Direct I/O for RNTuple Columnar Data

Jonas Hahnfeld[1,2] Jakob Blomer[1] Philippe Canal[3] Thorsten Kollegger[2]
jonas.hahnfeld@cern.ch

[1] CERN, Geneva, Switzerland

[2] Goethe University Frankfurt, Institute of Computer Science, Frankfurt, Germany

[3] Fermi National Accelerator Laboratory, Batavia, IL, USA

CHEP 2024 – October 21, 2024

This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA).

Motivation

- RNTuple: designated successor of TTree columnar format for HL-LHC
  - Modern design, optimized for current hardware, with parallelism in mind
  - See many presentations this week, including a plenary on Wednesday

Motivation

- RNTuple: designated successor of TTree columnar format for HL-LHC
  - Modern design, optimized for current hardware, with parallelism in mind
  - See many presentations this week, including a plenary on Wednesday
- Developed highly scalable parallel writing without merging / post-processing
  - Advantage: multi-threaded job produces one output file directly
  - Presented concepts and performance evaluation at Euro-Par 2024 (preprint on arXiv)

Motivation

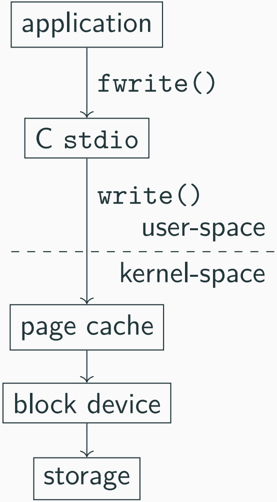
- RNTuple: designated successor of TTree columnar format for HL-LHC
  - Modern design, optimized for current hardware, with parallelism in mind
  - See many presentations this week, including a plenary on Wednesday
- Developed highly scalable parallel writing without merging / post-processing
  - Advantage: multi-threaded job produces one output file directly
  - Presented concepts and performance evaluation at Euro-Par 2024 (preprint on arXiv)
- Synthetic benchmarks: up to storage bandwidth limit on SSDs
  - Today: exploiting Direct I/O to increase that limit

Direct I/O

- Under Linux, by default files are accessed via the *page cache*
  - Reads are cached in unused memory
  - Writes are buffered and flushed in bulk at a later point

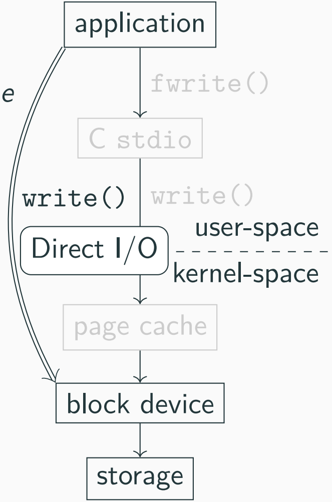
Direct I/O

- Under Linux, by default files are accessed via the *page cache*
  - Reads are cached in unused memory
  - Writes are buffered and flushed in bulk at a later point
- Page cache is only one layer in the storage system
  - Buffers in user-space, caches in kernel and hardware...



Direct I/O

- Under Linux, by default files are accessed via the *page cache*
  - Reads are cached in unused memory
  - Writes are buffered and flushed in bulk at a later point
- Page cache is only one layer in the storage system
  - Buffers in user-space, caches in kernel and hardware...
- Direct I/O allows bypassing the page cache
  - Originally implemented for database applications



Requirements for Direct I/O (from `man 2 open`)

- No clear documentation on requirements:
  - “may impose alignment restrictions on [...]”
  - “vary by filesystem and kernel version and might be absent entirely”
  - “handling of misaligned [Direct I/O] also varies”

Requirements for Direct I/O (from `man 2 open`)

- No clear documentation on requirements:
  - “may impose alignment restrictions on [...]”
  - “vary by filesystem and kernel version and might be absent entirely”
  - “handling of misaligned [Direct I/O] also varies”

- Alignment restrictions on...
  - ... file offset and byte count
  - ... user-space buffer addresses

Requirements for Direct I/O (from `man 2 open`)

- No clear documentation on requirements:
  - "may impose alignment restrictions on [...]"
  - "vary by filesystem and kernel version and might be absent entirely"
  - "handling of misaligned [Direct I/O] also varies"

- Alignment restrictions on...
  - ... file offset and byte count
  - ... user-space buffer addresses

- General advice: offsets, lengths, and addresses should be multiples of
  - "filesystem block size (typically 4096 bytes)", or
  - "logical block size of the block device (typically 512 bytes)"

Direct I/O for RNTuple

- RNTuple data stored in *pages* of variable size
  - Also transparently compressed with unknown ratio
  - Generally not aligned appropriately

Direct I/O for RNTuple

- RNTuple data stored in *pages* of variable size
  - Also transparently compressed with unknown ratio
  - Generally not aligned appropriately

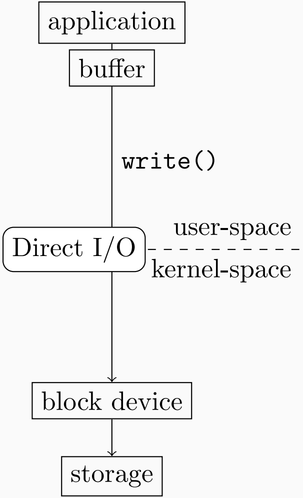
- Experiments showed significant gains for writing
  - (will come back to Direct I/O for reading at the end)

Direct I/O for RNTuple

- RNTuple data stored in *pages* of variable size
  - Also transparently compressed with unknown ratio
  - Generally not aligned appropriately

- Experiments showed significant gains for writing
  - (will come back to Direct I/O for reading at the end)

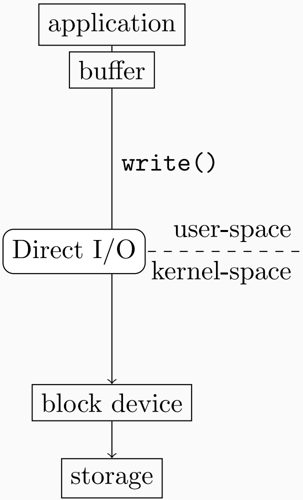
- Solution: implement aligned buffering in user-space
  - For writing now done when creating a new file



Direct I/O for RNTuple

- RNTuple data stored in *pages* of variable size
  - Also transparently compressed with unknown ratio
  - Generally not aligned appropriately
- Experiments showed significant gains for writing
  - (will come back to Direct I/O for reading at the end)
- Solution: implement aligned buffering in user-space
  - For writing now done when creating a new file
- Direct I/O can be activated with write option:

```
RNTupleWriteOptions options;
options.SetUseDirectIO(true);
```



RNTuple Writing with Direct I/O – Setup

- Benchmarking server with AMD EPYC 7702P (64 cores / 128 threads)
  - Running AlmaLinux 9.4, ROOT compiled with GCC 11.4.1
  - Samsung PM1733 NVMe SSD formatted with `ext4`

[1] https://github.com/hahnjo/rntuple-apps/tree/main/random-write

RNTuple Writing with Direct I/O – Setup

- Benchmarking server with AMD EPYC 7702P (64 cores / 128 threads)
  - Running AlmaLinux 9.4, ROOT compiled with GCC 11.4.1
  - Samsung PM1733 NVMe SSD formatted with `ext4`

- Synthetic benchmark, writing randomly generated data[1]
  - Fixed number of 20 million entries per thread
  - Two top-level fields: "event ID" and a vector of `floats`

[1] https://github.com/hahnjo/rntuple-apps/tree/main/random-write

RNTuple Writing with Direct I/O – Setup

- Benchmarking server with AMD EPYC 7702P (64 cores / 128 threads)
  - Running AlmaLinux 9.4, ROOT compiled with GCC 11.4.1
  - Samsung PM1733 NVMe SSD formatted with `ext4`

- Synthetic benchmark, writing randomly generated data[1]
  - Fixed number of 20 million entries per thread
  - Two top-level fields: "event ID" and a vector of `floats`

- 4 MiB buffer for writing (tradeoff between size and performance)
  - Offsets, lengths, and addresses aligned to 4096 bytes (see also next slide)

[1] https://github.com/hahnjo/rntuple-apps/tree/main/random-write

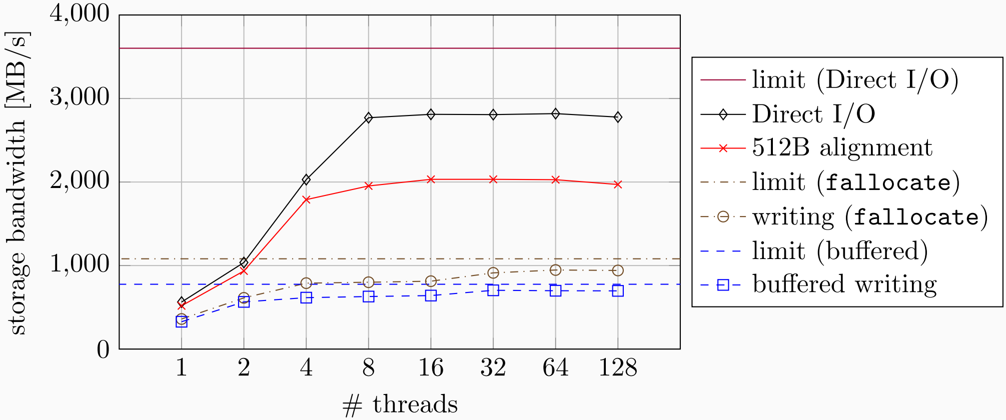
RNTuple Writing with Direct I/O – Setup

- Benchmarking server with AMD EPYC 7702P (64 cores / 128 threads)
  - Running AlmaLinux 9.4, ROOT compiled with GCC 11.4.1
  - Samsung PM1733 NVMe SSD formatted with `ext4`
- Synthetic benchmark, writing randomly generated data[1]
  - Fixed number of 20 million entries per thread
  - Two top-level fields: "event ID" and a vector of `floats`
- 4 MiB buffer for writing (tradeoff between size and performance)
  - Offsets, lengths, and addresses aligned to 4096 bytes (see also next slide)
- Reduced maximum page size to 128 KiB
  - Fits in L2 cache of benchmark system

[1] https://github.com/hahnjo/rntuple-apps/tree/main/random-write

RNTuple Writing with Direct I/O – No Compression

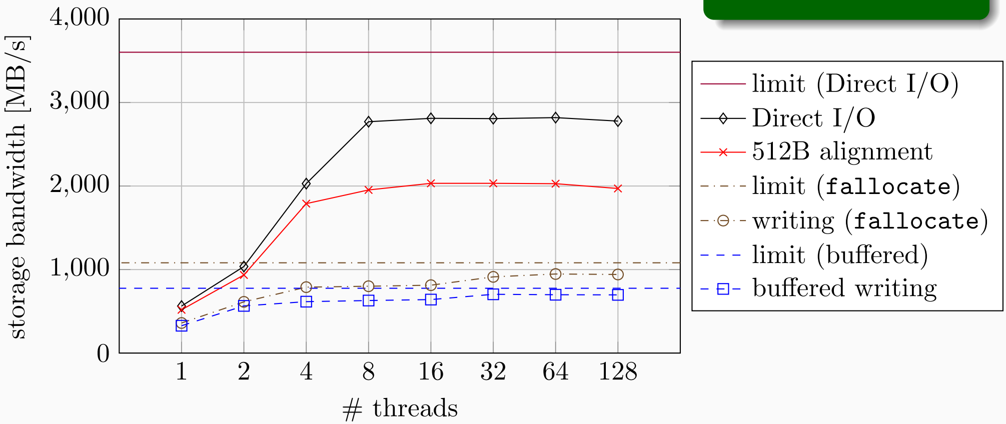
- Bandwidth limit: 775 MB/s → more than 3,600 MB/s with Direct I/O!
  - Measured with Flexible I/O Tester (`fio`)
  - Optimization with `fallocate` already presented at Euro-Par 2024



RNTuple Writing with Direct I/O – No Compression

- Bandwidth limit: 775 MB/s → more than 3,600 MB/s with Direct I/O!
  - Measured with Flexible I/O Tester (`fio`)
  - Optimization with `fallocate` already presented at E

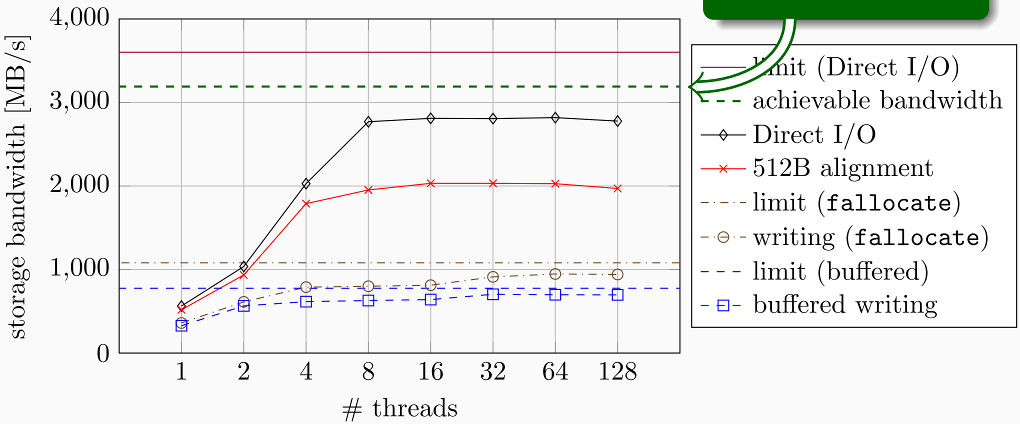
Can we do better?



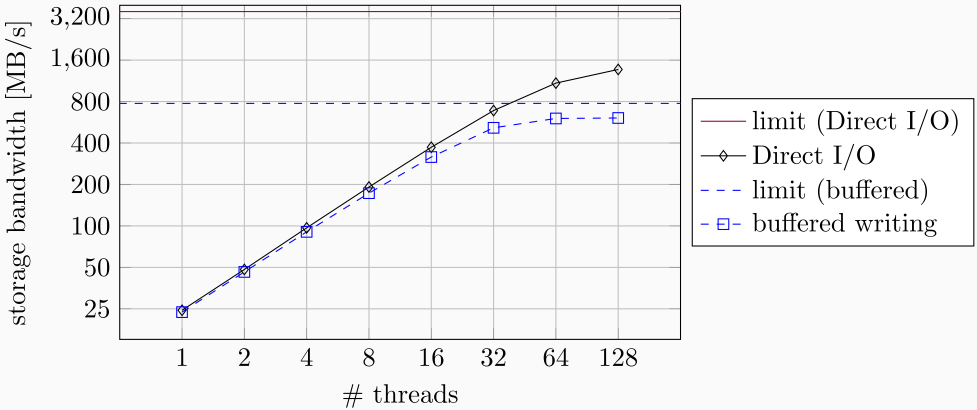
RNTuple Writing with Direct I/O – No Compression

- Bandwidth limit: 775 MB/s → more than 3,600 MB/s with Direct I/O!
  - Measured with Flexible I/O Tester (`fio`)
  - Optimization with `fallocate` already presented at E

Can we do better?

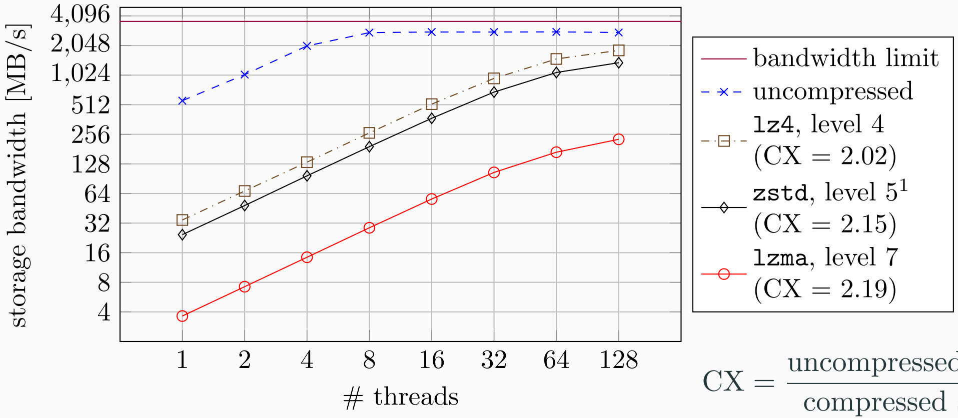


- Storage bandwidth: based on compressed size, what is written to storage



RNTuple Writing with Direct I/O – Compression Algorithms

- Storage bandwidth: based on compressed size, what is written to storage

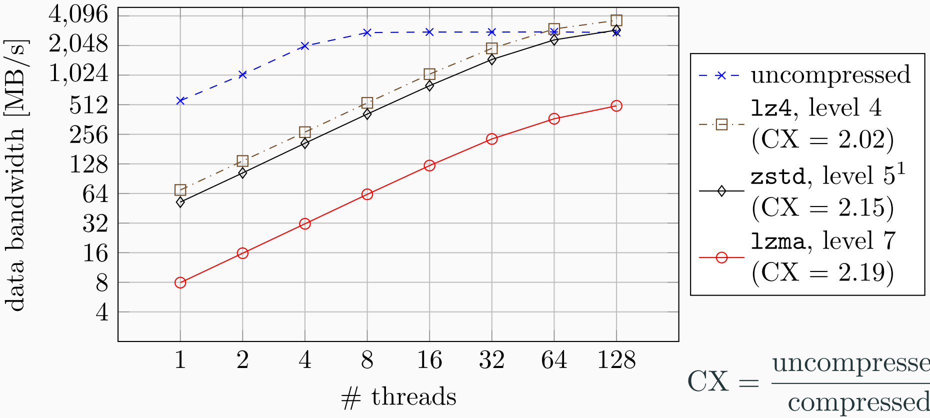


$$CX = \frac{\text{uncompressed size}}{\text{compressed size}}$$

[1] For zstd, ROOT maps level 5 to Zstandard compression level 10.

RNTuple Writing with Direct I/O – Data Bandwidth

- Data bandwidth: based on *un*compressed size, what the user fills into RNTuple



$$\mathrm{CX} = \frac{\text{uncompressed size}}{\text{compressed size}}$$

[1]For zstd, ROOT maps level 5 to Zstandard compression level 10.

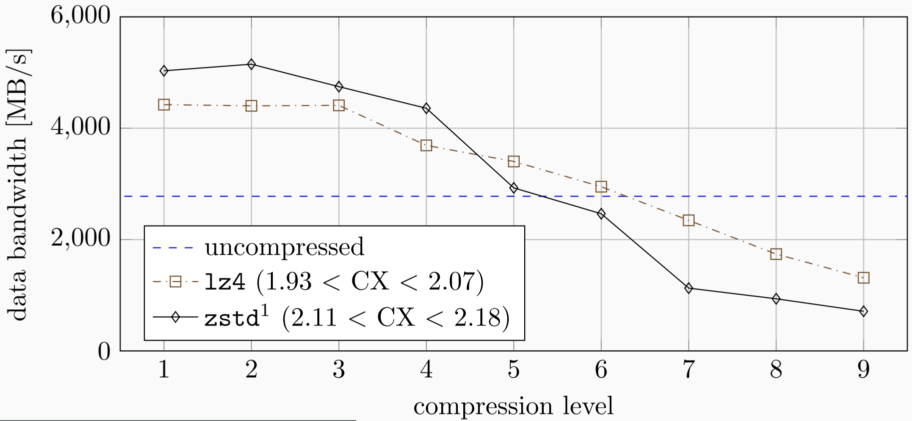
RNTuple Writing with Direct I/O – Maximize Data Bandwidth

- Q: At 128 threads, which compression level gives the highest data bandwidth?
  - Possible use cases: online data streaming, burst buffering

[1] For `zstd`, ROOT scales the compression level by a factor 2.

RNTuple Writing with Direct I/O – Maximize Data Bandwidth

- Q: At 128 threads, which compression level gives the highest data bandwidth?
  - Possible use cases: online data streaming, burst buffering



[1] For zstd, ROOT scales the compression level by a factor 2.

Direct I/O for Reading?

- Similar alignment challenges as for writing
  - Extend and align buffering for reading, add padding to read requests
  - Note: need to disable optimized reading with `io_uring`

[2] https://github.com/jblomer/iotools

Direct I/O for Reading?

- Similar alignment challenges as for writing
  - Extend and align buffering for reading, add padding to read requests
  - Note: need to disable optimized reading with `io_uring`

- Can observe faster read times in sample analysis benchmarks[2] (up to factor 2x)

[2] https://github.com/jblomer/iotools

Direct I/O for Reading?

- Similar alignment challenges as for writing
  - Extend and align buffering for reading, add padding to read requests
  - Note: need to disable optimized reading with `io_uring`

- Can observe faster read times in sample analysis benchmarks[2] (up to factor 2x)
- However, much lower effect on overall run time
  - Up to 12 % in LHCb sample analysis with a single thread and no compression
  - No gain for ATLAS sample analysis with sparser reading pattern
  - Reasons: Asynchronous cluster prefetching and reads with `io_uring`

[2] https://github.com/jblomer/iotools

Direct I/O for Reading?

- Similar alignment challenges as for writing
  - Extend and align buffering for reading, add padding to read requests
  - Note: need to disable optimized reading with `io_uring`

- Can observe faster read times in sample analysis benchmarks[2] (up to factor 2x)
- However, much lower effect on overall run time
  - Up to 12 % in LHCb sample analysis with a single thread and no compression
  - No gain for ATLAS sample analysis with sparser reading pattern
  - Reasons: Asynchronous cluster prefetching and reads with `io_uring`

- Also tested with Analysis Grand Challenge
  - Dataset of 787 files converted to RNTuple
  - No statistically significant change in performance

[2] https://github.com/jblomer/iotools

Conclusions

- Implemented option for using Direct I/O in RNTuple writing
  - Demonstrated benefits together with scalable parallel writing
  - Reaching up to 2.8 GB/s for uncompressed data (can be improved to 3.2 GB/s)
  - Up to 5 GB/s data bandwidth with cheap compression level

- If you have use cases for high bandwidths with parallel writing, please talk to us!