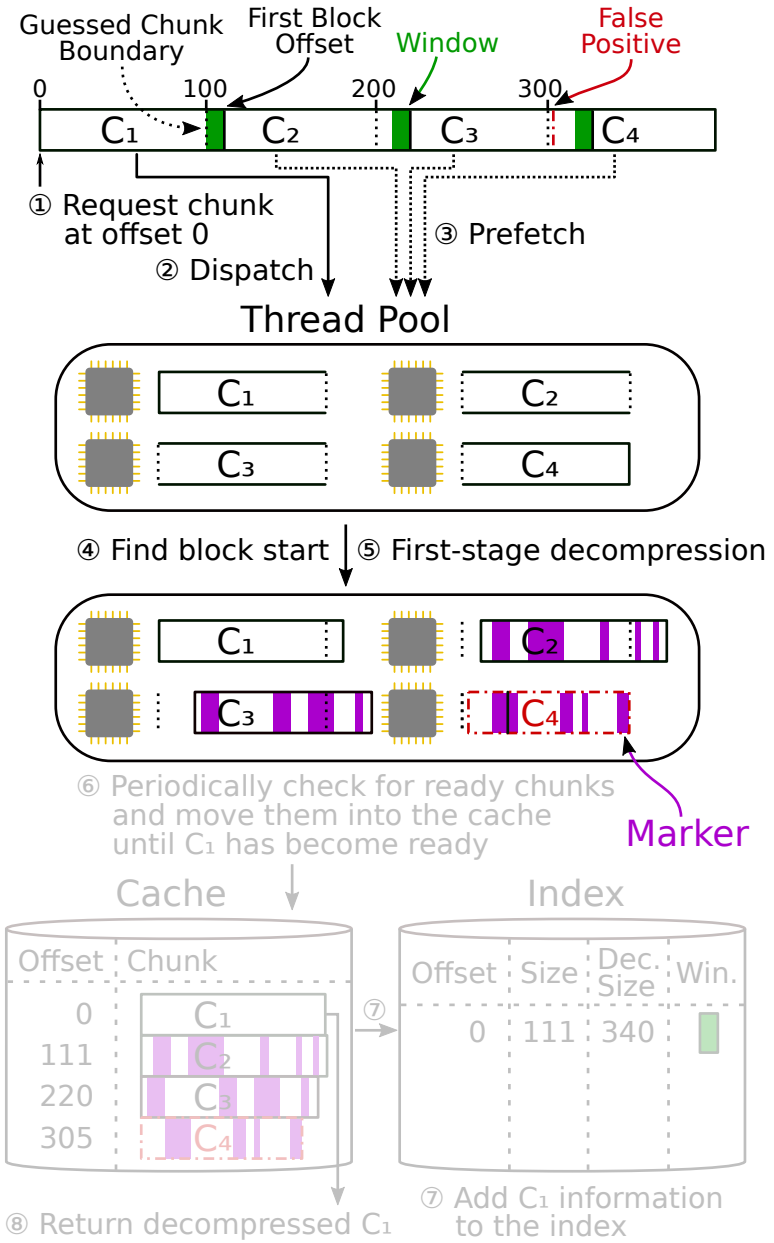


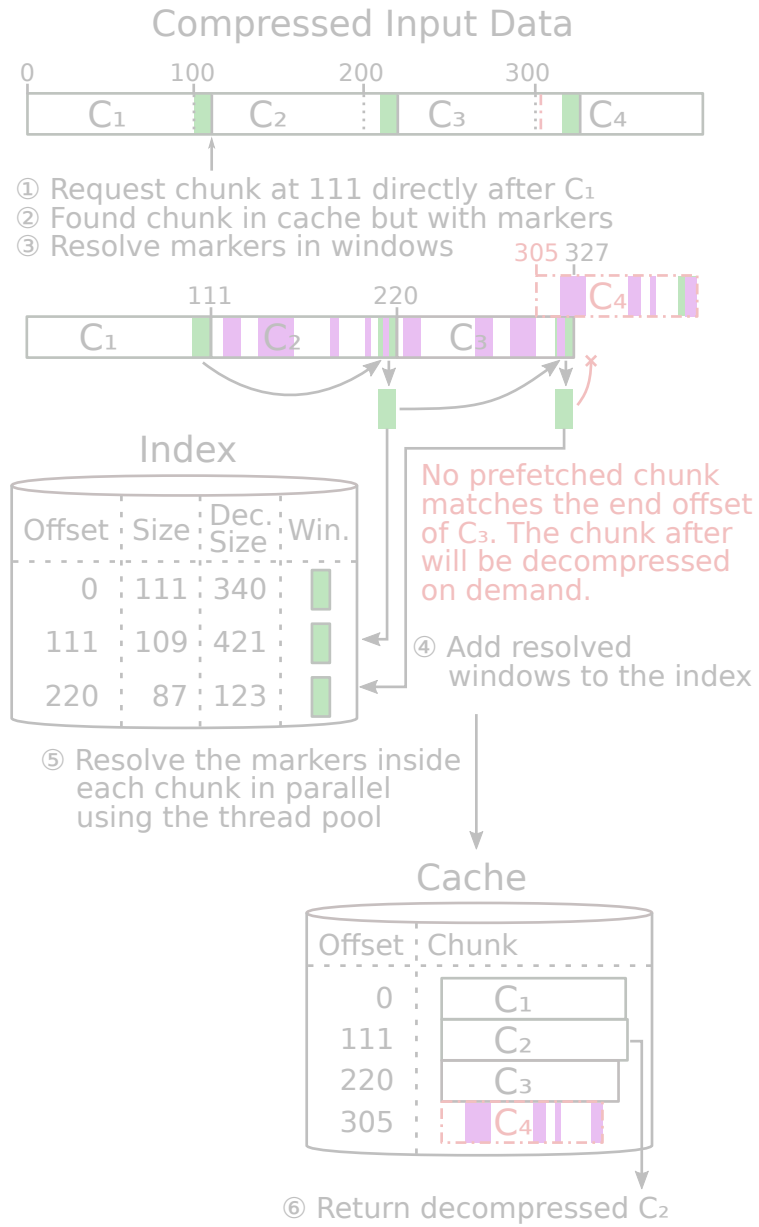
Sequences inside the compressed Block Data might be recognized as Deflate block headers

→ These cases need to be detected and handled

Implementation

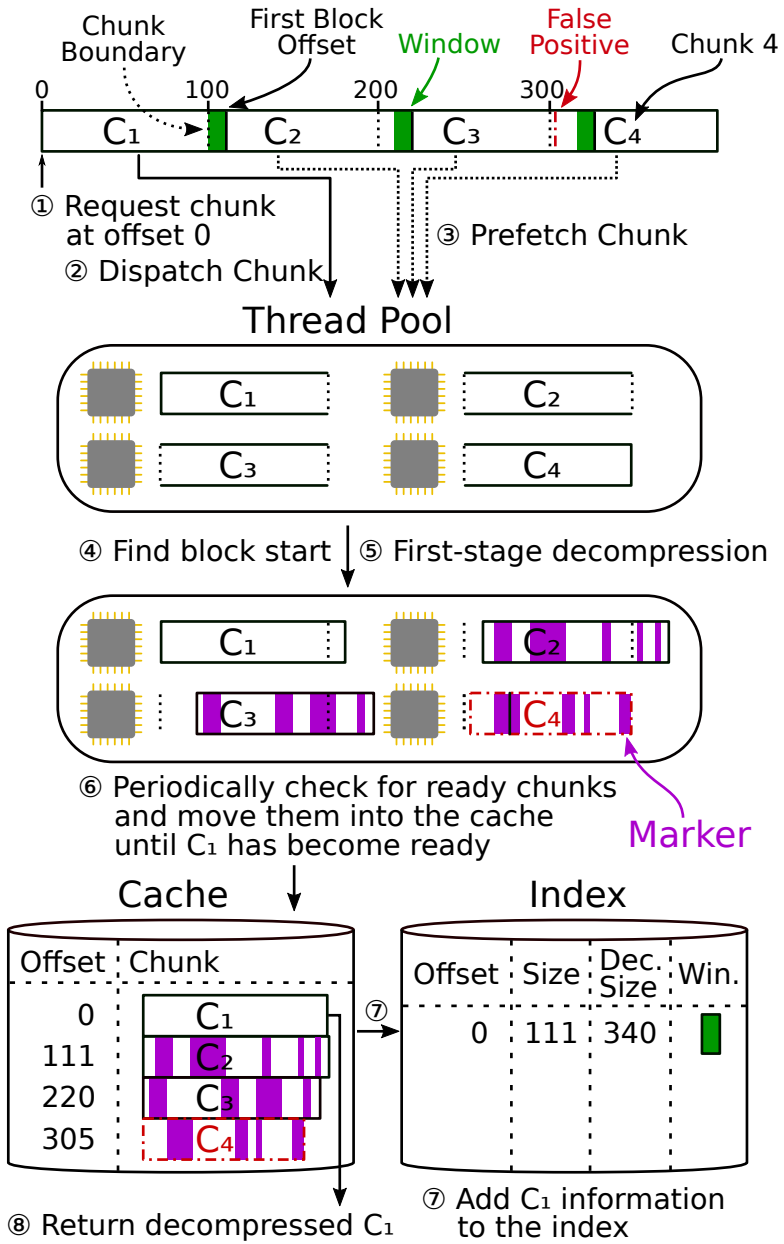
- Prefetching to generate work that can be processed in parallel
- Thread pool for work balancing
- Cache to speed up seeking and concurrent decompression
- Use block offset as cache key to catch false positives
- On-demand cache fill to recover from errors

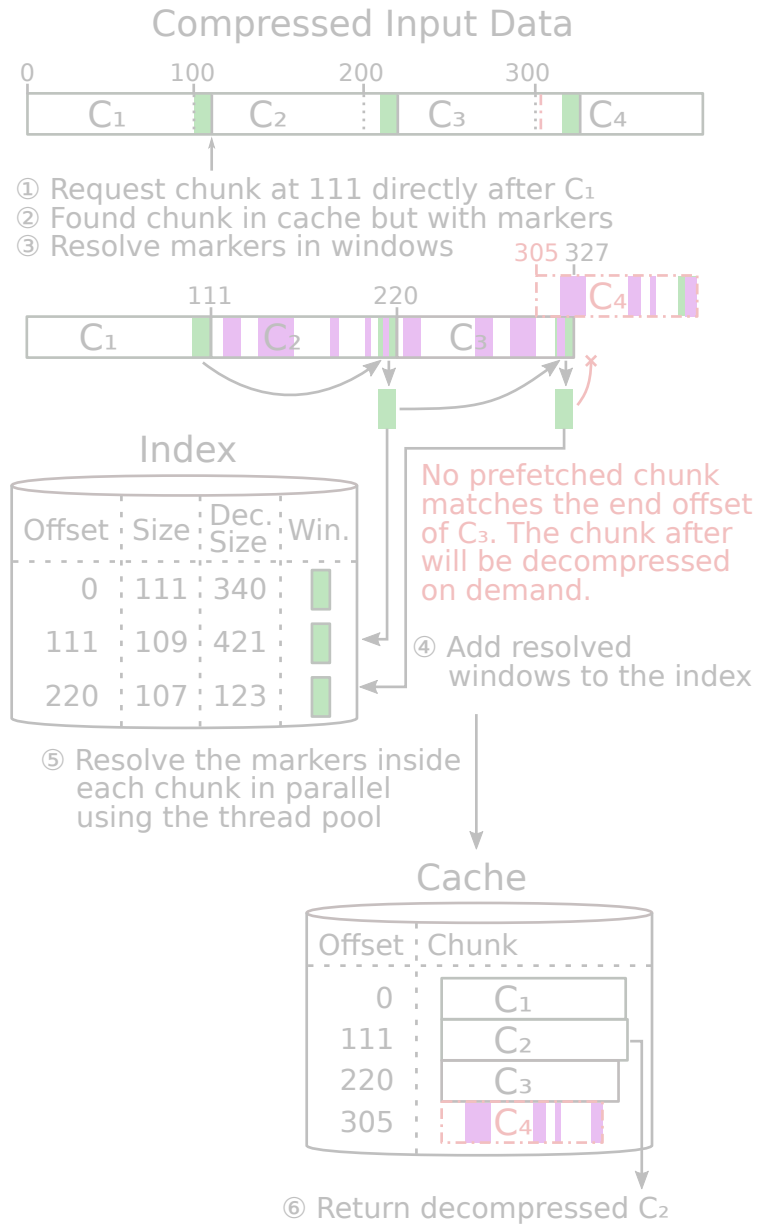




Implementation

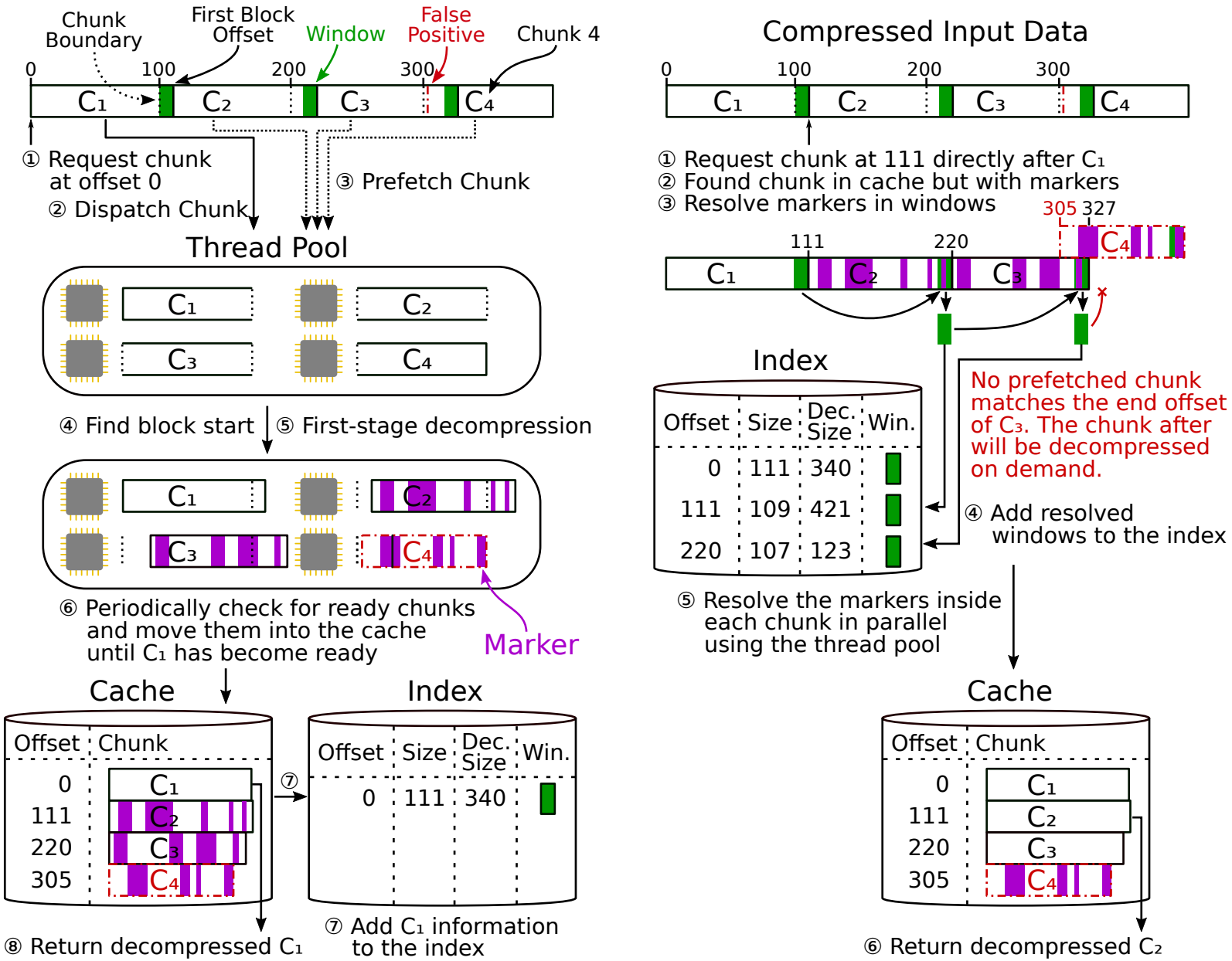
- Prefetching to generate work that can be processed in parallel
- Thread pool for work balancing
- Cache to speed up seeking and concurrent decompression
- Use block offset as cache key to catch false positives
- On-demand cache fill to recover from errors





Implementation

- Prefetching to generate work that can be processed in parallel
- Thread pool for work balancing
- Cache to speed up seeking and concurrent decompression
- Use block offset as cache key to catch false positives
- On-demand cache fill to recover from errors



Optimal Chunk Size

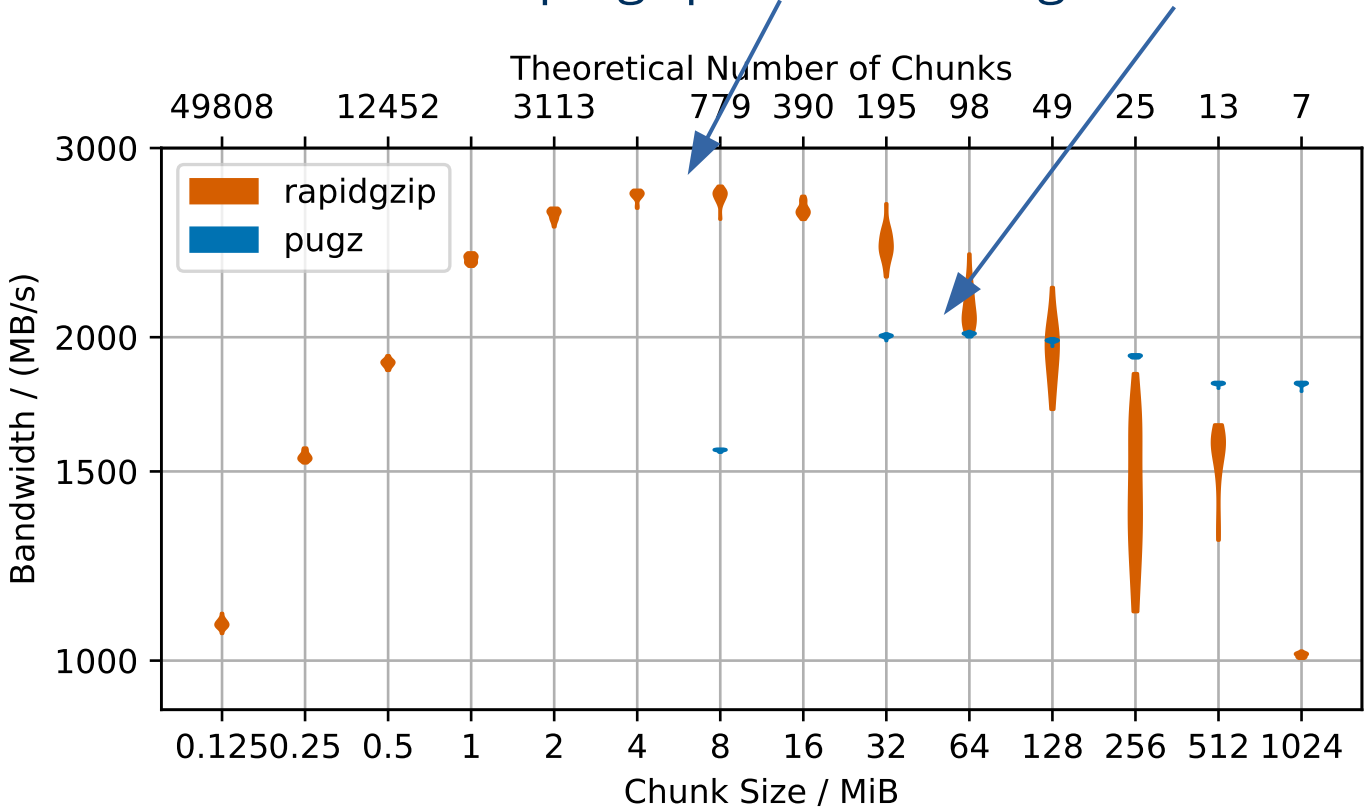
- For smaller chunk sizes, the block finder overhead leads to worse performance
- Larger chunk sizes lead to work balancing issues and also might adversely affect the cache behavior and allocation speed

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

3.8×

DBF … Dynamic Block Finder
NBF … Non-Compressed Block Finder

Optimal chunk sizes: Rapidgzip: 4-8 MiB, Pugz: 32-64 MiB

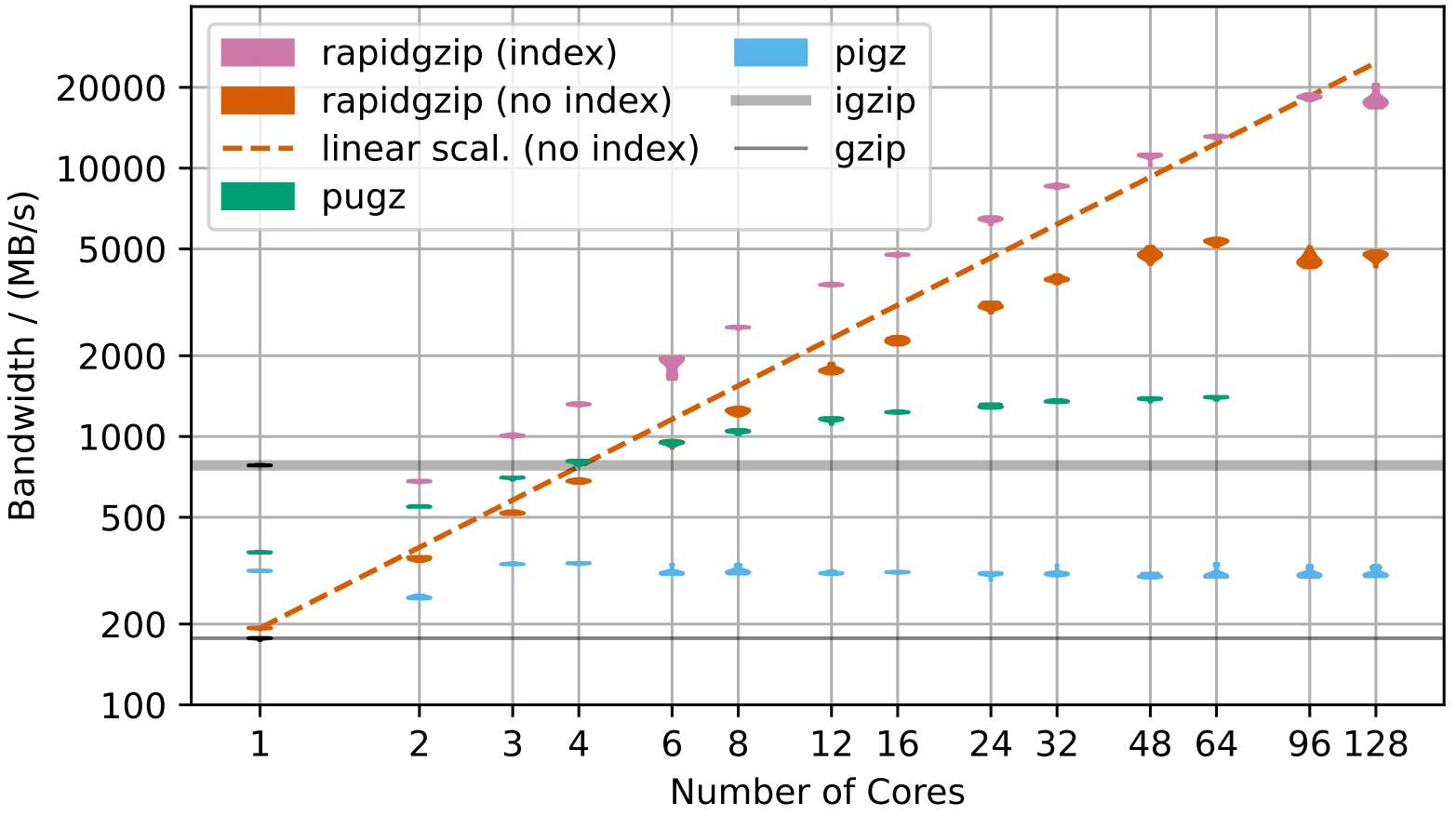


Decompression bandwidth using 16 cores and a 6.08 GiB test file, which decompresses to 8 GiB.

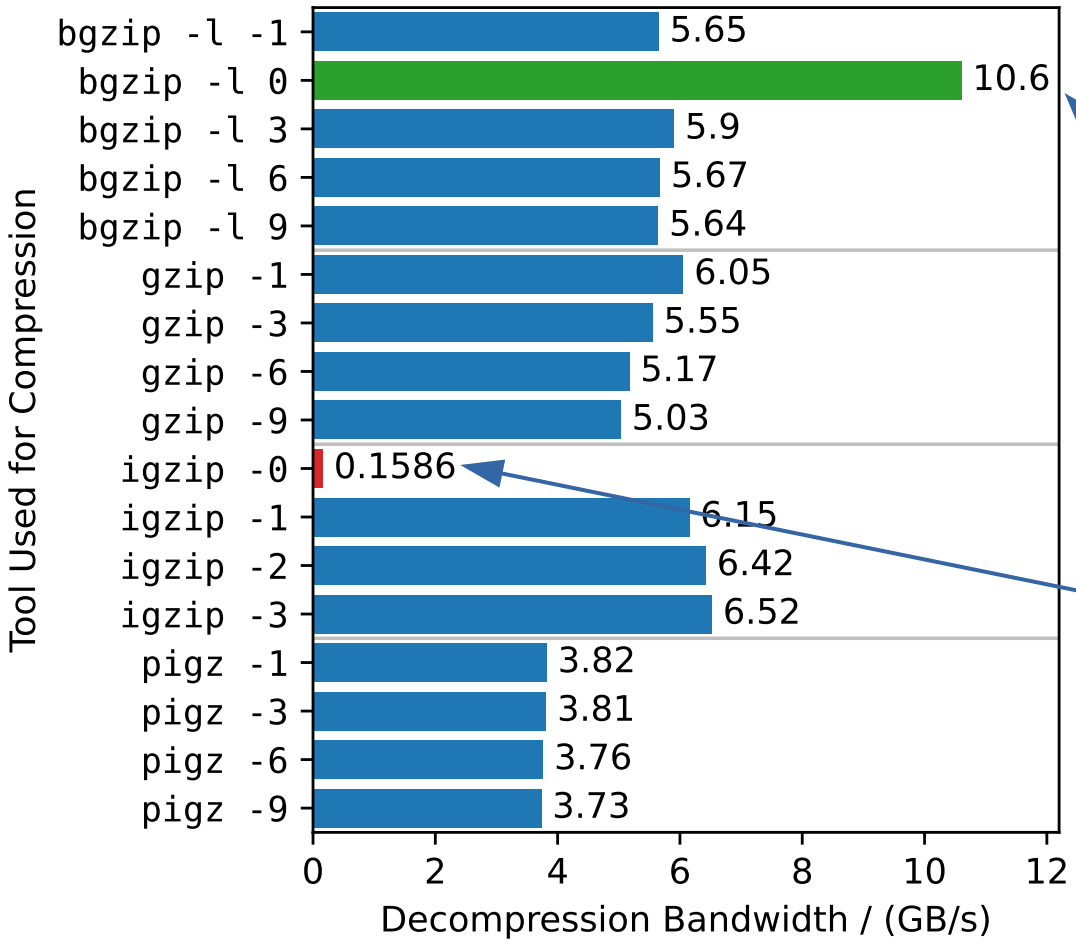
Decompression Benchmark: FASTQ File

Weak-scaling and writing the results to /dev/null

- `gzip` is the slowest, half as slow as zlib-based `pigz`
- `rapidgzip` with an index scales up to 20 GB/s, without an index up to 5 GB/s
- `pugz` tops out at 1.4 GB/s and crashes for 96+ cores
- `igzip` by Intel shows leading single-core performance
- `pigz` does not parallelize



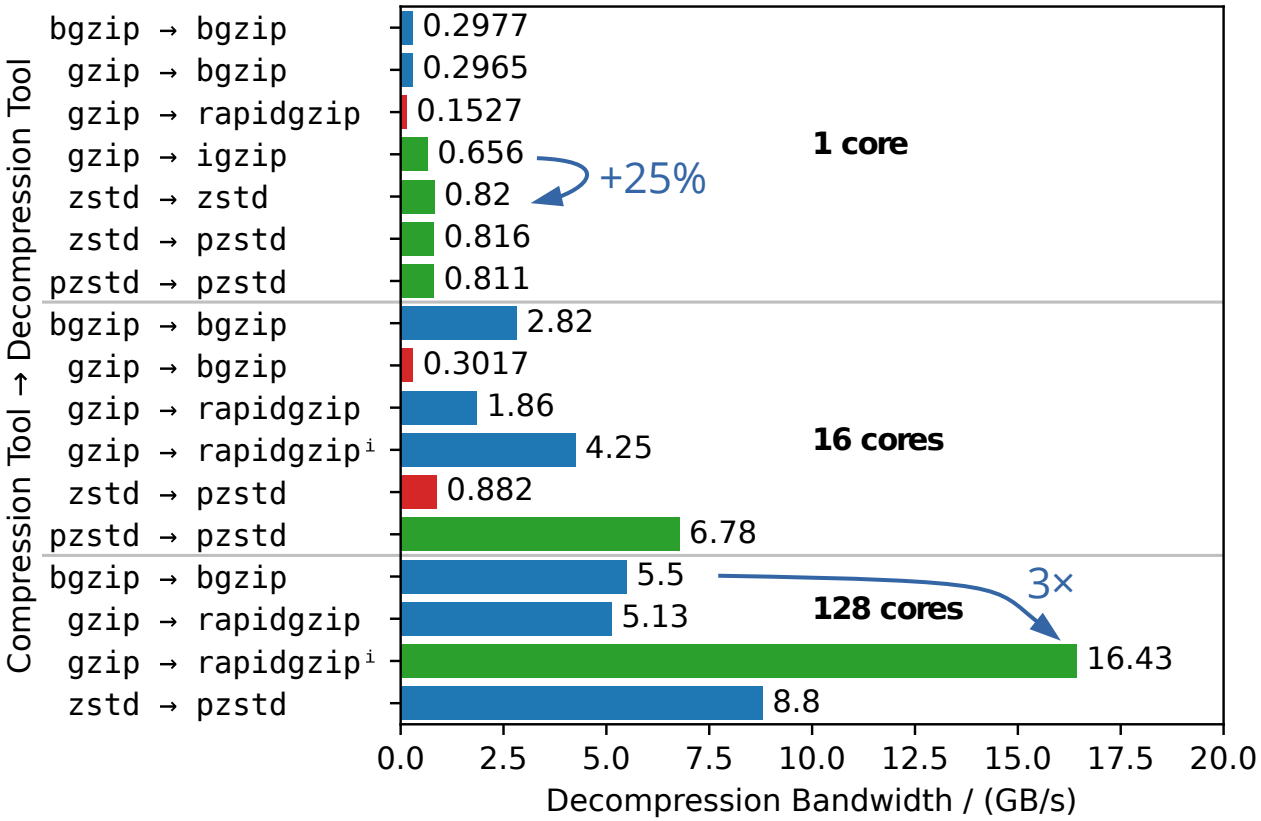
Benchmark: Various Gzip Compressors → Rapidgzip Decompression



- `rapidgzip` can parallelize decompression for gzip files produced with a wide variety of tools and compression levels
- Contains only Non-Compressed Deflate blocks so that decompression is reduced to a fast copy and some accounting
- Contains only a single Deflate block with Fixed Huffman coding and therefore cannot be parallelized

Benchmark: Competing File Formats

- `igzip` is surprisingly competitive to `zstd`
- `zstd` is the fastest in single-core decompression
- `bgzip` and `pzstd` can only decompress gzip/zstd files produced by themselves in parallel
- `zstd`-compressed (TAR) files are not eligible for random access and parallel decompression. `pzstd` is recommended instead to create multi-frame Zstandard files.
- Applying the `rapidgzip` approach to arbitrary Zstandard files might be infeasible because their window size is not limited to 32 KiB.



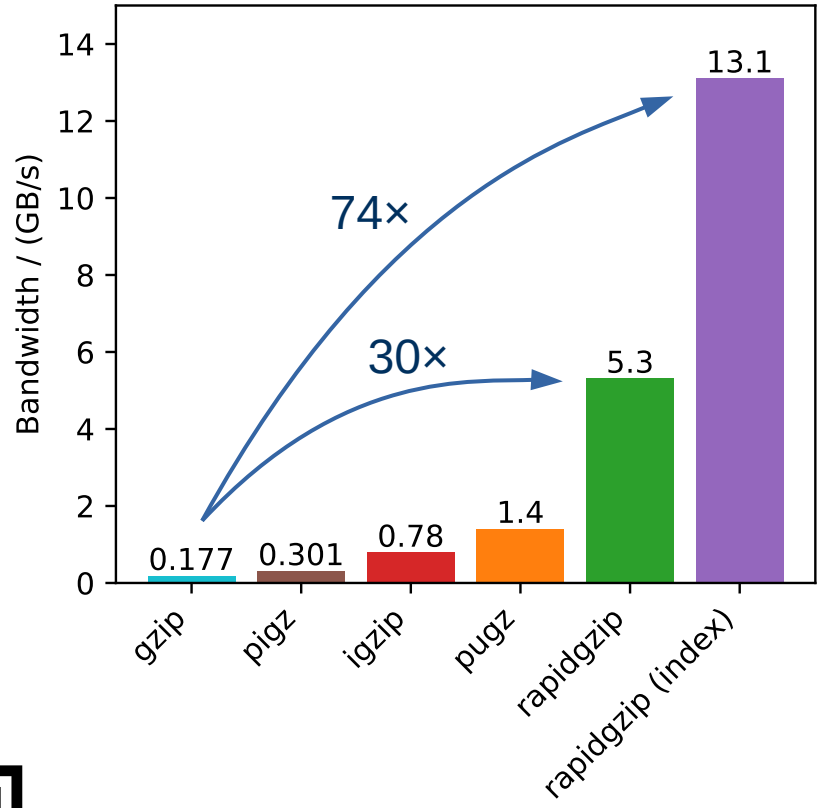
[i] rapidgzip decompression with an existing index

Improvements Since Submission

- High memory usage has been alleviated by limiting the decompressed chunk size
  Worst case (compression ratio ~1000)
  was: ~ 9 GB per thread
  now: **~ 200 MB per thread (configurable)**
- The Inflate implementation has been improved for high compression ratios
  → **25 % faster** for Silesia by using memcpy/memset for long references
- CRC32 computation has been added
  → The slice-by-16 algorithm has been implemented and parallelized using crc32_combine.
  Achieves **~ 4 GB/s per core** (**~ 6 % overhead** independent of parallelism)

Summary

- We have shown that the specialized approach for parallelized gzip decompression introduced by Kerbiriou and Chikhi (2019) can be generalized without affecting performance and stability.
- Our architecture achieves better performance, scales to more cores, adds robustness against false positives, and also increases versatility by adding fast seeking capabilities.
- An index is created internally on first time decompression and it can be exported and imported to speed up subsequent decompression and seeking.
- Can be used with ratarmount to mount .tar.gz archives.
- Available at https://github.com/mxmlnkn/rapidgzip



Decompression benchmarks on a 12 GB FASTQ file using 64 cores of an AMD EPYC 7702 @ 2.0 GHz processor.