Journal of Physics: Conference Series

PAPER • OPEN ACCESS

Online data compression in the ALICE O^2 facility

To cite this article: Matthias Richter and ALICE Collaboration 2017 *J. Phys.: Conf. Ser.* **898** 032049

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 032049 doi:10.1088/1742-6596/898/3/032049

Online data compression in the ALICE O^2 facility

Matthias Richter
for the ALICE collaboration

Department of Physics, University of Oslo, and
CERN - European Organization for Nuclear Research

E-mail: matthias.richter@cern.ch

Abstract. The ALICE Collaboration and the ALICE O^2 project have carried out detailed studies for a new online computing facility planned to be deployed for Run 3 of the Large Hadron Collider (LHC) at CERN. Some of the main aspects of the data handling concept are partial reconstruction of raw data organized in so called time frames, and based on that information reduction of the data rate without significant loss in the physics information.

A production solution for data compression has been in operation for the ALICE Time Projection Chamber (TPC) in the ALICE High Level Trigger online system since 2011. The solution is based on reconstruction of space points from raw data. These so called clusters are the input for reconstruction of particle trajectories. Clusters are stored instead of raw data after a transformation of required parameters into an optimized format and subsequent lossless data compression techniques. With this approach, a reduction of 4.4 has been achieved on average.

For Run 3, not only a significantly higher reduction is required but also improvements in the implementation of the actual algorithms. The innermost operations of the processing loop effectively need to be called up to $O(10^{11})/s$ to cope with the data rate. This can only be achieved in a parallel scheme and makes these operations candidates for optimization. The potential of template programming and static dispatch in a polymorphic implementation has been studied as an alternative to the commonly used dynamic dispatch at runtime.

In this contribution we report on the development of a specific programming technique to efficiently combine compile time and runtime domains and present results for the speedup of the algorithm.

1. Introduction

The ALICE experiment at CERN is going to be significantly upgraded before the Run 3 period of the Large Hadron Collider (LHC) starting after 2020. In particular there will be a new readout concept to inspect all collisions delivered by the accelerator. The sustained data rate delivered by the detectors will be more than 3 TByte/s. This requires a large fraction of the data processing and reconstruction to be carried out already online. The presented work is in the context of the ALICE O^2 project [1], which is developing a combined online and offline computing system for the upgraded ALICE detector.

In general, there is an increasing interest in online data reconstruction in future experiments and upgrades, and online data reduction becomes more and more important with increasing data volume. Lossless compression techniques allow to restore the original data from the compressed ones, but have a limited potential. Lossy techniques can reach higher data reduction factors because some of the information is discarded. Depending on the application, the information loss can be such that there is only a negligible impact to the final product even though the

original data can not be restored. The actual processing flow for reduction of data involves often multiple steps and the combination of different techniques. In the case of particle detectors, lossy techniques include pre-processing of raw data to extract relevant information for reconstruction of particle trajectories with sufficient precision.

For a detector system like the ALICE Time Projection Chamber (TPC), lossless data compression techniques applied directly to raw data allow for a data reduction factor of about 2 at most. Higher reduction can be achieved by using information about the data itself. This allows for adaptive selection of data and adjustment of precision. In the current implementation of data compression for the ALICE TPC, spacepoint properties, so called clusters, are reconstructed from raw data and stored with adjusted precision and additional Huffman compression, which is a variant of lossless entropy encoding.

The number of individual operations naturally correlates with the number of runtime data objects to be compressed. Furthermore, each object might have multiple parameters. A flexible implementation often builds upon a class structure with virtual inheritance. This comes at the price of dynamic dispatch at runtime and has a noticeable impact on the performance when it comes to heavily used parts of the program. The alternative would be to either drop flexibility or to switch to a static dispatch scheme allowing for compiler optimization. The latter option is the focus of this paper.

The general data compression strategy is introduced in section 2 together with constraints and requirements. Based on these considerations and the programming techniques introduced in section 3, a generic prototype has been developed. Sections 4 and 5 describe the development in more detail. Benchmark measurements in section 6 conclude the paper.

2. Overall data compression scheme

2.1. Current solution and experience

The ALICE experiment is using data rate reduction in an online production system since 2011, with the primary goal to reduce temporary disk space and permanent storage. Online reconstruction and compression of data from the ALICE TPC is carried out inside the ALICE High Level Trigger [2]. The scheme is based on three steps: (i) reconstruction of space point properties from raw data, (ii) transformation to reduced precision with negligible impact to physics, and (iii) lossless Huffman encoding. The last step is embedded in the writing of data to memory.

A common situation for this process is depicted in figure 1. A sequence of runtime objects, with multiple parameters and identical layout each, has to be stored in memory, on disk, or has to be transported over network. Different strategies and algorithms can be used to map values of the runtime objects to memory buffer. With the simplest approach, each value is stored in the buffer in **linear storage**. The fixed bit length follows the byte alignment. This method allows direct access to data members, however it introduces unused bits to keep the alignment. The **stream deflater** approach uses a specific bit length for every parameter but skips all the alignment bits. The number of required bits for every parameter depends on range and precision of the variable. Finally, applying a **Codec**, i.e. a lossless coding algorithm, each value is represented in the buffer by a variable number of bits. The two latter methods both require to first decode and/or shift a data element before using it for calculation. Furthermore, introducing a coding step often makes direct access of data impossible and requires decoding from the beginning of the sequence or a specific synchronization marker.

2.2. Requirements and boundary conditions for ALICE O^2

In ALICE O^2, data compression will be applied not only for TPC data but for several other detectors. Furthermore, memory consumption in a worker machine and network traffic when shipping data between machines or even data centers need to be optimized.

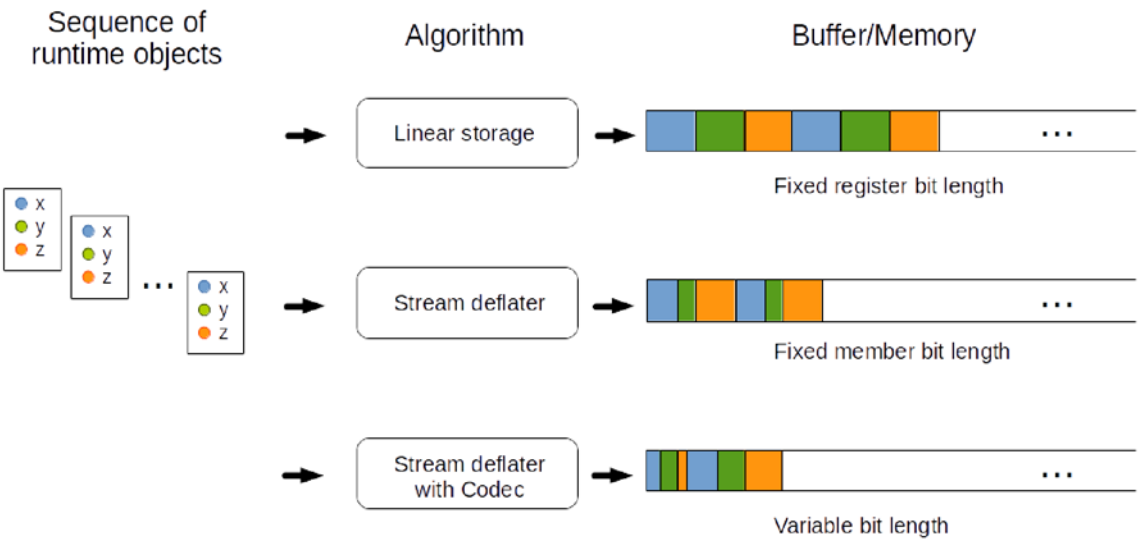


Figure 1. Schematic illustration of the processing of a sequence of runtime objects with multiple parameters. Each algorithm results in a specific storage pattern of data, shown in specific colors for individual parameters. The figure also illustrates the effective reduction of data if adjusted or variable bit length is used.

While 1st-level reconstruction of raw data with required precision is detector dependent, the lossless entropy coding and storage in optimized format are suited for a generic framework. Because of the wide application, we aim for a flexible, polymorphic solution.
The general boundary condition can be summarized as such:

- Sequence of runtime objects of identical type needs to be stored in data stream,
- Objects can have multiple parameters with individual characteristics and probability models,
- Framework has to support multiple codec types, e.g Huffman and arithmetic coding.

Another important aspect comes from the readout scheme. Data will be partially read out in a continuous stream and organized in so called timeframes. In the presented work, the length of timeframes has been set to 10 ms. Note that the actual value does not have an impact to the conclusion of this paper. Within such a timeframe there will be $\sim 250 \times 10^6$ clusters with 7 parameters each. Consequently, the innermost functions of the processing loop are called $O(10^9)$ per timeframe, thus requiring a full optimization for this part of the code. Even a minor performance optimization allows for big effects.

3. Programming techniques

In this section we introduce programming techniques used in this work.

Compiler optimization: During the translation of a program written in some programming language into an executable program, the compiler can apply optimization to minimize or maximize some feature of the program. The most common optimization is minimization of execution time, which is also the focus of the presented work.

In order to carry out powerfull and efficient code optimization a compiler makes use of the information it has about the program, e.g. data structures, and the architecture to run it on. The more information is available to the compiler the better can be the result of the optimization.

The C++ compiler provides different optimization levels which are controlled by the `-O<n>` switch, where `<n>` specifies the optimization level between 0 (no optimization) and 3 (highest

optimization). Measurements throughout this paper are marked with the optimization level, e.g. `-O2`. The individual optimization techniques used by the compiler are not discussed.

Polymorphism denotes a concept in computing which allows to decide which piece of code is executed based on the type of something, e.g. the type of a runtime object. The executable program dispatches to different branches of executable code. There are two options to apply polymorphism in programming:

- **Runtime polymorphism:** the actual binding of the type of an object is deferred until runtime. This concept is usually realized using classes with virtual inheritance and virtual functions. The pointer to a runtime object can be of multiple types in the program, dynamic dispatch at runtime checks every time a virtual function is called which branch needs to be executed.
- **Static polymorphism:** is completely resolved at compile time. This eliminates dynamic binding and results into more efficient implementations. There are some notable features:
  · type checks at compile time,
  · selection among code branches which would not compile in all cases,
  · facilitates implementation of generic algorithms,
  · allows generic handling of multiple types,
  · compiler has a lot more information for code optimization.

Template programming is a technique to define a functional piece of a program independently of the type of data to be processed. It is the prime technique to realize static polymorphism. The concept involves two operations, (i) the definition of a template, and (ii) its instantiation for a set of types by the compiler.

Meta programming makes use of template programming to build programs that are executed at compile time. This technique is used to calculate derived parameters and evaluate sets of data types from information available at compile time. The *boost Metaprogramming Library (MPL)* provides a powerfull toolbox and is used in the prototype development.

4. Prototype implementation

4.1. Task analysis and decomposition

A policy-based design strategy is applied, which means to decompose processing into small entities. Policies in programming are small functional entities (classes) which take care of separate behavioral or structural aspects. Complex entities are assembled from several small policies. The application is decomposed into the following policies which are discussed in more detail throughout this section:

- Input policy,
- Alphabet,
- Probability Model,
- Codec (Coder/Decoder), and
- Output policy.

The input and output policies are technical aspects which are outside the scope of this work. The prime fact for the design is the availability of the information at compile time or runtime. While the alphabet and the codec algorithm are fixed at compile time, the probability model is runtime dependent. This is an important aspect for the compiler as no optimization can be applied based on runtime dependent information. The challenge is that static polymorphism is based on data types that can not hold a runtime state. In contrast to that a runtime object needs to hold the state of runtime information. Static polymorphism can still be applied if the runtime object follows the type definition at compile time as shown in section 5.

4.2. Alphabet implementation

An alphabet is a set of symbols to be treated by a data compression algorithm. It is an intrinsic feature of the data and fixed at compile time. As such, template programming can be used to define individual alphabets. Very often, an alphabet is a set of contiguous integral numbers of a specific data type **T** between a minimum and maximum value. In template notation such an alphabet can be defined like

```
template<typename T, T _min, T _max, typename NameT> class ContiguousAlphabet {...};
```

The template argument `NameT` is not important for the definition of the alphabet itself but allows for named entries in the list. Specializations for distinct cases are defined, e.g. alphabet from 0 to some maximum, or alphabet for a n-bit field. With such definitions, all key parameters of the alphabet are known at compile time and are exploited by the compiler optimization.

As mentioned before, runtime objects have multiple parameters with individual probability models. The parameters need to be stored in a continuous data stream. To fit into this concept, *sets* of parameter types are defined at compile time. The framework makes use of the *Boost Metaprogramming Library* to define such sets of data types as illustrated in the following source code example. Note that a sequence of individual arguments of type `char` must be used to create the template argument `NameT`. String literals, such as `"row"`, can not be used as valid template arguments.

```
typedef boost::mpl::vector<
  BitRangeContiguousAlphabet<int16_t,     6 , boost::mpl::string < 'r','o','w'> >,
  ContiguousAlphabet<int16_t, -16384, 16383 , boost::mpl::string < 'p','a','d'> >,
  ContiguousAlphabet<int16_t, -32768, 32767 , boost::mpl::string < 't','i','m','e'> >,
  BitRangeContiguousAlphabet<int16_t,    16 , boost::mpl::string < 'q','t','o','t'> >,
  ZeroBoundContiguousAlphabet<int16_t, 1000 , boost::mpl::string < 'q','m','a','x'> >
  > tpccluster_parameter;
```

4.3. Probability model

The probability model describes the statistical occurrence of symbols of an alphabet. Statistics is gathered from a runtime data sample and the model is usually not fixed at compile time. This is in conflict with a concept based on static polymorphism and compile time optimization. A type-safe *runtime container* has been designed as part of this work and is introduced in section 5. This interface between compile time and runtime domains allows a combination of compile time type definitions and runtime objects.

4.4. Codec

The task of the codec is to translate the sequence of incoming symbols into a sequence of output bits in a deterministic procedure. The codec makes use of the alphabet and the probability model. The algorithm of the codec is fixed at compile time. The first prototype uses Huffman coding, the framework allows to plug in other algorithms like e.g. arithmetic coding.

5. The *runtime container* implementation

As explained in the previous section, the individual alphabets are defined in a compile time list of template implementations. The *runtime container* as it has been developed allows to associate runtime data with elements of the compile time list. Still, the compiler can perform static type casts at compile time and apply optimization rather than dispatching at runtime.

As sketched in figure 2, based on the C++ *mixin class* design pattern [3], a composite structure with the ability of static dispatch is assembled. However, rather than having different components combined in the mixin class, a recursive definition of *identical* templates is used.

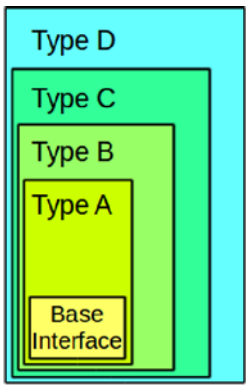
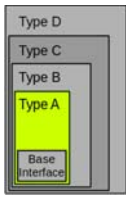


Figure 2. Schematic view of the *runtime container*. It comprises a recursive, nested definition of types where each type includes the previous ones. It follows the C++ *mixin class* pattern and assembles identical templates each wrapping one data type from the list and defining a member holding the runtime data. The compiler can walk through the container levels by static type cast. Virtual inheritance and interfaces can be eliminated from the design.

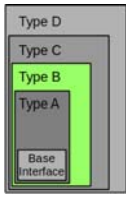
Each data type level is accessible through static cast at compile time.

```
static_cast<level&>(containerobject).doSomething();
```

The operation is either implemented as specific method in the *runtime container* or passed to specific level through a *functor*. Figure 3 illustrates the access to data members of multiple types in the individual levels of the container.



`static_cast<TypeA>(object)`



`static_cast<TypeB>(object)`

Figure 3. The *runtime container* wraps objects of different data types and allows generic, 100% type-safe access to multiple data types. Individual levels of the container can be accessed by type casts. `static_cast` is evaluated already at compile time and provides the compiler with the necessary information for code optimization.

There are three possible access patterns which have been studied in this paper. All access patterns are based on static dispatch.

(i) Recursive: generic recursive set of compiled functions from a meta function template, the requested level is specified as parameter.

(ii) Loop unrolling: explicit loop unrolling in a generic dispatcher function of the *runtime container*, eliminates recursive function calls by direct cast to required level in a runtime switch.

(iii) Bulk access: Specific method for the runtime object to be processed. It makes use of the data structure layout to apply direct type cast to individual levels, but without additional runtime switch.

The different options result in different optimization results and different timing characteristics. It has to be noted that options (i) and (ii) are generic methods of the *runtime container*, while option (iii) is a specific method which makes use of the knowledge about the data structure layout.

Figure 4 shows time measurements for the two generic access patterns. We measure the time needed per operation in a loop of up to $\sim 10^9$ operations. Each operation is an access

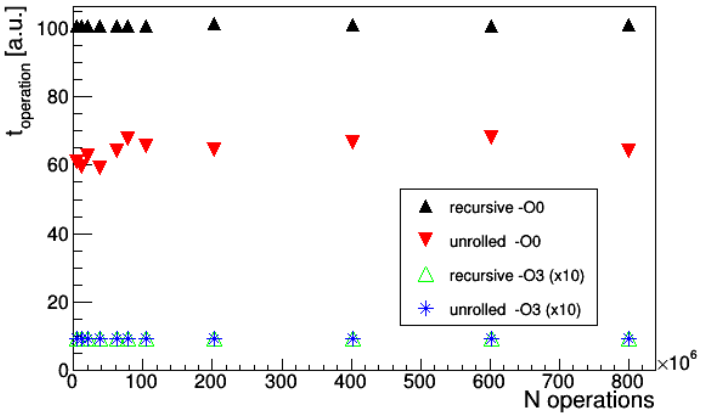


Figure 4. Benchmark measurements for two access patterns of the *runtime container* in arbitrary units. The unoptimized version `-O0` shows a clear difference between the generic recursive and explicitly unrolled access. For compiler optimization `-O3`, the two access patterns give the same result, one order of magnitude faster than the unoptimized recursive version. This illustrates the potential of the compiler optimization.

to the specific member of the runtime container level based on a random number. It can be concluded that the technique allows the compiler to automatically create efficient unrolled code. This would not have been possible with dynamic runtime dispatch. It clearly shows that the explicit unrolling of code, which comes along with the loss of flexibility, can be dropped from the implementation.

6. Prototype measurements

The *runtime container* has been added as a new programming tool to the ALICE O^2 framework. The prototype implementation uses Huffman coding as example operation. Performance of the prototype has been studied in the three modes *recursive*, *unrolled*, and *bulk operation* with different compiler optimization levels 0 to 3. The results are shown in figure 5

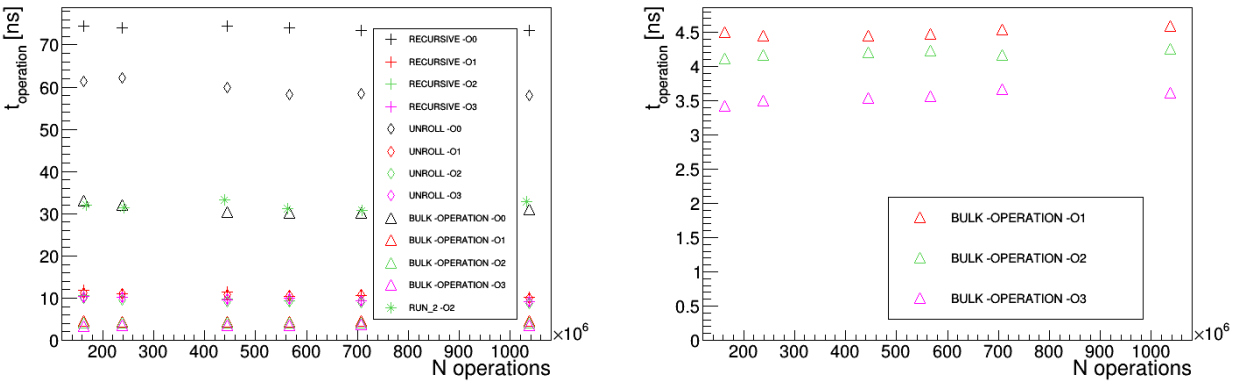


Figure 5. Average time per operation for different access patterns and compiler optimization levels. Left: all measurements, Right: Comparison of compiler optimization for access pattern `BULK_OPERATION` as the fastest option.

No significant dependence on the number of operations is observed. The bulk operation is the fastest option. Again, there is no significant difference between generic recursive and unrolled versions if compiler optimization is active.

Figure 6 shows a comparison of measurements for compiler optimization `-O2`. The current implementation of Huffman compression in AliRoot for ALICE Run 1 and 2 uses runtime polymorphism, base class interfaces and virtual inheritance. Using static polymorphism in both

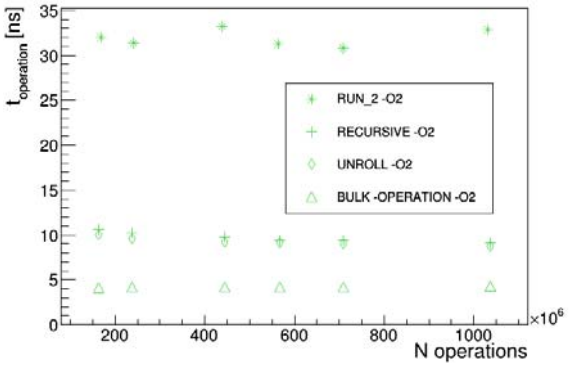


Figure 6. Comparison of current implementation based on runtime polymorphism with the ALICE O2 prototype for compiler optimization `-O2`.

generic recursive method and specializations in the ALICE O^2 framework allows for a significant speedup of the processing

Table 1 shows the average processing time per operation for different implementations in compiler optimization `-O2`. We conclude with the observation that the number of cores needed for executing the sequence of Huffman compression can be reduced by a factor of 3 to 8. This leads to a better utilization of computing resources just by applying an alternative programming technique like meta programming and utilization of static polymorphism.

Table 1. Average processing time per operation for the RUN2 implementation and different options in the prototype. The minimum number of cores to cope with the data rate at $O(10^9)$ operations in 10 ms timeframes is given as a rough concluding estimation.

	RUN2	RECURSIVE	UNROLL	BULK-OPERATION
Avrg. processing time	32 ns	9.7 ns	9.2 ns	4.1 ns
Min. number of cores	>3200	> 1000	> 1000	> 400

7. Summary

In order to cope with the amount of data in LHC Run 3, the ALICE experiment will apply data compression for both intermediate data and permanent storage. Depending on the amount of runtime objects to be compressed and stored or transported in optimized format, the number of operations to be applied is of the order of 10^{11}/s. An efficient implementation is thus a crucial aspect with respect to efficient resource utilization. We have shown in this paper the potential of meta programming and compile time optimization. A new tool has been developed to associate runtime data with statically dispatched code execution branches. This enhances compile time optimization. A speedup of a factor of 3 to 8 has been measured depending on the applied access pattern to data. This results in a significant reduction of necessary computing resources.

References

[1] The ALICE Collaboration. Technical Design Report for the Upgrade of the Online-Offline Computing System. ALICE-TDR-019, 2015. CERN-LHCC-2015-006.
[2] Richter M *et al*, The ALICE Collaboration, Data compression in ALICE by on-line track reconstruction and space point analysis 2012 Journal of Physics: Conference Series **396**
[3] Smaragdakis Y and Batory D, Mixin-Based Programming in C++, 2001 *Series Lecture Notes in Computer Science* **2177** 164-178