Client-Side Data Compression

Artem Blagodarenko / Colin Faber

Transparent client-side data compression & decompression

Whamcloud

- Why develop this feature?
  - Reduces cost per-GB of data stored
  - Inexpensive way to improve interconnect bandwidth
  - Clients are easier to scale than servers

- The Lustre client-side data compression (CSDC) feature provides a transparent mechanism to compress and decompress data on Lustre clients
  - Configurable per file/directory via standard Lustre tools
  - Multiple compression algorithms supported (and more can be added)
  - Transparent to end-users when configured by admin
  - Scalable compression speed with number of client CPU cores

What's different about this compression method?

Whamcloud

- **Client-based compression model**
  - Compressed data stored on the servers, compressed and decompressed only on the clients
- **Works with LDISKFS based file systems**
  - No changes to LDISKFS on-disk data structures
- **Provides support for popular compression algorithms**
  - Currently `lzo`, `lz4`, `lz4hc` and `gzip`
- **Easy to expand framework to add new compression types**
  - Up to 255 compression types possible, selectable on a per-file/directory or per-component basis
- **Does not depend on ZFS compression on the server**
  - Transparent ZFS compression integration possible in the future

Features

- In-kernel `lzo` algorithm available for all client kernel versions
- Backport modern kernel `lz4` algorithm (and variants), for older kernels
  - Balance compression/decompression speed and space usage
- Per-file component enable / disable compression algorithm, level, chunk size
- Can be set as default for directory tree or whole filesystem
- Existing files can be (re)compressed after write, or during migration to slower storage
- Transparent compressed write / read to the application
- Older clients will **see** compressed files, but not be able to read / write data
- Compatible with other Lustre features (migration, mirroring, encryption*, pools, quota, etc.)
- No on-disk data structure changes necessary

Tools changes to enable/check compressed files

- '`lfs setstripe`' to set file layout components using data compression
  - **`--compress|-Z <type>[:<level>]`** Set component compression algorithm type and level (1-15)
  - **`--compress_chunk=<size>`** Compression chunk size, adjust to power-of-two multiple of 64KiB, <= stripe_size

Example:

```
$ lfs setstripe -E eof -Z lz4:5 --compress-chunk=512k <dir|new_file>
```

- '`lfs getstripe`' to show compressed component parameters

Example:

```
$ lfs getstripe <file> | grep compr
lcme_compr_type:     lz4
lcme_compr_lvl:      5
lcme_compr_chunk_kb: 512
lmm_pattern:         raid0,compress
```

Tools changes to enable/check compressed files (cont.)

- '`lfs find`' to search for (un-)compressed files
  - **`--comp-flags=[^]compress`** to locate file with/without compressed components
  - **`--comp-flags=[^]nocompr`** to locate file with/without setting component compress preference
  - **`[!] --layout=compress`** to locate file with/without compressed components
  - **`[!] --compress-type=<type>`** find files with/without specified compress algorithm
  - **`[!] --compress-level=[+-]<level>`** find files with/without specified compress level

Examples:

- Find already compressed files

```
$ lfs find --comp-flags=compress <dir>
```

- Find compressed files with type other than `lz4` (e.g. to recompress with `lz4` in background)

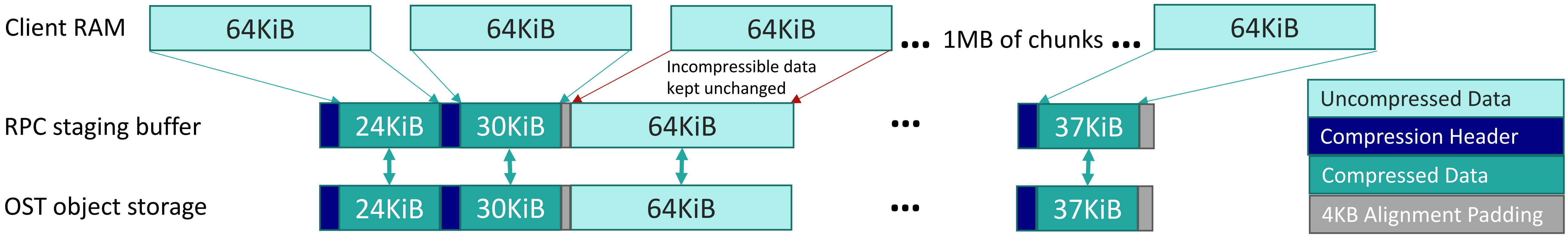
```
$ lfs find --compress-type=^lz4 <dir>
```

- Find compressed files with level < 5 (e.g. to recompress to a higher level)

```
$ lfs find --compress-level=-5 <dir>
```

Data Compression Pipeline

- Compress data on client in chunks (64KiB-1MiB+)
  - Keep "uncompressed" chunks for incompressible data/file (`.gz`, `.jpg`, `.mpg`, …), chunks

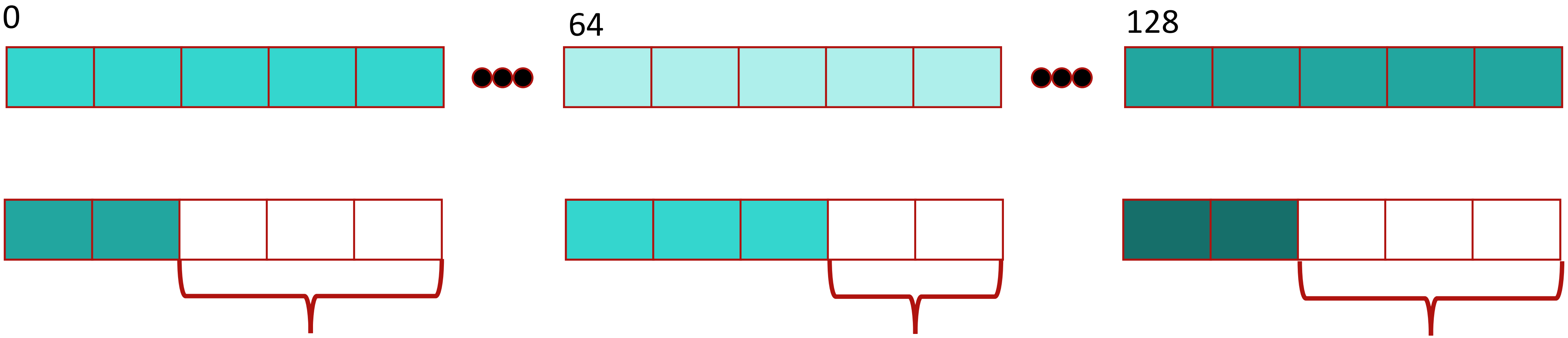


- Client writes/reads whole chunk(s), (de-)compresses to/from RPC staging buffer
  - Independent/parallel compression for each request on separate core to reach GB/s speeds
  - Larger chunks improve compression, but higher read-modify-write overhead
- Optional write to uncompressed file mirror for random IO pattern
- Optional data (re-)compression during mirror/migrate (via transfer agent)

LDISKFS allocator changes for improved data density

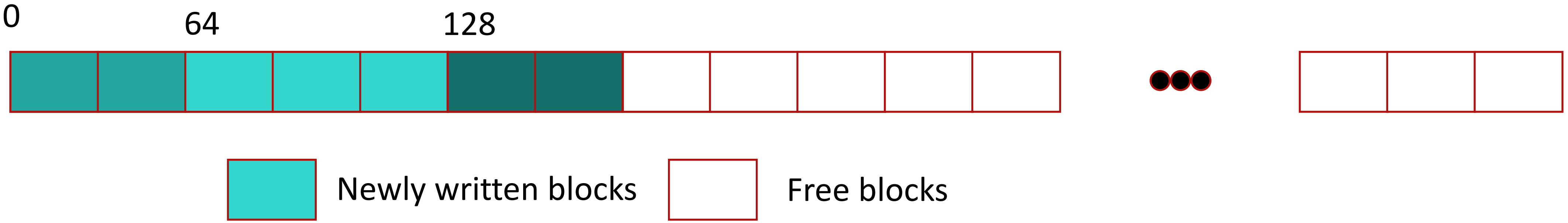
- Compression will always reduce data size by at least one 4KB block, or it is skipped
- OST will write chunks starting at file logical offset for each chunk to LDISKFS
  - Client must read and write **whole chunks** starting at an even multiple of the chunk offset
- From LDISKFS perspective compressed chunks have holes between them in file/block allocation
  - For example, 64KiB chunk compressed to 24KiB the next chunk will have a 40KiB "hole” from LDISKFS logical offset perspective
- Optimize on-disk blocks to be contiguous
- Client sends `OBD_BRW_COMPRESSED` flag with each compressed write RPC
- Flag informs allocator that holes will never be filled, and should pack chunks densely

LDISKFS changes to avoid allocation holes in files



LDISKFS optimization. These blocks normally reserved for multi-client interleaved writes, but in case of compression these blocks will be unused. Gaps decrease read performance (for **HDDs**) and add fragmentation.

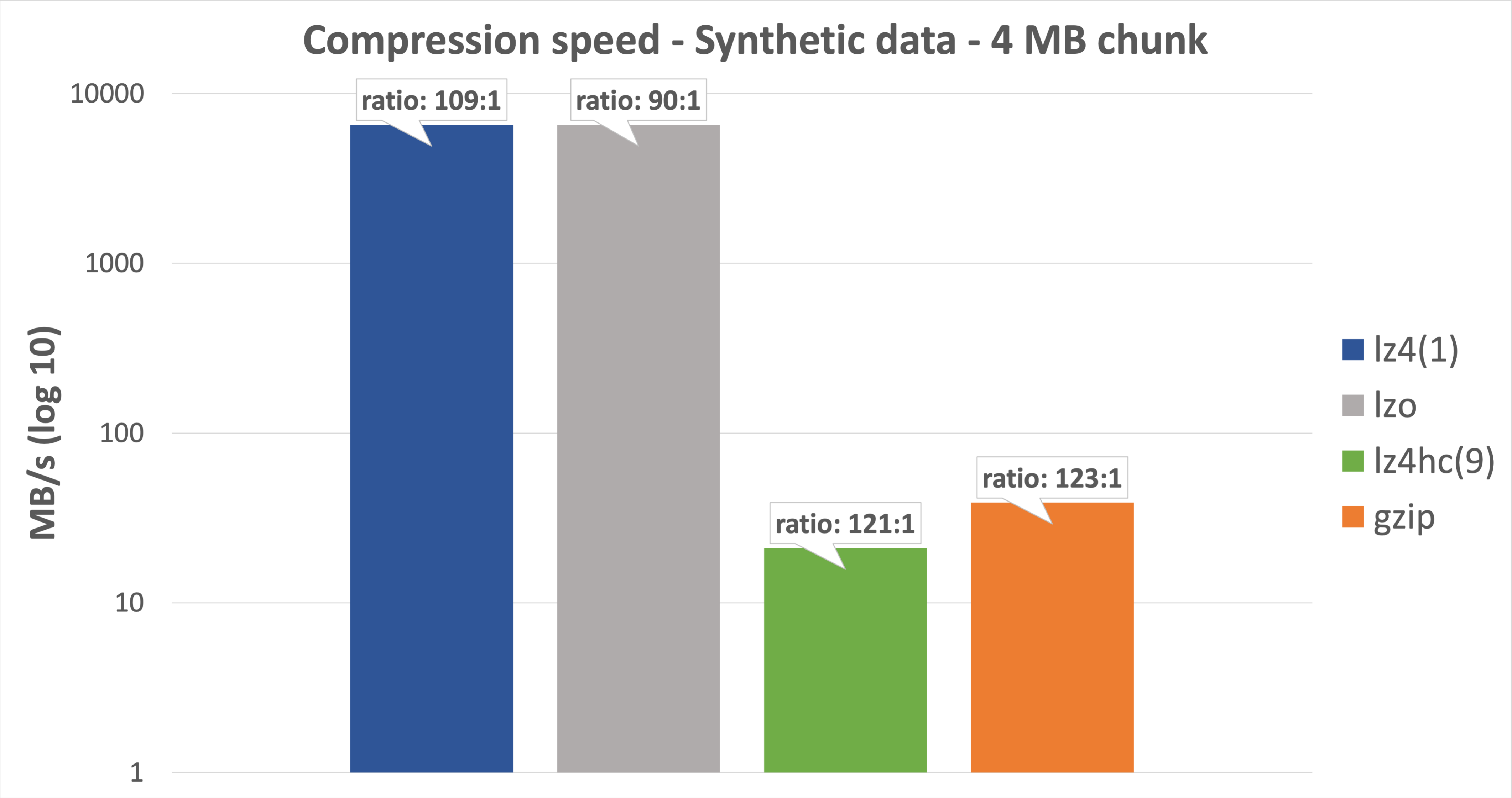
```
LDISKFS receives OBD_BRW_COMPRESSED flag and disables the optimization.
Blocks are being written sequentially. This optimizes writing and reading.
```



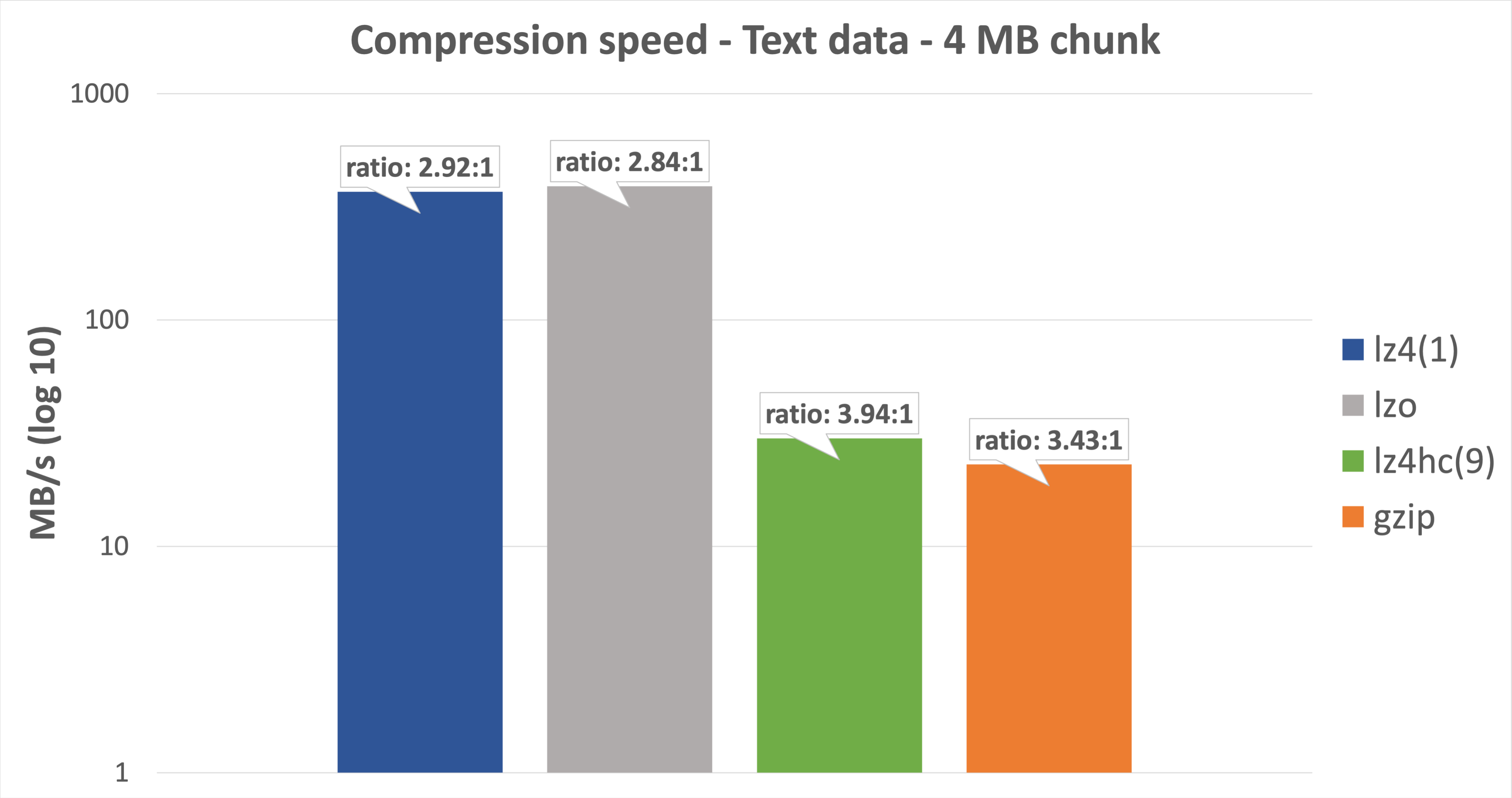
Compression algorithms performance

- Given by `kcompr.ko` test module results
- Compare `lz4`, `lz4hc`, `lzo` and `gzip` algorithms with different chunk sizes
- 3 different input files to compress/decompress:
  - Synthetic binary data
  - Text file made of Lustre sources
  - HDF ocean climate data from https://nsidc.org/data/ae_dyocn/versions/2
- Comprehensive performance testing is on-going

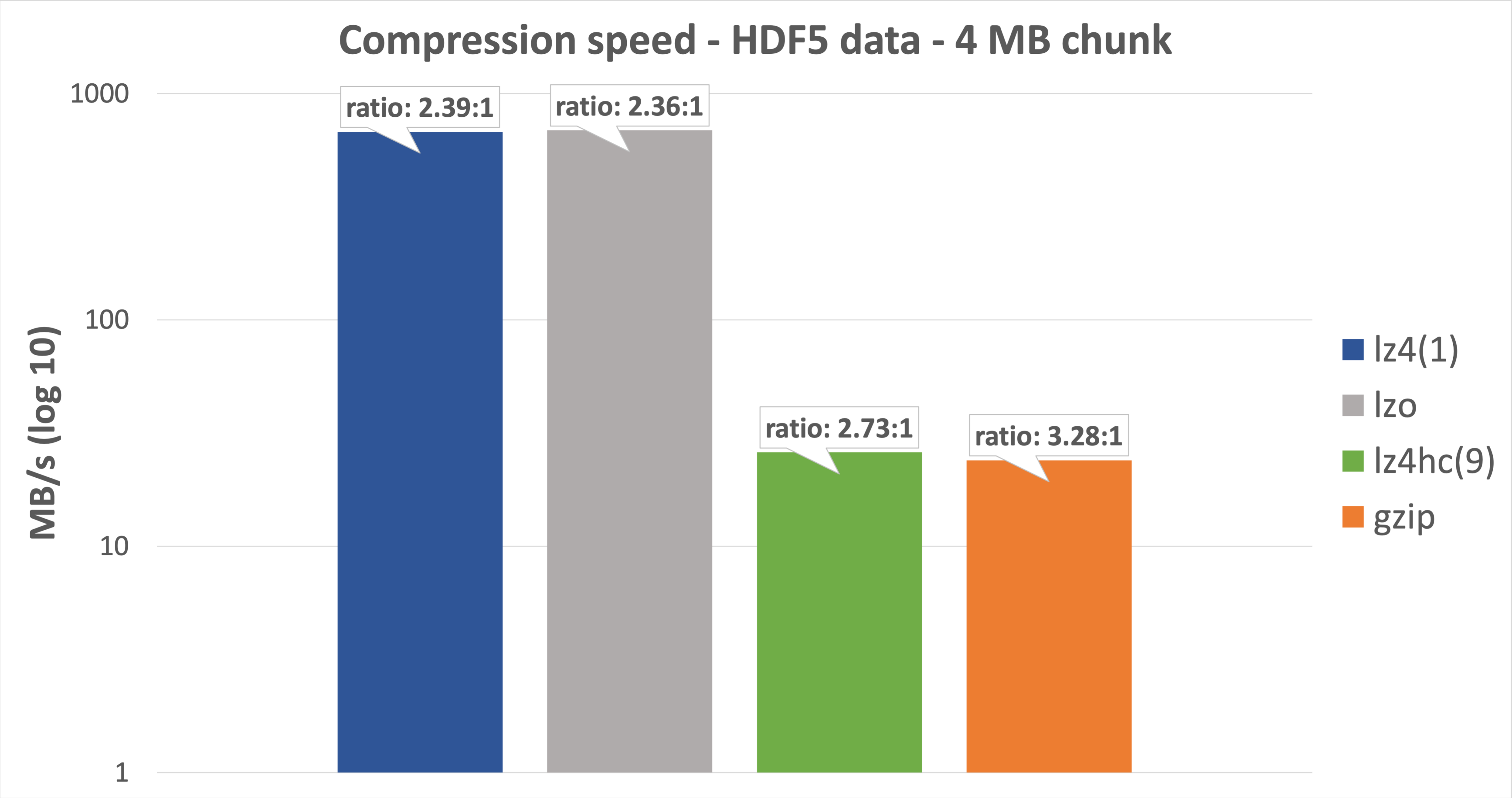
Userland compression algorithm testing: Synthetic data



Userland compression algorithm testing: Text data



Userland compression algorithm testing: HDF5 climate data



Performance of compression algorithms & considerations

Whamcloud

- Performance and compression ratio depend on the data being compressed
  - Ability to define algorithm, level and chunk size per directory or file component
- Compression algorithms and chunk size settings have different demands on CPUs
- Compression buffer pools allocate client memory
- Block allocation on target disks will behave differently for compressed data
- Flash based targets will yield the best overall performance for compressed data as allocator determinations have less impact

A simple demonstration of CSDC with sanity testing

Whamcloud

```
lfs setstripe -Eeof -Z lzo --compress-chunk=64k sanity.sh
```

```
== sanity test 460a: Compress/decompress text test  15:19:46 (1675178386)

Compression:
13+1 records in
13+1 records out
879690 bytes (880 kB, 859 KiB) copied, 0.0068837 s, 128 MB/s

Decompression:
13+1 records in
13+1 records out
879690 bytes (880 kB, 859 KiB) copied, 0.00628479 s, 140 MB/s

860K    /usr/src/lustre/tests/sanity.sh
352K    /mnt/lustre/d460a.sanity/sanity.sh
860K    /tmp/decompressed_sanity.sh

860 -rwxrwxr-x 1 ubuntu ubuntu 879690 Jan 31 15:11 /usr/src/lustre/tests/sanity.sh
352 -rw-r--r-- 1 root root 879690 Jan 31 15:19 /mnt/lustre/d460a.sanity/sanity.sh
860 -rw-r--r-- 1 root root 879690 Jan 31 15:19 /tmp/decompressed_sanity.sh

74dcbb585d95ffe21e3cee5c6fb38f92  /usr/src/lustre/tests/sanity.sh
74dcbb585d95ffe21e3cee5c6fb38f92  /mnt/lustre/d460a.sanity/sanity.sh
74dcbb585d95ffe21e3cee5c6fb38f92  /tmp/decompressed_sanity.sh
```

Original data
Compressed data
Decompressed data

What about a ZFS backend?

- Can a ZFS backend be used with CSDC?
  - Possibly yes (untested), to store compressed data directly to ZFS OST objects
  - Further development/integration needed to properly integrate with ZFS compression
  - Client may need to use different compression types/header to fully integrate
- CSDC is focused on LDISKFS, however nothing prohibits ZFS as a backend
  - If issues arise, they can be fixed/optimized by handling OBD_BRW_COMPRESSED flag on the server
- ZFS copy-on-write approach assumes a new block allocation on every write
  - Holes in file allocation are less of a problem, which differs from LDISKFS

When is this feature expected?

- CSDC is planned to be available with the Lustre 2.17 release
  - Will likely need both client and server updates to fully support

- Major functionality under development right now
  - Basic client, server, and user tool changes already implemented
  - Further optimizations and cross-feature integration ongoing

- CSDC specific testing for performance and stability - on-going

- Feature under active development since October 2022

Whamcloud

Thank You!
Questions?