Online Data Compression in the ALICE O2 facility

Matthias Richter
for the ALICE collaboration

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

22nd International Conference on Computing in High Energy and Nuclear Physics, Hosted by SLAC and LBNL, Fall 2016
Outline

- Data Compression in the ALICE Upgrade
- Prototype of a generic Data Compression Framework
- Programming techniques for a generic solution.
- Prototype in operation
- Summary
Data Compression in the ALICE Upgrade

A challenging goal ...

Inspect all Pb-Pb collisions at min bias rate of 50 kHz to provide access to rare physics probes. Continuous processing of data stream of 3.4 TByte/s.

- ALICE run 3 readout-electronics partly based on continuous readout; e.g. the Time Projection Chamber (TPC) is read out continuously.
- TPC readout data rate > 3 TByte/s @ 50 kHz min bias Pb-Pb
  required data reduction factor: $\sim 40$

Assumptions for the Data Compression Framework

- Data are organized in so called Time Frames (TF) of fixed length, currently set to 10 $ms$
- TPC clusters (spacepoints) per TF: $\sim 250 \times 10^6$, 7 parameters each;
  10 bytes/cluster in optimized bitstream; 24 bytes in linear storage
  up to 250 $\times 10^6$ TPC clusters need to be processed per TF;
Overall Data Reduction Scheme

The overall scheme is based on experience with an implementation of data reduction for the TPC in the ALICE High Level Trigger in Run 1 and 2.

Standard data compression algorithms applied to raw data can only provide compression factors up to $\sim 2$; higher factors can be reached by using knowledge about the data model.

Steps in the processing flow:

1. Reconstruction of Clusters from raw data
2. Transformation to reduced precision with negligible impact to physics
3. Entropy coding - lossless data compression

Two challenges:

- **Efficiency**: Meet requirements for data reduction factor
- **Performance**: Processing time has to fit into available resources
Prototype of a generic Data Compression Framework

While 1st-level reconstruction of raw data with required precision is detector dependent, the lossless entropy coding and storage in optimized format is suited for a generic framework.
Requirements and Boundary Conditions

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

⇒ need a polymorphic solution in the framework, i.e. decide which piece of code to execute based on the type of something

- Innermost functions of the processing loop are called \( O(10^9) \) per TF

⇒ even a minor performance optimization allows for big effects
A Word on Polymorphism

Runtime polymorphism: the actual binding of the type of an object is deferred until runtime, usually realized using classes with virtual functions.

Static polymorphism: completely resolved at compile time
- type checks at compile time
- select among code branches which would not compile in all cases
- generic algorithms
- generic handling of multiple types
- compiler has a lot more information for code optimization

Tools for implementation of static polymorphism
Template programming and meta programming allow to move a significant part of computation and code dispatch from runtime to compile time.
Policy-based design - Decomposing Processing into small entities

- Input policy
- Alphabet
- Probability Model
- Codec
- Output policy

Policies are small functional entities (classes) which take care of separate behavioral or structural aspects. Complex entities are assembled from several small policies.

<table>
<thead>
<tr>
<th>Alphabet</th>
<th>fixed at compile time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability Model</td>
<td>runtime dependent</td>
</tr>
<tr>
<td>Codec</td>
<td>algorithm fixed at compile time</td>
</tr>
</tbody>
</table>

Challenge: need a runtime object which holds the state (pure types do not have a state)

Runtime object has to follow the type definition at compile time
Defining Alphabets

**Individual Alphabets:**
- An alphabet is a set of symbols to be treated by a data compression algorithm
- The alphabet is **fixed** at **compile time**
- A contiguous alphabet of integral numbers of type $T$ between a minimum and maximum value can be defined like
  ```cpp
template<typename T, T _min, T _max, typename NameT> class ContiguousAlphabet {...};
```
- Specializations for distinct cases: alphabet from 0 to some maximum, alphabet for an n-bit field

**Multiple Parameters:**
- Runtime objects have multiple parameters with individual probability models, the parameters need to be stored in a continuous data stream.
- Sets of parameter types defined at compile time, the framework makes use of the *boost Metaprogramming Library.*

Note: these are types, not runtime objects; all information is available at compile time
Probability Model

The probability model describes the statistical occurrence of the symbols of an alphabet

Usually not fixed at compile time; statistics is gathered from a runtime data sample

A type-safe runtime-container

Need a combination of compile time type definitions and runtime objects, an interface between compile time and runtime domains

A solution: Mixin-Based Programming technique

- A recursive definition of identical templates, each wrapping one data type from the list
- The container comprises a recursive definition of types where each type includes the previous ones
- The compiler can walk through the container levels by static cast
- No virtual inheritance

Matthias.Richter@scieq.net (UiO, CERN)
Runtime Dispatch to Container Levels
Access to each data type level through static cast at compile time.

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

The operation is either implemented as specific method in the runtime container or passed to the runtime container level through a functor.

Two options:

1. Dynamic dispatch: compiler creates recursive list of compiled functions from a meta function template
2. Static dispatch: loop unrolling in a specific dispatcher function

What the compiler can do:

- Without optimization: the explicitly unrolled version is faster than the generic recursive approach.
- With optimization: both versions show effectively equal performance, one to two orders of magnitude faster (note x10 scaling in the figure to make them visible)
Optimization of Runtime Dispatch

- Runtime container wraps objects of different data types
- Individual levels of the container can be accessed by type casts
- \texttt{static\_cast} is evaluated already at compile time
  \Rightarrow \text{Generic, 100\% type-safe access to multiple data types}

\textbf{Access patterns:}

- **Recursive access**
  - Generic method;
  - Recursive loop of meta functions, level as parameter

- **Unrolled access**
  - Generic method;
  - Direct cast to required level in a runtime switch; no recursive function calls

- **Bulk access**
  - Specific method for the runtime object to be processed; direct cast to individual levels; no additional runtime switch
Data Compression Framework Prototype in Operation

Testing the framework in the three modes *recursive, unrolled, and bulk operation* with different compiler optimization levels: \(-O0, -O1, -O2, -O3\).

Testing Huffman coding as example operation

\[ \Rightarrow \text{Compiler optimization leads automatically to unrolled code} \]
\[ \Rightarrow \text{Bulk operation is the most performant option} \]
\[ \Rightarrow \text{Static polymorphism: faster operation than runtime polymorphism} \]
Comparison with existing Implementation

Time per operation for compiler optimization level 2

- Current implementation of Huffman compression in AliRoot for ALICE Run 1 and 2 uses runtime polymorphism, base class interfaces and virtual inheritance.

- ALICE O² framework uses static polymorphism for both generic recursive method and specializations.

A concluding estimation

To cope with the data rate @ O(10⁹) operations in 10ms timeframes:

- 30s per TF ⇒ minimum 3000 cores
- 3.5s per TF ⇒ minimum 350 cores
Summary

- ALICE O2 data reconstruction and storage will extensively use data compression
- A generic framework prototype has been developed to facilitate different applications
- Meta programming allows for flexibility AND compile time optimization
- A type-safe interface between compile time and runtime domain has been developed, backbone of polymorphism in the framework
- Encouraging results, method is neither restricted to data compression framework nor bound to particular detector
- Meta programming has a huge potential for online processing at large data rates

Thank you!
Backup slides
Alphabet examples

- Alphabet of contiguous range of symbols between \([\text{min}, \text{max}]\)
  
  \[
  \text{template<typename T, T \_min, T \_max, typename NameT> class ContiguousAlphabet {...};}
  \]

- Alphabet of contiguous range of symbols between \([0, \text{max}]\)
  
  \[
  \text{template<typename T, T \_max, typename NameT> class ZeroBoundContiguousAlphabet {...};}
  \]

- Alphabet for an n-bit field, contiguous range \([0, 2^n]\)
  
  \[
  \text{template<typename T, std::size_t n, typename NameT> class ZeroBoundContiguousAlphabet {...};}
  \]

Examples (omitting \textit{name} template parameter):

\[
\begin{align*}
\text{typedef} & \ \ \text{ContiguousAlphabet<int, -16384, 16383 > MyContiguousAlphabetType}; \\
\text{typedef} & \ \ \text{ZeroBoundContiguousAlphabet<int16_t, 1000 > MyZeroBoundAlphabetType}; \\
\text{typedef} & \ \ \text{BitRangeContiguousAlphabet<int8_t, 6 > My6BitAlphabetType};
\end{align*}
\]
Multiple parameters

The runtime objects have multiple parameters with individual probability models, the parameters need to be stored in a continuous sequence.

To define sets of parameter types at compile time, the framework makes use of the *boost Metaprogramming Library*.

```cpp
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','d','i','f' > >,
  ContiguousAlphabet<int16_t, -32768, 32767 , boost::mpl::string < 't','i','m','e','d','i','f' >>,
  BitRangeContiguousAlphabet<int16_t, 8 , boost::mpl::string < 's','i','g','m','a','Y','2' > >,
  BitRangeContiguousAlphabet<int16_t, 8 , boost::mpl::string < 's','i','g','m','a','Z','2' > >,
  BitRangeContiguousAlphabet<int16_t, 16 , boost::mpl::string < 'c','h','a','r','g','e' > >,
  ZeroBoundContiguousAlphabet<int16_t, 1000 , boost::mpl::string < 'q','m','a','x' > >
> tpccluster_parameter;
```

- Can mix different types of alphabets.
- Again, this is a data type without a state.