Data Compression Framework for ALICE O^2

Matthias Richter

Offline Week Nov 04 2016

Preface

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

We have a solution for TPC data currently in production in the HLT.

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

Goal for the new Compression Framework Prototype

Two challenges:

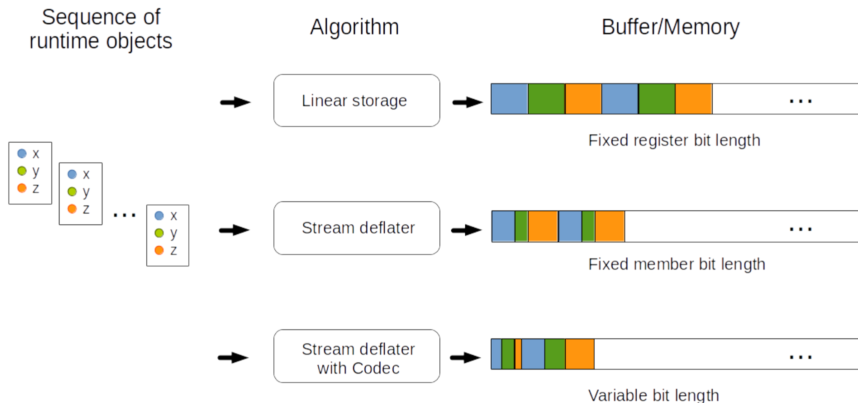
Efficiency	Performance
Meet requirements for data reduction factor	Processing time has to fit into available resources

topic of this talk

- Significant part of the HLT cluster resources is today used for data compression, the implementation has never been optimized
  $\rightarrow$ check what can be optimized, in particular what can be done already at compile time
- Current implementation builds heavily on virtual inheritance and overloaded functions for sake of flexibility
  $\rightarrow$ requires runtime dispatch
- Overcome some structural limitations, e.g. more flexible alphabets, better integration of arithmetic coding

Data Flow

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

$\Rightarrow$ 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

$\Rightarrow$ 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.

Software modules

Policy-based design - Decomposing Processing into small entities

- Input/Output policy
- Alphabet
- Probability Model
- Parameter model
- Codec (Huffman, Arithmetic)

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

Alphabet	fixed at compile time
Probability Model	runtime dependent
Codec	algorithm fixed at compile time

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

```
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

Alphabet examples

- Alphabet of contiguous range of symbols between [min, max]

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

- Alphabet of contiguous range of symbols between [0, max]

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

- Alphabet for an n-bit field, contiguous range [0, 2^n]

```
template<typename T, std::size_t n, typename NameT> class ZeroBoundContiguousAlphabet {...};
```

Examples (omitting *name* template parameter):

```
typedef ContiguousAlphabet<int, -16384, 16383 > MyContiguousAlphabetType;

typedef ZeroBoundContiguousAlphabet<int16_t, 1000 > MyZeroBoundAlphabetType;

typedef BitRangeContiguousAlphabet<int8_t, 6 > My6BitAlphabetType;
```

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*.

```
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','f' >  >,
  ContiguousAlphabet<int16_t, -32768, 32767 , boost::mpl::string < 't','i','m','e','d','i','f','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.

Probability Model

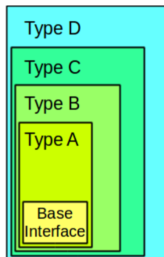
The probability model describes the statistical occurrence of the symbols of an alphabet

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

https://github.com/matthiasrichter/gNeric

Runtime Dispatch to Container Levels

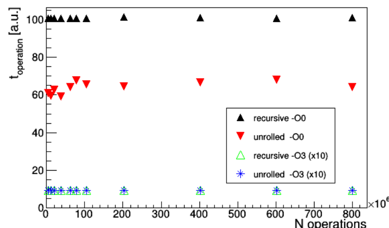
Access to each data type level through static cast at compile time.

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

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

Two options:

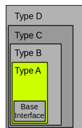
1. Dynamic dispatch: compiler creates recursive list of compiled functions from a meta function template
2. Static dispatch: loop unrolling in a specific dispatcher function



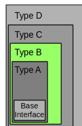
What the compiler can do:

- 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



`static_cast<TypeA>(object)`



`static_cast<TypeB>(object)`

...

- Runtime container wraps objects of different data types
- Individual levels of the container can be accessed by type casts
- `static_cast` is evaluated already at compile time

$\Rightarrow$ Generic, 100% type-safe access to multiple data types

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

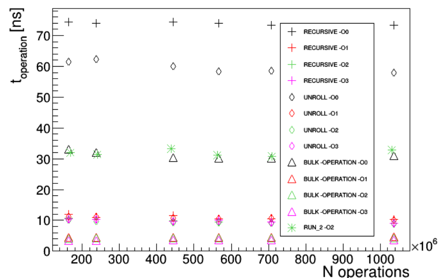
Bulk access

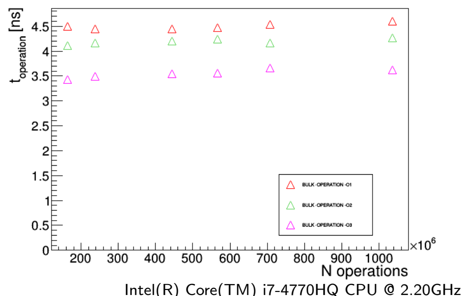
Specific method for the runtime object to be processed; direct cast to individual levels; no additional runtime switch

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





⇒ Compiler optimization leads automatically to unrolled code

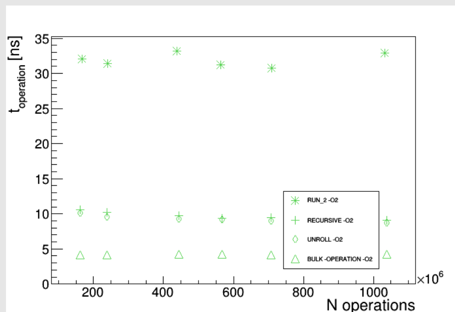
⇒ Bulk operation is the most performant option

⇒ 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 O^2 framework uses static polymorphism for both generic recursive method and specializations.



Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz

Next steps

Ongoing work:

- Individual parts of prototype development tested separately so far, integration underway
- Development fork:
  https://github.com/matthiasrichter/AliceO2/tree/dev-datacompression
  commits are currently being squashed and pull requests are prepared
- Migration of all individual test programs into unit tests

Summary

- 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