Augmenting PODIO Serialisation

Ali Fatemi Abhari
Department of Physics,
University of Tehran, Tehran, Iran

Supervisors: Graeme Stewart, Benedikt Hegner

September 2018

PODIO, or plain-old-data I/O, is a C++ library to support the creation and handling of data models in particle physics. It is based on the idea of employing plain-old-data (POD) data structures wherever possible and avoiding deep-object hierarchies and virtual inheritance. This project tests the validation of serialisation/deserialisation of data from memory to/from disk for different I/O implementations. After the basic functionality is validated, the performance of these options is considered.

1. INTRODUCTION

A POD (plain old data) object has one of these data types: a fundamental type, pointer, union, struct, array, or class. It has no constructor and excludes virtual functions and base classes. The idea of employing plain-old-data data structures and avoiding deep-object hierarchies and virtual inheritance is to both improve runtime performance and simplify the implementation of persistency services.

PODIO provides the necessary high-level functionality to the physicist, such as support for inter-object relations, and automatic memory-management. In addition, it provides a (ROOT assisted) Python interface. To simplify the creation of efficient data models, PODIO employs code generation from a simple yaml-based markup syntax.

To support the use of modern software technologies, PODIO was written with concurrency in mind and gives basic support for vectorization technologies.

This section describes first how to define and create a specific data model, then how to use the created data. Afterwards the overall design and a few of the technical details of the implementation will be explained.

Many of the design choices are inspired by previous experience of the LCIO package used for the studies of the International

Linear Collider, and the Gaudi Object Description applied in the LHCb collaboration at the LHC.

1.1. DESIGN AND IMPLEMENTATION DETAILS

The driving considerations for the PODIO design are:

1. The concrete data are contained within plain-old-data structures (PODs).
2. User-exposed data types are concrete and do not use inheritance.
3. The C++ and Python interface should look as close as possible.
4. The user does not do any explicit memory management.
5. Classes are generated using a higher-level abstraction and code generators.

LAYOUT OF OBJECTS

The data model is based on four different kind of objects and layers, namely:

1. user visible (physics) classes (e.g. `Hit`); these act as transparent references to the underlying data;
2. a transient object knowing about all data for a certain physics object, including inter-object references (e.g. `HitObject`);
3. a plain-old-data (POD) type holding the persistent object information (e.g. `HitData`);
4. a user-visible collection containing the physics objects (e.g. `HitCollection`).

These layers are described below:

- **THE USER LAYER**

The user visible objects (e.g. `Hit`) act as light-weight references to the underlying data, and provide the necessary user interface. For each of the data-members and one-to-one relations declared in the data model definition, corresponding setters and getters are created. For each of the one-to-many relations a vector-like interface is provided.

With the chosen interface, the code written in C++ and Python looks almost identical if taking proper advantage of the `auto` keyword.

- **THE INTERNAL DATA LAYER**

The internal objects give access to the object data, i.e. the POD, and the references to other objects. These objects inherit from `podio::ObjBase`, which takes care of object identification (`podio::ObjectID`), and object-ownership. The `ObjectID` consists of the index of the object and an ID of the collection it belongs to. If the object does not belong to a collection yet, the data object owns the POD containing the real data, otherwise the POD is owned by the respective collection. For details about the inter-object references and their handling within the data objects see below.

- **THE POD LAYER**

The plain-old-data (POD) contains just the data declared in the Members section of the data model definition. Ownership and lifetime of the PODs is managed by the other parts of the infrastructure, namely the data objects and the data collections.

- **THE COLLECTIONS**

The collections created serve three purposes:

1. giving access to or creating the data items;
2. preparing objects for writing into PODs or preparing them after reading;
3. support for the so-called notebook pattern.

HANDLING MUTABILITY

Depending on the policy of the client of PODIO, data collections may be read-only after creation, or may be altered still. While the base assumption of PODIO is that once-created collections are immutable once leaving the scope of its creator, it still allows for explicit unfreezing collections afterwards. This feature has to handled with care, as it heavily impacts thread-safety.

1.2. DATA MODELS AND DATA MODEL DEFINITIONS

To describe a data model PODIO provides a data model definition syntax. This is to ease the creation of optimized data formats, which may take advantage of a so-called struct-of-array data layout. From the user-provided data model description PODIO creates all the files necessary to use this data model.

PODIO encourages the usage of composition, rather than inheritance. One of the reasons for doing so is the focus on efficiency friendly plain-old-data. For this reason, PODIO does not support inheritance within the defined data model. Instead, users can combine multiple `components` to build a to be used `datatype`.

To allow the datatypes to be real PODs, the data stored within the data model are constrained to be POD-compatible data. Those are

1. basic types like int, double, etc.;
2. components built up from basic types or other components;
3. arrays of those types.

A component is just a flat struct containing data. In addition, PODIO supports the storage of one-to-one and one-to-many relations between objects.

Some examples of supported interface can be found in Appendix A.

1.3 PERSISTENCY

The library is build such that it can support multiple storage backends. However, the tested storage system being used is ROOT. The following explains the requirements for a user-provided storage back-end, which is used for writing and testing the ASCII system.

- **WRITING BACK-END**

There is no interface a writing class has to fulfill. It only needs to take advantage of the interfaces provided in PODIO. To persistify a collection, three pieces of information have to be stored:

1. the ID of the collection;
2. the vector of PODs in the collection;
3. the relation information in the collection.

Before writing out a collection, the data need to be put into the proper structure. For this, the method `prepareForWrite` needs to be invoked. In the case of trivial POD members this results in a no-op. In case, the collection contains references to other collections, the pointers are being translated into `collID:objIndex` pairs.

- **READING BACK-END**

There are two possibilities to implement a reading-back end. In case one uses the `podio::EventStore`, one simply has to implement the `IReader` interface.

If not taking advantage of this implementation, the data reader or the eventstore has to implement the `ICollectionProvider` interface.The strong assumption here is that all references are being followed up directly and no later on-demand reading is done.

2. ANALYSIS AND RESULTS

An ASCIIWriter (a sequential file writer) was provided before I started the project. I have enhanced this writer to adapt to new requirements arising from the implementation of a new ASCIIReader. Theiwriter can write every kind of data model with any number of entries. However, the ASCIIWriter can not write the references between objects. All of the program source codes can be found at here.

2.1. ASCIIREADER

The ASCIIReader reads a text file and compares the written results with what it is supposed to be written. However, it is important to note that in this phase, because

the data model itself is not persistified, the reader can read just one type of data model and it can not handle them automatically. Each data model can be added to the reader in a hardcoded way.

2.2. FIRST RUN RESULTS

In the first step a struct containing two integers is chosen as a data model and each collection contains just one struct. Figure 1 and Figure 2 show the performance results for writers and readers respectively. As it can be seen from plots the ASCIIWriter is much more time consuming for larger numbers of events. The ROOTReader is up to 22 times faster. However, for fewer number of events ASCIIWriter is more efficient. The profiling results (GNU profiler) indicate that there is a constant time consumed for initialisation of the ROOT serialiser.

As far as readers are concerned, ASCIIReader, unexpectedly, turns out to be more efficient. Probably the simplicity of data model was the reason of this result, so more complicated situations was needed to be tested.

2.3. SECOND RUN RESULTS

In this level I chose a struct with two float members instead of integers and this time each written collection contains 100 structs instead of one.

Figure 3 and Figure 4 demonstrate the results. Results for writers are nearly same with previous test but this time ROOTReader shows a better performance and nearly 6 times faster speed is achieved.

Profiling again shows that better performance of ASCII reader and writer over ROOT ones for fewer number of events are related to base time consumed by the internal ROOT implementation.

3. SUMMERY & OUTLOOK

- An ASCIIReader is written and performance tests are performed. Results demonstrate the superiority of ROOT storage system for complicated and high number of events.
- The performance of ASCII storage system for simple and fewer number of events is studied partially by profiling, but deeper investigations are needed.
- Binary writers and readers can be provided for wider tests and better comparison.
- A non-ROOT implementation of data model serialisation would allow a wider range of tests to be run more easily.

ACKNOWLEDGMENT

I would like to express my very great appreciation to my supervisors Graeme Stewart and Benedikt Hegner for their great guidance and patience. I would also like to thank CERN for providing this great opportunity for me to participate in this project.

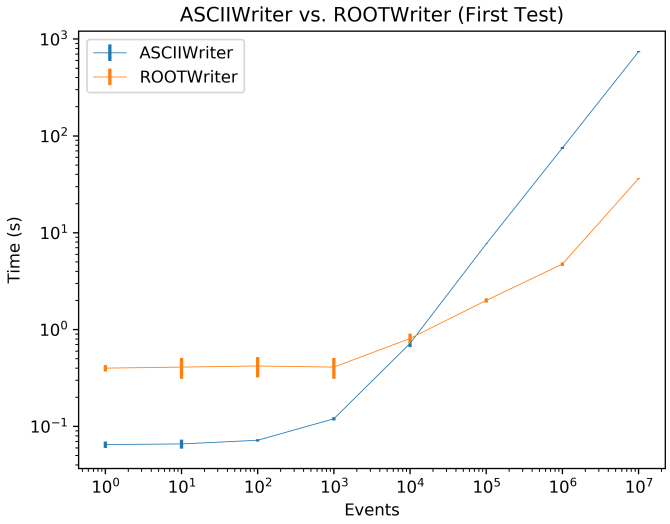

Figure 1: ASCIIWriter vs. ROOTWriter for one struct with two integers

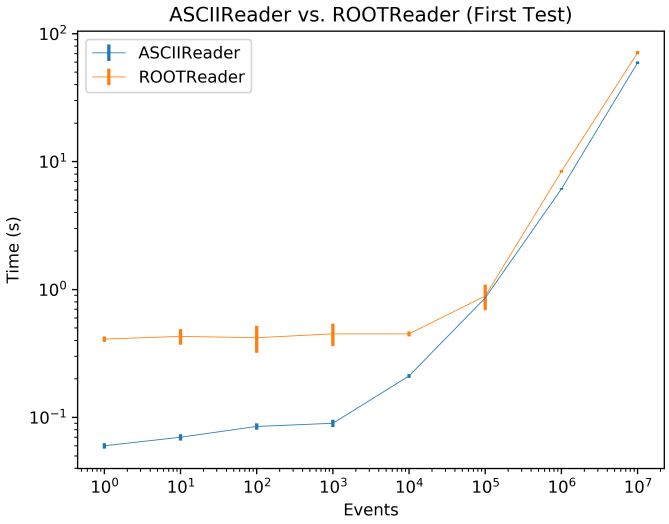

Figure 2: ASCIIReader vs. ROOTReader for one struct with two integers

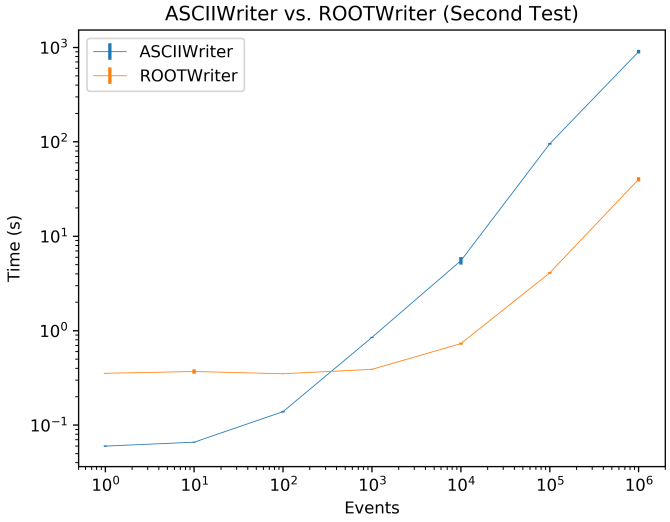

Figure 3: ASCIIWriter vs. ROOTWriter for 100 structs with two floats

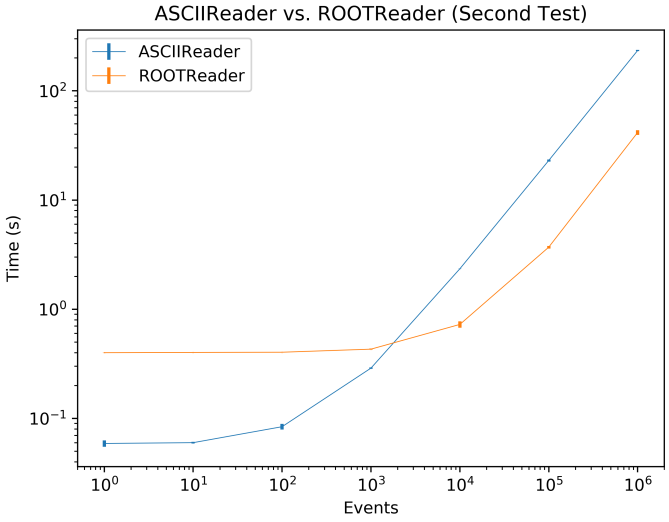

Figure 4: ASCIIReader vs. ROOTReader for 100 structs with two floats

Appendix A

The Syntax

1. DATA MODEL SYNTAX

DEFINITION OF CUSTOM COMPONENTS

A component is just a flat struct containing data. it can be defined via:

```
components:
      # My example component
      MyComponent:
        x : float
        y : float
        z : float
        a : AnotherComponent
```

DEFINITION OF CUSTOM DATA CLASSES

Here an excerpt from "datamodel.yaml" for a simple class, just containing one member of the type int.

```
datatypes :
      EventInfo :
        Description : "My first data
type"
        Author : "It's me"
        Members :
        - int Number // event number
```

Using this definition, three classes will be created: `EventInfo`, `EventInfoData` and `EventInfoCollection`. These have the following signature:

```
class EventInfoData {
      public:
        int Number;
    }
```

```
   class EventInfo {

    public:
      …
        int Number() const;
        void Number(int);
      ...
    }
```

DEFINING MEMBERS

The definition of a member is done in the Members section in the form:

```
Members:
      <type> <name> // <comment>
```

where type can be any build in-type or a component.

DEFINITION OF REFERENCES BETWEEN OBJECTS:

There can be one-to-one-relations and one-to-many relations being stored in a particular class. This happens either in the OneToOneRelations or OneToManyRelations section of the data definition. The definition has again the form:

```
OneToOneRelations:
      <type> <name> // <comment>
OneToManyRelations:
      <type> <name> // <comment>
```

2. EXAMPLES FOR SUPPORTED INTERFACE

Objects and collections can be created via factories, which ensure proper object ownership:

```
  auto& hits =
store.create<HitCollection>("hits")
      auto hit1 =
hits.create(1.4,2.4,3.7,4.2);  // init
with values
      auto hit2 = hits.create(); //
default-construct object
      hit2.energy(42.23);
```

Looping through collections is supported in two ways. Via iterators:

```
  for(auto i = hits.begin(), end =
hits.end(); i != end; ++i) {
        std::cout << i->energy() <<
std::endl;
      }
```

and via direct object access:

```
  for(int i = 0, end = hits.size(),
i != end, ++i){
        std::cout << hit[i].energy()
<< std::endl;
      }
```

3. RUNNING TESTS

An up-to-date installation and quick start guide for each branch of project can be found at the github page under the `doc/branchREADME` of each branch.