ATLAS PUB Note

ATL-COM-SOFT-2019-001

26th March 2020

Draft version 1.0

Impact of ROOT file parameters on ATLAS Analysis Object Data

The ATLAS Collaboration

ATLAS Analysis Object Data (xAOD) and derivative representations (DxAOD) are based on the ROOT file format. In this study, we investigate the impact of ROOT file parameters on file size and reading speeds of (D)xAODs in common ATLAS workflows. This covers LZMA, ZLIB and LZ4 compression at different levels, as well as ROOT features like *Autoflush*, *Splitlevel* and the *TTreeCache*. Recommendations are given for different scenarios, mainly with respect to different event sizes. These recommendations can be used as a starting point for detailed studies of new usage scenarios or new proposed file formats.

Contents

1 Introduction

In the coming runs, the LHC accelerator will provide higher luminosity of particle collisions to the ATLAS experiment [1]. This results in more simultaneous collisions per event, which also leads to a higher demand of disk space to store the events at the same level of detail as in previous runs. The demand for efficient processing is increasing at a similar rate, since more events per time will have to be processed with the available hardware. Multiple technical and conceptual approaches to this issue are explored within the collaboration, coordinated by the AMSG-R3 group [2, 3].

In this study we investigate how the ATLAS analysis object data format (AOD) is affected by different file storage options, provided by ROOT. Our focus lies on optimizing file sizes on disk and reading speeds from disk. The main AOD format is called (primary) xAOD [4] and contains reconstructed physics objects (e.g. particles, tracks, jets, ...) for each event. There are many derivative formats, called DxAOD, which prepare the primary xAOD for their intended analyses by removing or adding information to the stored objects and events.

The measurements cover different compression algorithms and specific ROOT options influencing how data is structured when stored on disk. Additionally, we study the thread scaling of ROOTs multithreaded branch decompression. All measurements are done for a primary xAOD file, as well as a “big”, a “medium” and a “small” derivation.

In the subsections of Section 2, each measurement is discussed for all four (D)xAOD formats. Section 3 mentions ideas for follow-up studies and gives a summary of all recommendations for default configurations of (D)xAODs.

2 Measurements

All tests in this study collect I/O performance metrics while reading the test files from disk. Reading is done by emulating ATLAS typical accesses to the files by fetching the data from the main TTree [5], named CollectionTree . The measurements are done for a primary xAOD file, a “big” (TOPQ1, ~170 kB per event on average), a “medium” (SUSY5, ~60 kB) and a “small” (TRUTH3, ~5 kB) derivation. The primary xAOD contains simulated ttbar events and the TOPQ1 and SUSY5 files are derived from it. The TRUTH3 file is derived from a different (EVNT) file, but its small event size makes it a good candidate to study the behavior of proposed future formats with small event sizes (DAOD_PHYSLITE) [3]. All branches inheriting from IParticleContainer can be read from the CollectionTree, which is accounting for roughly 90% of the whole file size.

For file access, three separate backends can be used: EventLoop, AthAnalysis and ROOT itself. All three approaches have different initialization behaviors. EventLoop is used by default and is a lightweight implementation of xAOD reading in ATLAS. It has short initialization times, which is the time before the eventloop starts and data is read from the CollectionTree for every event. Eventloop is typically used in ATLAS analysis frameworks. AthAnalysis is using the same infrastructure as the ATLAS reconstruction framework Athena. It is not used in any tests presented here. ROOT standalone reading allows us to test implicit multithreaded branch decompression.

During the tests, I/O performance metrics are collected in two ways. First, ROOT provides the PerfStats mechanism to access a range of performance statistics form within the process:

- Time spent in the event loop (starting from the 2nd event)
- Time spent reading
- Time spent unzipping the data
- Read bytes
- Average number of bytes read in one go
- Number of read operations
- Reading speed in MB/s
- Number of read events

Second, *Dstat* is used to collect system data from the Linux kernel. For this reason all tests assume a "quiet" machine with as few parallel programs executed as possible, to not bias these metrics. This is always an approximation, which is why tests are repeated multiple times to give an average measurement with its standard deviation. Dstat collects the following metrics on a per-second resolution:

- CPU load
- Disk I/O
- Network I/O
- Total I/O
- Used memory & Vmem
- Paging operations

Some of these metrics are independent of initialization times, but some, especially all Dstat metrics, are biased by longer/shorter initialization times. Usually these differences become relatively small when the job runs over many events.

Clearing the relevant system caches between individual tests ensures that test files are read from disk and not cached in memory.

All tests are executed on the same hardware:

- Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 4 physical cores, 8 logical cores
- 2× 128kB L1 (instruction, data), 1MB L2 (unified), 8MB L3 (unified)
- 8GB DDR4 RAM @ 2133 MHz
- HDD: WDC WD5000AZRZ-0 500GB, 64MB Cache, 5400rpm, 150 MB/s (Host to disk), 6GB/s (Buffer to Host), ~10 to 50 IOPS
- 1000 MB/s ethernet connection

Table 1: Different compression algorithms and levels with their effect on reading and file size for the primary xAOD test file.

Algorithm	**Level**	**File size** (B)	**Event rate** (Hz)
lz4	1	8.756×10^9	63.0±1.1
lz4	3	8.756×10^9	62.3±0.8
lz4	5	7.674×10^9	67.3±1.1
lz4	7	7.597×10^9	67.5±0.8
lz4	9	7.556×10^9	68.1±1.2
zlib	1	7.381×10^9	58.7±0.6
zlib	3	7.268×10^9	59.6±0.7
zlib	5	7.030×10^9	59.4±0.7
zlib	7	6.896×10^9	60.5±0.8
zlib	9	6.782×10^9	61.1±0.8
lzma	1	6.203×10^9	15.4±0.0
lzma	3	6.149×10^9	15.6±0.0
lzma	5	5.978×10^9	15.5±0.1
lzma	7	5.901×10^9	15.6±0.0
lzma	9	5.901×10^9	15.5±0.0

2.1 Compression Algorithm & Level

ROOT provides three compression algorithms: LZ4, ZLIB and LZMA. Each is offering increasingly strong compression at the cost of increasing compression- and decompression times for the same data. All three algorithms can be fine tuned via the compression level option, ranging from 1 to 9, where higher levels offer stronger compression. Compression times increase with higher compression levels as well, but decompression times are unaffected by design.

To compare the effect of all three compression algorithms, all IParticleContainer branches are read for the first 2000 events in the primary xAOD test file.

Table 1 gives a summary of the results, showing the impact on file size and reading speed (event rate and total walltime). All results are given as the mean value and its corresponding 1σ standard deviation of 10 repetitions to minimize the impact of system fluctuations.

LZ4 is the fastest in reading, but results in the largest files. LZMA provides much better compression, at the cost of significantly slower reading speeds. ZLIB is between these two, although the reading speeds are closer to LZ4 than to LZMA. The gain in file size reduction is in all three cases larger between level 1 and 5 than between level 5 and 9.

Although there are small fluctuations in reading performance, decompression times are almost constant. The exception is LZ4, where level 1 to 3 is different from 4 to 9, since the “high compression” mode is used for levels above 3.

Figure 1 shows the event throughput for all four tests file formats. Unfortunately, TOPQ1 and SUSY5 could not be compressed with LZ4, since they are produced with an older AthDerivation release that does not provide LZ4 compression yet. All three compression algorithms have the same characteristic behavior,

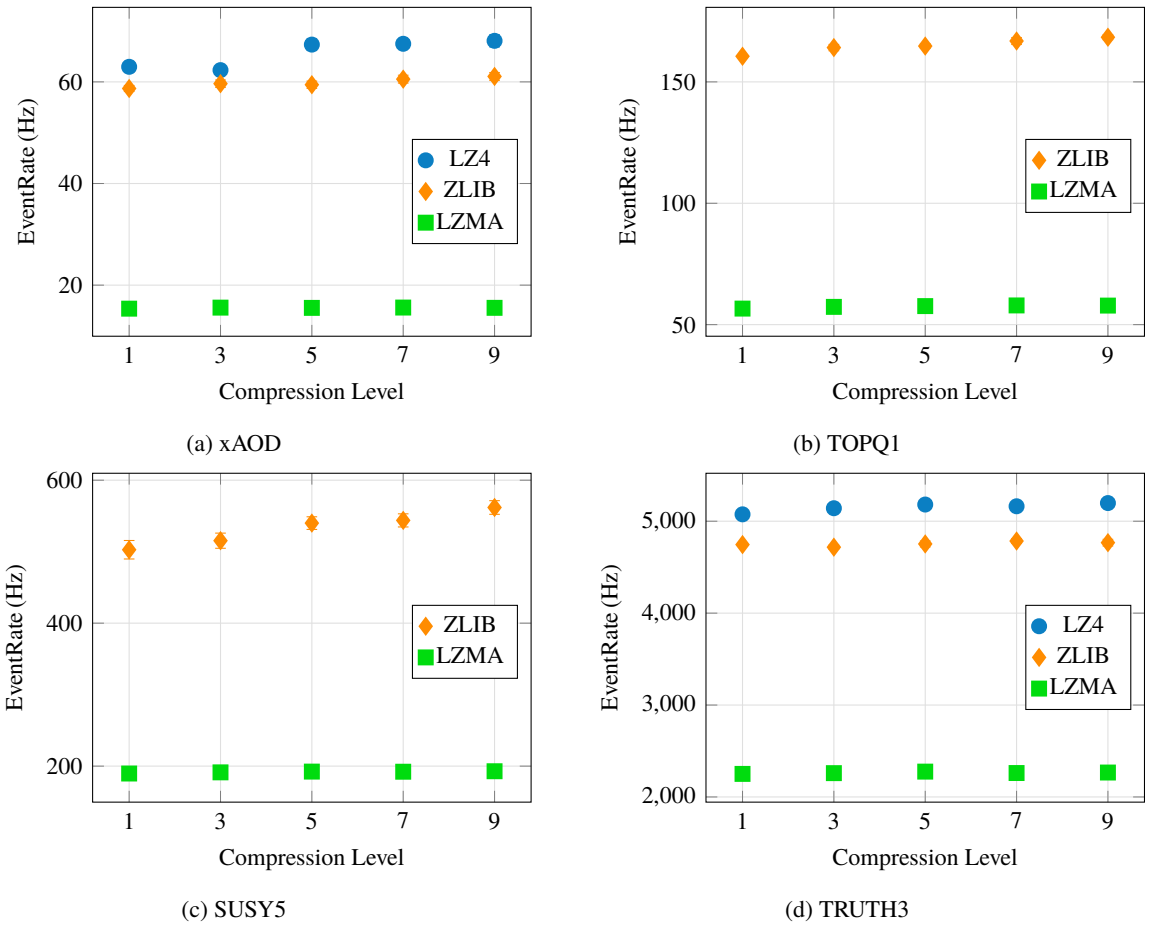
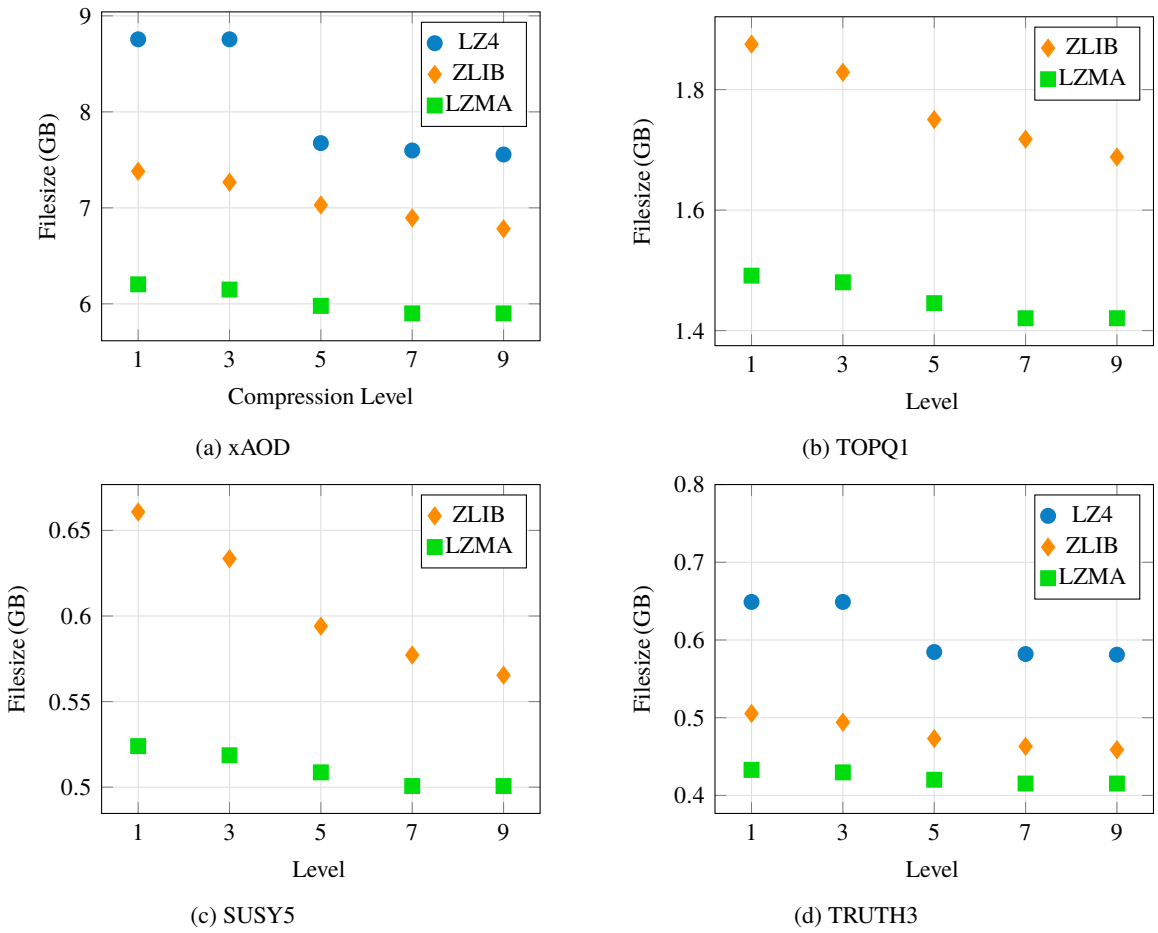


Figure 1: Test results of reading 2000 events from primary xAOD, TOPQ1, SUSY5 and TRUTH3 (100000 events) for different compression algorithms and levels. Values given as mean of 10 repetitions, error bars of 1σ mostly too small to be visible.

independent of how large the events are. Faster event throughput with smaller events is a direct result of streaming less data from disk.

Figure 2 shows the file sizes for all four tests formats and different compression algorithms/levels. Again, LZ4 is not available for TOPQ1 and SUSY5 derivations. For TRUTH3, ZLIB behaves more like LZMA with respect to compression ratios (Fig. 2(d)) and more like LZ4 with respect to event throughput (Fig. 1(d)), making it a very interesting choice for very small formats.

In general, LZ4 should be considered for cases where fast reading is much more important than file size reduction. This is likely the case for relatively small derivations that are produced centrally and are used by many different Analysers. LZMA should be considered for the opposite case, providing the strongest compression at the cost of much slower decompression speeds. High compression levels, 5 or 7, should be considered in all three cases. The gain of a compression level of 9 flattens out for LZMA and LZ4, which makes it only relevant for cases where file size reduction is the most important metric.



(a) xAOD (b) TOPQ1

(c) SUSY5 (d) TRUTH3

Figure 2: File sizes for primary xAOD, TOPQ1, SUSY5 and TRUTH3 for different compression algorithms and levels.

2.2 Autoflush

ROOT offers the *Autoflush* option that specifies how large a single compression unit of a given TTree should be. Setting it to 0 disables the feature, values < 0 set a number of Bytes and > 0 set a number of events that are compressed as one unit. The current ATLAS default is 100 events for primary xAODs and 10 MB for derivations.

Defining a reasonably sized compression unit is important, since compression algorithms work more efficiently with more data to compress. Too large units on the other hand can be inefficient if only parts of the data are read, since the whole unit has to be decompressed for a single access.

For this test we vary the Autoflush setting by number of events, ranging from 10 to 1000. This is done for all three compression algorithms and the four test file formats.

Figure 3 shows that all four formats have a low event throughput for too small event numbers (≤ 50). Large formats, like the primary xAOD or TOPQ1, reach a plateau at around 100 events, which validates the current default for xAODs. The small TRUTH3 format reaches the plateau at around ≥ 200 events. All

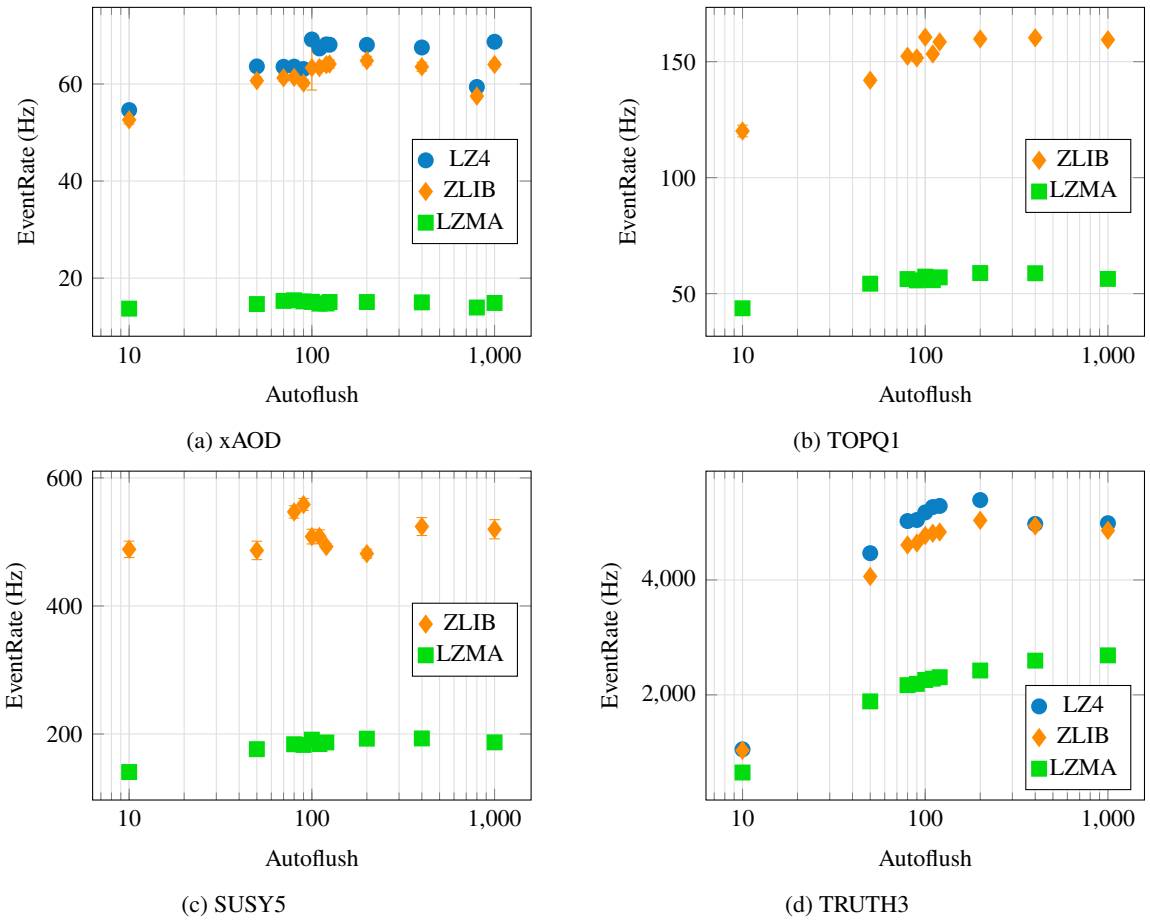


Figure 3: Test results of reading 2000 events from primary xAOD, TOPQ1, SUSY5 and TRUTH3 (100000 events) for different Autoflush sizes. Values given as mean of 10 repetitions, error bars of 1σ too small to be visible.

three compression algorithms show roughly the same scaling behavior with respect to the Autoflush setting, apart from already observed speed differences between the algorithms.

Figure 4 shows how the file sizes of all four test formats scale with the Autoflush option. For all formats, larger compression units result in smaller files, since the compression algorithms can work more efficiently. The smallest file sizes are reached at around 200 events, except for TRUTH3, which reaches its smallest file size at 1000 events. Note that ZLIB is very efficient at compressing the file format with very small events. As observed before, it behaves more like LZMA than LZ4, while still being efficient at reading.

Figure 5 shows that the memory footprint of decompressing the Autoflush units scales exponentially. Too small Autoflush values also result in a larger memory footprint if the event size is very small (cf. Fig 5(c) and 5(d)). In the case of TRUTH 3 (Figure 5(d)) the growing memory footprint for large Autoflush values is outside of the tested range. According to this, it is most important to not choose too large Autoflush values for file formats with large event sizes.

While 100 Events for primary xAOD is a good default, the current default of 10 MB for derivations is too small. Large derivations like TOPQ1 have roughly 50 events in one compression unit with a 10 MB Autoflush and would benefit from a larger setting. Very small formats like TRUTH3 benefit from a much larger Autoflush size, which is why the default for derivation formats should still be set in MB. 20 MB

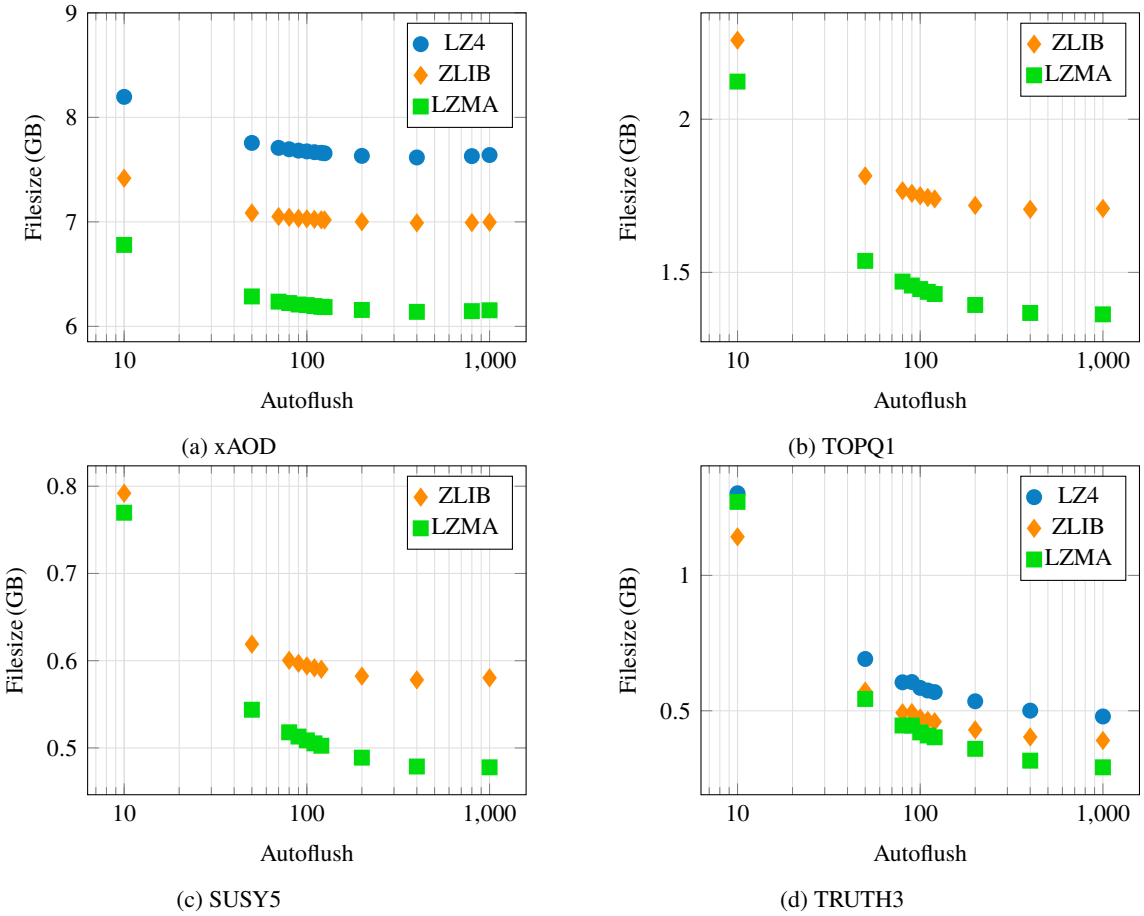


Figure 4: File sizes for primary xAOD, TOPQ1, SUSY5 and TRUTH3 for different Autoflush sizes.

would correspond to roughly 175 events in the TOPQ1 case (assuming 170 kB per event) and 4000 events for the TRUTH3 format.

2.3 Splitlevel

For TTree branches with sub-branches, e.g. events with multiple objects per event, the *Splitlevel* option sets the depth until which sub-branches will be stored next to their parent. With a Splitlevel of 1, all first level sub-branches are split off. A Splitlevel of 2 means that the subbranches of subbranches are also stored separately. This goes until 99, where all objects are stored at the root of the TTree.

Splitting the branches has an impact on serialization. For example, split sub-branches have individual buffers while reading instead of a single shared buffer.

In this test we split all branches of the (D)xAOD test files and measure the impact on event rate and file size.

Figure 6 shows the event rate for all four test formats with respect to different Splitlevels. There is no clear dependency for SUSY5 and TOPQ1. xAOD and TRUTH3 have the highest event rates for a Splitlevel of 0.

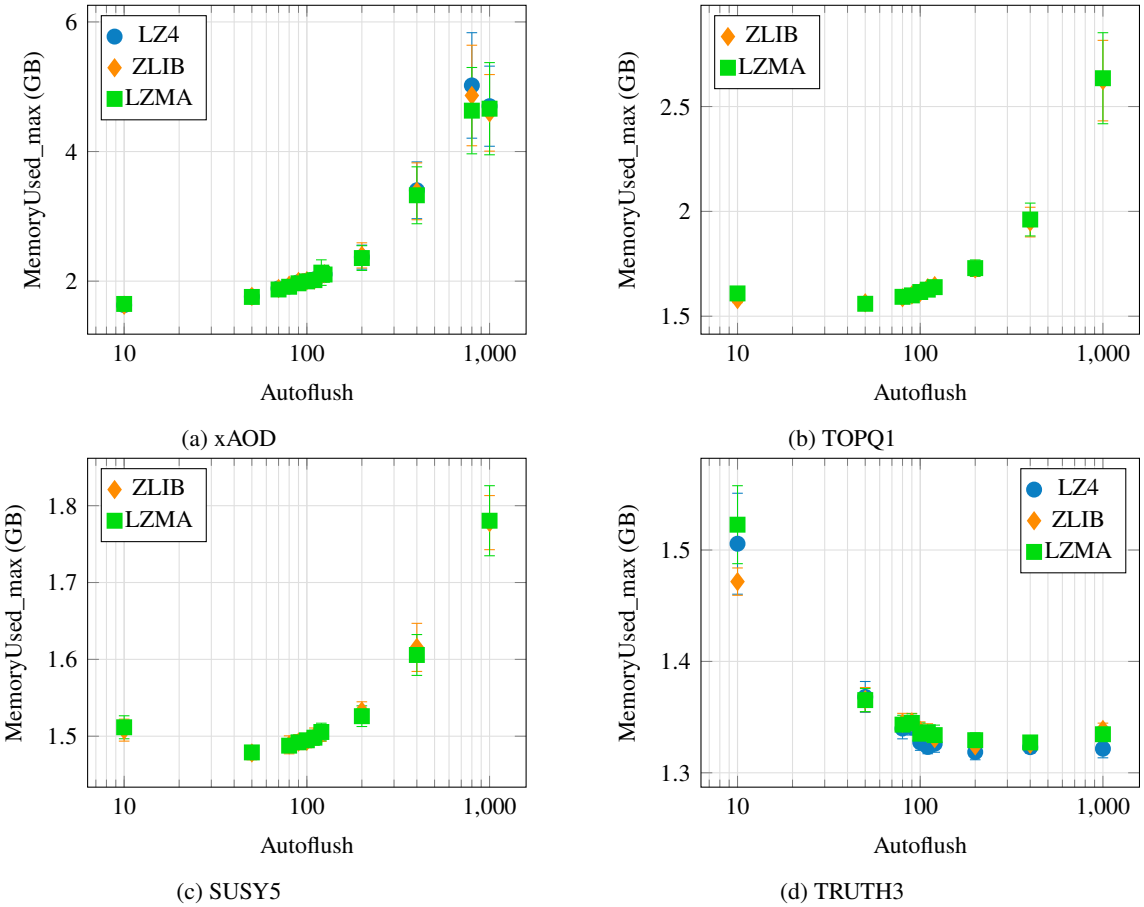


Figure 5: Memory usage while reading 2000 events from primary xAOD, TOPQ1, SUSY5 and TRUTH3 (100000 events) for different Autoflush sizes. Values given as mean of 10 repetitions, error is 1σ.

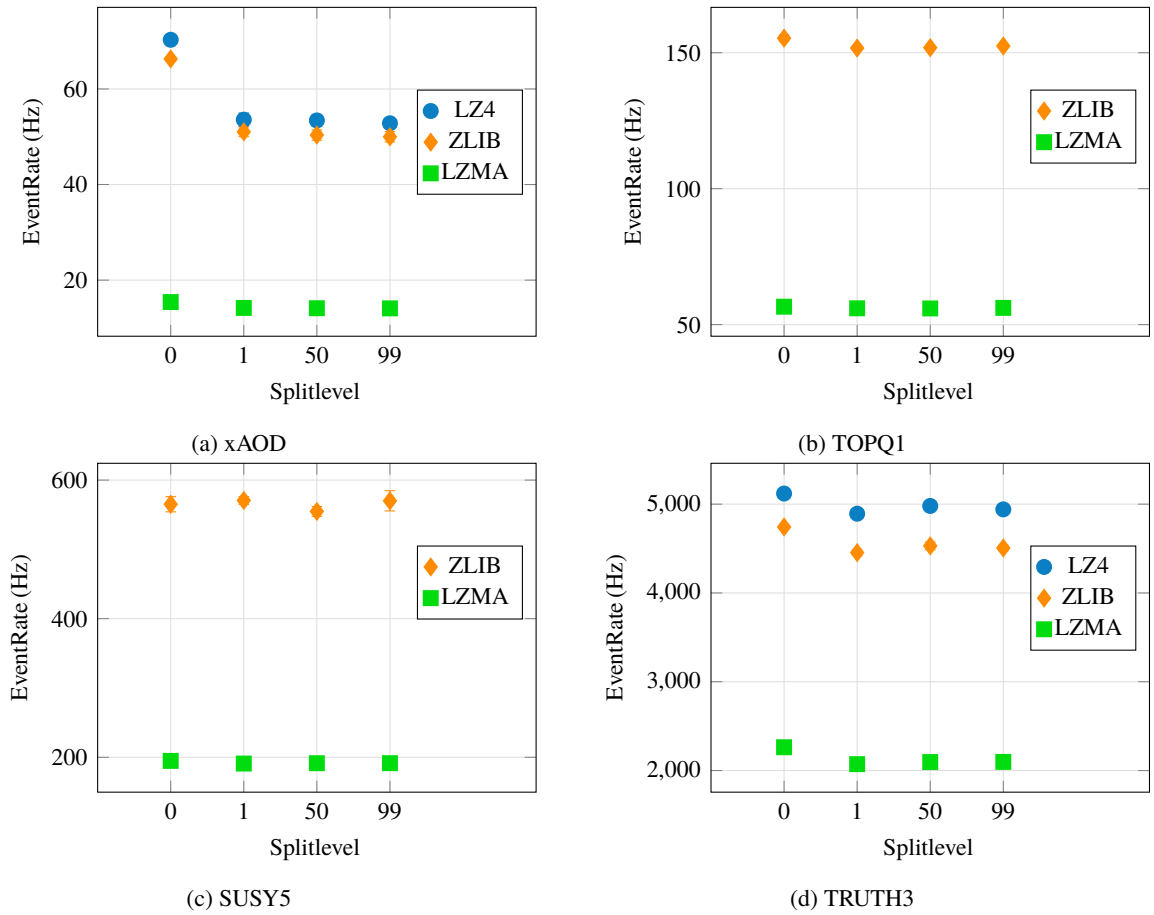
Figure 7 shows how the test file sizes change with respect to the Splitlevel. All formats are smallest for a Splitlevel of 0. This is most dominant for xAOD and TRUTH3. The file size is constant, but larger for Splitlevels > 0.

xAODs have a default Splitlevel of 0 for most branches. For derivations the default is 1 and following our results, we recommend to change the default for derivations to 0.

2.4 TTreeCache

ROOT provides a caching mechanism for reading from TTrees, called *TTreeCache*. It automatically recognizes which branches are accessed and pre-loads them when looping through all events. The TTreeCache size can be set to a value of -1, which corresponds to the exact size of the Autoflush setting. For the primary xAOD test file this is $\approx$ 100MB. The first N (variable) events are used by the TTreeCache to "learn" which branches should be pre-fetched while reading the following events.

To test the impact of the TTreeCache on reading speeds, we vary the cache size and measure the event throughput when reading from spinning disk. For primary xAOD, we also set the "learn size" to 0 and 100 (=Autoflush), to check if it has a strong influence on caching efficiency.



(a) xAOD (b) TOPQ1

(c) SUSY5 (d) TRUTH3

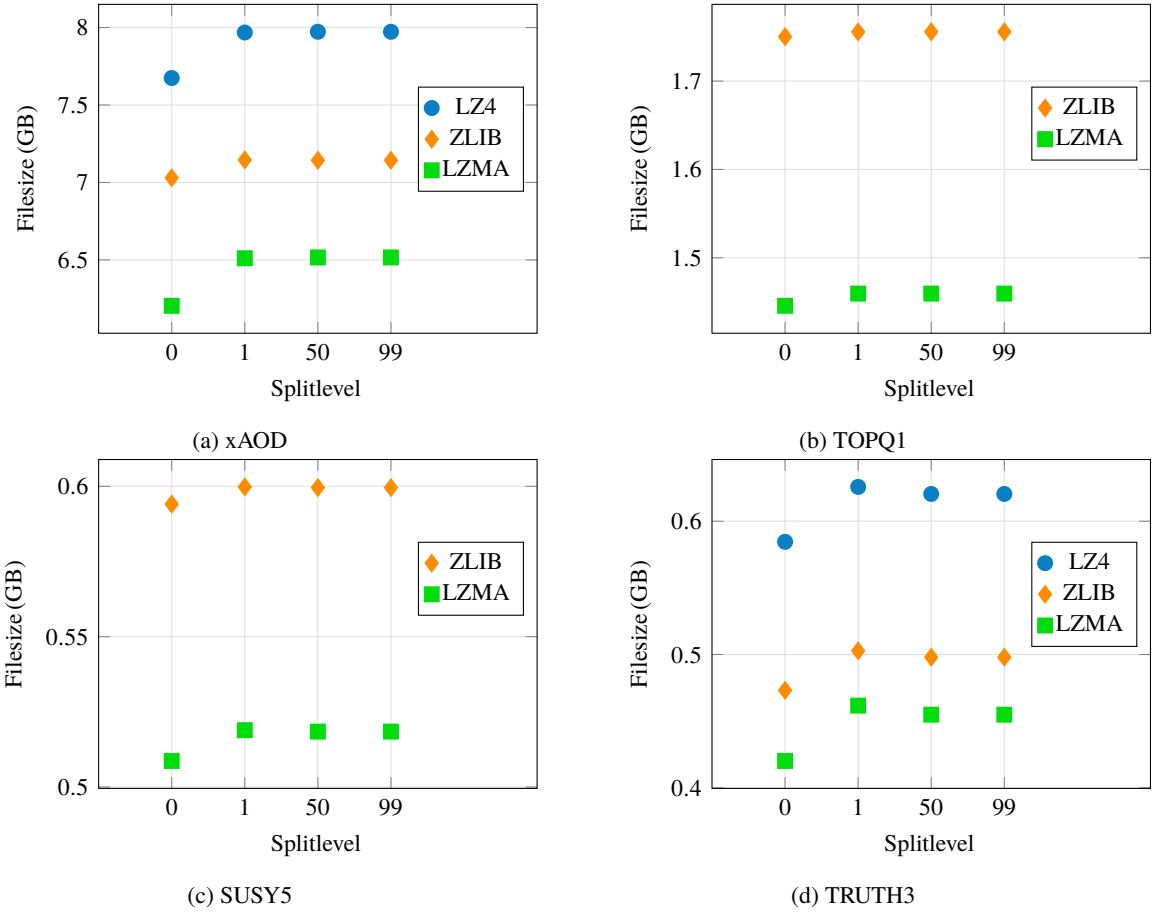
Figure 6: Test results of reading 2000 events from primary xAOD, TOPQ1, SUSY5, TRUTH3 (100000 events) for different splitlevels. Values given as mean of 10 repetitions, error bars of 1σ too small to be visible.

Figure 8 shows that the TTreeCache can improve reading performance up to a certain point. For too small caches, e.g. below 50 MB, which corresponds to about half the size of the ATLAS default Autoflush setting for primary xAODs, the event throughput drops (cf. Fig 8(a)). Changing the number of events used for learning does not have a big impact on cache performance. The derivation test formats are read efficiently even with small TTreeCaches.

Figure 9 shows that the number of reading operations is significantly larger for very small cache sizes. This is why a TTreeCache is most important when reading from sources with high latencies, e.g. from a network storage system.

In a small test we found that a TTreeCache also removes the issue of reduced reading rates while reading branches in a different order from how they are stored on disk.

To conclude, for all (D)xAODs we recommend to use the TTreeCache with the -1 option.



(a) xAOD (b) TOPQ1

(c) SUSY5 (d) TRUTH3

Figure 7: Filesize of primary xAOD, TOPQ1, SUSY5, TRUTH3 for different splitlevels.

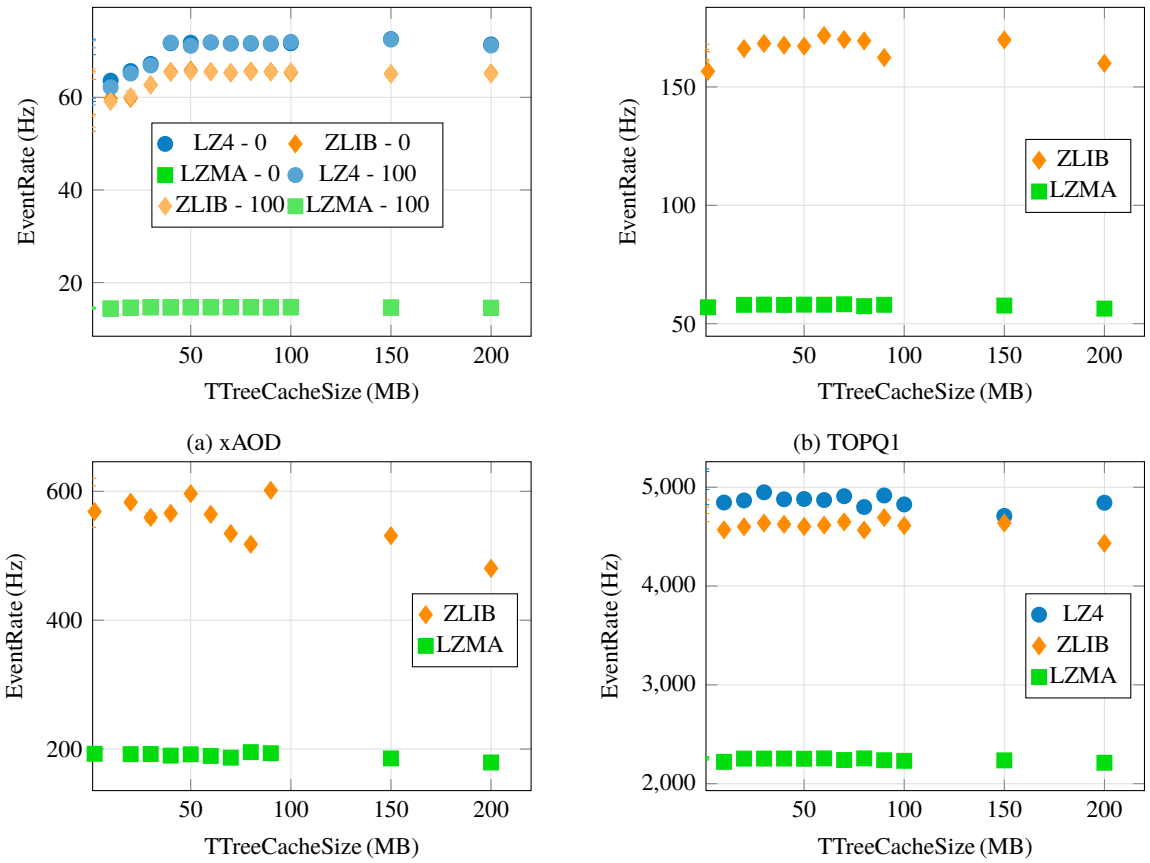
2.5 ROOT Multithreaded Branch Decompression

ROOT provides the option to enable implicit multithreading, in which branches read from a TTree are decompressed in parallel.

In a ROOT standalone version of the reading tests, all IParticleContainer branches are read from the (D)xAOD input files, with a varying number of threads.

Figure 10 shows that reading LZMA compressed files do benefit from parallel branch decompression. Reading ZLIB and LZ4 compressed files benefit from parallel decompression as well, but event throughput decreases for higher number of threads. There is no improvement for more than 4 threads for primary xAOD (cf. Fig 10(a)), which corresponds to the number of physical cores of the machine. Either this process does not benefit from hyper threading or parallel branch decompression of xAODs has a bad thread-scaling behavior. For primary xAOD, all three compressed test files have $\approx$ 25% better event throughput, when using 4 threads to decompress the data.

Parallel branch decompression for smaller derivations (Figures 10(b) to 10(d)) is only beneficial for LZMA compressed files. All ZLIB and LZ4 compressed DxAOD files lose performance if more than one thread is used.



(a) xAOD

(b) TOPQ1

(c) SUSY5

(d) TRUTH3

Figure 8: Test results of reading 2000 events from primary xAOD, TOPQ1, SUSY5 and TRUTH3 (100000 events) for different TTreeCache sizes. Values given as mean of 3 repetitions, error bars of 1σ too small to be visible. Figure 8(a) also shows results for a "learn entry" size of 0 and 100.

If decompression is a measurable or even limiting factor for a given ATLAS workflow, it can be beneficial to investigate ROOTs parallel branch decompression further. The xAOD reading code needs a revision with respect to multithreading, since occasional crashes and deadlocks have been observed.

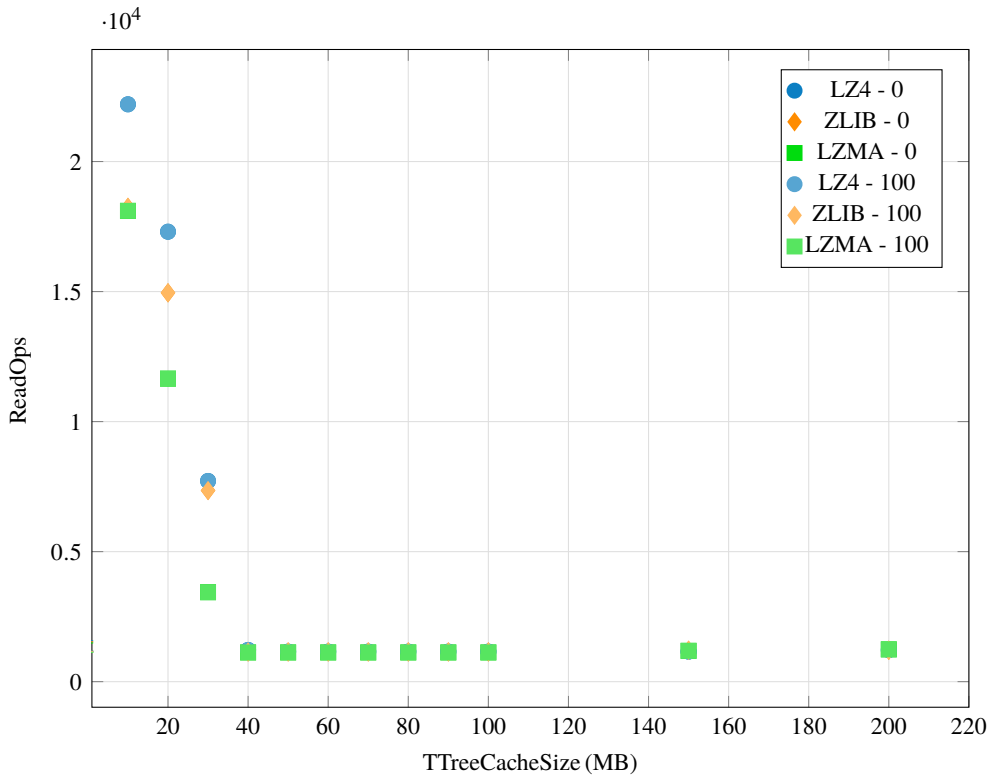
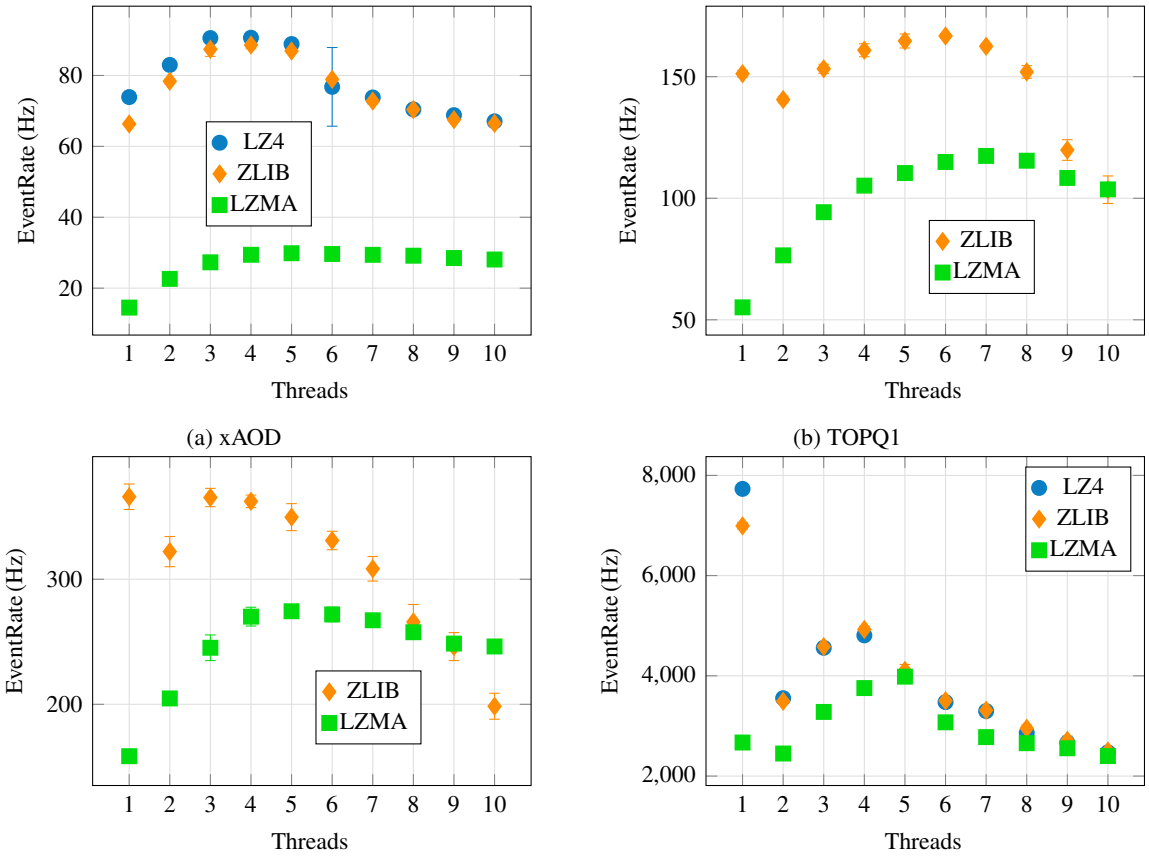


Figure 9: Test results of reading 2000 events from primary xAOD for different TTreeCache sizes. Values given as mean of 3 repetitions, error bars of 1σ too small to be visible. The number given in the legend shows the used TTreeCache learn entries for that measurement.



(a) xAOD

(b) TOPQ1

(c) SUSY5

(d) TRUTH3

Figure 10: Test results of reading 2000 events from primary xAOD, TOPQ1, SUSY5, TRUTH3 (100000 events) in ROOT standalone implementation with a different number of threads. Values given as mean of 10 repetitions, error is 1σ.

3 Conclusion

During our studies we found multiple approaches to extend and improve our tests. These ideas are collected here to document useful starting points for future studies.

There are modern compression algorithms that potentially perform better than LZ4, ZLIB and LZMA. Some are optimized implementations of the existing algorithms, e.g. cloudflare ZLIB[6], LZHAM[7], vectorized LZ4 by Intel[8]. Two new approaches are proio – varint[9] and zstd[10], where both could lead to better compression ratios while maintaining/improving reading speeds. In order to test these algorithms, they have to be implemented in ROOT itself, which possibly requires joined effort between ATLAS and the ROOT team.

Further studies with branch specific settings (e.g. Autoflush, Splitlevel) should be done for the largest N (e.g. 10) branches, which could lead to further compression and reading speed improvements.

Tests with varying TTreeCache sizes while reading from EOS and HDD could lead to slightly better caching behavior, when reading from network resources.

All of the current tests ran on the same hardware. Future tests could try to isolate hardware dependencies, such that optimization towards the "average" hardware used in ATLAS are possible.

We did not profile AthDerivation jobs during in our studies. Such measurements would be a test for reading xAODs and *writing* derivations at the same time. This approach could give additional information about ideal compression algorithms, levels and Autoflush settings for xAODs and DxAODs.

It would be interesting to develop a cost model of available CPU time and disk space to find an optimal balance point between occupied disk space and necessary CPU time for decompression for files used in common ATLAS workflows (Reconstruction, Derivation, Analysis). Such a model would weigh CPU time to (de-)compress a file against the time it occupies a certain amount of disk space, including how often it is used and possibly reproduced.

3.1 Summarized Recommendations

This study finds that LZ4 compression is best suited for workflows that require fast reading, while LZMA compression is best to minimize file sizes. ZLIB offers an intermediate solution to fine tune the balance between these two metrics and its performing best for formats with very small event sizes.

The default Autoflush of 100 events is a good setting for primary xAODs. The current default of 10 MB for derivations is too small for larger derivations (e.g. with 170 kB per event) and it should be doubled to 20 MB.

The default Splitlevel of 1 for derivations should be changed to 0 to reduce file sizes and slightly improve reading rates.

While reading, a TTreeCache set to -1 should be used to set its size to be equal to the Autoflush. This is most important if files are read through a network or other high latency file systems.

ROOTs parallel branch decompression is very beneficial for large (D)xAOD formats in combination with LZMA. In some situations even ZLIB and LZ4 compressed input files can be faster decompressed with multiple threads.

References

[1] ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST **3** (2008) S08003.

[2] J. Elmsheuser et al., *Evolution of the ATLAS analysis model for Run-3 and prospects for HL-LHC*, (2019), URL: https://cds.cern.ch/record/2696416.

[3] J. Elmsheuser et al., *Evolution of the ATLAS analysis model for Run-3 and prospects for HL-LHC*, tech. rep. ATL-SOFT-PROC-2020-002, CERN, 2020, URL: https://cds.cern.ch/record/2708664.

[4] A. Buckley et al., *Implementation of the ATLAS Run 2 event data model*, tech. rep. ATL-SOFT-PROC-2015-003. 7, CERN, 2015, URL: https://cds.cern.ch/record/2014150.

[5] *ROOT TTree Documentation*, URL: https://root.cern.ch/doc/master/classTTree.html (visited on 18/12/2018).

[6] *Github: zlib*, URL: https://github.com/cloudflare/zlib (visited on 26/03/2020).

[7] *Github: LZHAM codec*, URL: https://github.com/richgel999/lzham_codec (visited on 26/03/2020).

[8] *Building a faster LZ4 with Intel® Integrated Performance Primitives*, URL: https://software.intel.com/en-us/articles/building-a-faster-lz4-with-intel-integrated-performance-primitives (visited on 26/03/2020).

[9] D. Blyth, J. Alcaraz, S. Binet and S. Chekanov, *ProIO: An event-based I/O stream format for protobuf messages*, Computer Physics Communications **241** (2019) 98, ISSN: 0010-4655, URL: http://dx.doi.org/10.1016/j.cpc.2019.03.018.

[10] *Github: zstd*, URL: https://github.com/facebook/zstd (visited on 26/03/2020).