EPJ Web of Conferences **245**, 05024 (2020) https://doi.org/10.1051/epjconf/202024505024
CHEP 2019

PODIO: recent developments in the Plain Old Data EDM toolkit

Frank Gaede[1,*], *Benedikt* Hegner[2,**], and *Graeme A.* Stewart[2,***]

[1]DESY, 22607 Hamburg, Germany
[2]CERN, 1211 Geneva 23, Switzerland

Abstract. PODIO is a C++ toolkit for the creation of event data models (EDMs) with a fast and efficient I/O layer. It employs plain-old-data (POD) data structures wherever possible, while avoiding deep object-hierarchies and virtual inheritance. A lightweight layer of handle classes provides the necessary high-level interface for the physicist. PODIO creates all EDM code from simple instructive YAML files, describing the actual EDM entities. Since its original development PODIO has been very actively used for Future Circular Collider (FCC) studies. In its original version, the underlying I/O was entirely based on the automatic streaming code generated with ROOT dictionaries. Recently two additional I/O implementations have been added. One is based on HDF5 and the other uses SIO, a simple binary I/O library provided by LCIO. We briefly introduce the main features of PODIO and then report on recent developments with a focus on performance comparisons between the available I/O implementations. We conclude with presenting recent activities on porting the well-established LCIO EDM to PODIO and the recent EDM4hep project.

1 Introduction

PODIO is a C++ toolkit for the creation of event data models (EDMs) with a fast and efficient I/O layer. It was developed in the AIDA2020 project to address the needs of the Future Circular Collider (FCC) studies with the goal to be applicable also to the linear collider community and HEP in general. PODIO employs plain-old-data (POD) data structures wherever possible, avoiding deep object-hierarchies and virtual inheritance for optimal performance. We will provide an overview of the main features of PODIO and its design in section 2, followed by a description of recent developments in section 3 and a presentation of the I/O performance of different persistency solutions in section 4.

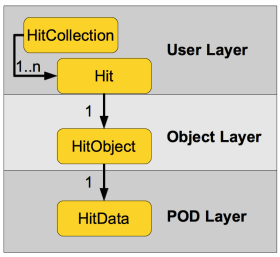
2 Overview

The key idea of PODIO is to use plain-old-data (POD) data structures wherever possible and to avoid deep object-hierarchies and virtual inheritance. A lightweight layer of handle classes provides the necessary high-level interface for the physicist, such as support for inter-object

*e-mail: frank.gaede@desy.de
**e-mail: benedikt.hegner@cern.ch
***e-mail: graeme.andrew.stewart@cern.ch

relationships, convenient iteration through objects or automatic memory-management. An intermediate layer is used for the inter-object relations. The in-memory pointers used for these relations are converted to relative indices when the data is persistified.

The three layers are shown in Figure 1 (left) for an example hit class. PODIO creates all




```
# LCIO MCParticle
MCParticle:
  Description: "LCIO MC Particle"
  Author : "F.Gaede, B. Hegner"
  Members:
    - int pDG               // PDG code of the particle
    - int generatorStatus   // status as defined by the ge
    - int simulatorStatus   // status from the simulation
    #...
  OneToManyRelations:
    - MCParticle parents // The parents of this particle.
    - MCParticle daughters // The daughters this particle.
  ExtraCode:
    const_declaration:
    "bool isCreatedInSimulation() const {
        return simulatorStatus() != 0 ;
     } \n"
```

Figure 1. Left: the three layers of PODIO: the actual *POD Layer* with the data, the *Object Layer* for relations and on top the lightweight *User Layer* with handles and collections. Right: simplified example of a YAML file describing an MCParticle class with relations and additional C++ code.

necessary C++ EDM code from simple instructive YAML files, describing the actual EDM entities, see Figure 1 (right). This is done using Python scripts together with code templates for the various types of objects in PODIO. Object ownership in the generated code is handled via the collections and the event store. Before objects are added to a collection they are reference counted and garbage collected. An additional Python interface is created from the YAML files using PyROOT [1]. A default EventStore implementation is provided with PODIO that provides access to the event collections. HEP frameworks can implement different ways of accessing the event data, more suited to their own internal design.

3 Recent Developments

Since its original development PODIO has been very actively used for Future Circular Collider (FCC) studies. In its original version, the underlying I/O was entirely based on the automatic streaming code generated with ROOT [2] dictionaries. Recently, two additional I/O implementations have been added. One is based on HDF5 [3] and the other uses SIO [4], a simple binary I/O library provided by LCIO [5] (the Linear Collider I/O EDM).

3.1 HDF5 persistency

HDF5 is heavily used in many other science fields as well as by the machine learning community. Providing the option to persistify the EDM in this way allows HEP data to be easily used with tools based around that ecosystem. There was a *Google Summer of Code* project last year with the goal to develop an implementation of an HDF5 I/O layer for PODIO. During this project, prototype code for writing example data structures to HDF5 files was developed, mapping events to *hdf5-groups* and collections to *hdf5-datasets*. It is not entirely clear if this is the optimal way of storing HEP data, which is inherently heterogeneous, in HDF5 that was originally developed for more regular, homogeneous data, such as images. Further work is planned on this topic.

3.2 SIO persistency

SIO (*Simple Input Output*) is the underlying I/O layer of LCIO, used in the linear collider community for more than 15 years. It can store binary records, holding complete events using a simple mechanism to store pointers between objects via a *unique-ID* matching. SIO has recently been re-implemented as a stand-alone package with build-in thread-safety and the possibility to split the reading of events into three steps: the reading of compressed buffers from the file, the decompression of the record and the unpacking of the data in the defined structures and classes. With this feature one can use a *lazy-unpacking* strategy in parallel processing frameworks such as MarlinMT [7], significantly improving I/O performance and scaling. The implementation of the SIO persistency layer for PODIO exploits the array-of-struct data layout with the goal of optimising the I/O performance. Complete events are stored in one record and the unpacking is reduced to the minimum necessary for handling relations and vector members (see section 3.3). A performance comparison of the new SIO layer with the ROOT layer and LCIO is presented in section 4.

3.3 Vector Members

Dynamic length array members, or *vector members*, in EDM data structures break the *POD-ness* and should, ideally, not be used in PODIO. However, they are are used occasionally in many existing HEP event data models. Therefore, the possibility to have them in PODIO was implemented. The implementation for storing vector members follows the treatment of reference vectors:

- hold the vector in the *Object Layer*, store the start and end index in the *POD Layer* and provide iterators in the *User Layer*;
- stream the vector members as one large vector per collection;
- after reading back from the files, vector members are kept in one large vector to minimize copying.

This approach keeps the overhead introduced by vector members to a minimum. A warning message is printed at the time of code generation, encouraging the users to avoid, or at least reduce, the use of vector members.

3.4 pLCIO and EDM4hep

While PODIO was developed for the FCC community, from the start the goal was to be also applicable to the linear collider community as well as HEP in general. Many features in PODIO were introduced to allow porting of the LCIO EDM to PODIO. This has been done now in the pLCIO package which implements the complete LCIO EDM (see Figure 2). The original idea, to be able to create classes that are (almost) 100% backward compatible, was not able to be realised. While it worked for most of the actual member functions of the EDM classes, the handling of collections and collection types, the creation of objects and the handling of user defined parameters would have to be modified in the large existing code base that uses LCIO. However, we believe that a transition from LCIO to pLCIO would be feasible at *reasonable cost*, given the performance advantages expected from PODIO. Recently the particle physics community has decided to develop a common software stack Key4hep [8] for future collider studies with a common event data model EDM4hep [9]. This will be implemented using PODIO and combine the best features of the LCIO EDM and the fcc-edm. The EDM4hep project has just started and is currently under rapid development. As the idea is to preserve most, if not all, the data classes from LCIO, it will very likely replace the pLCIO package.

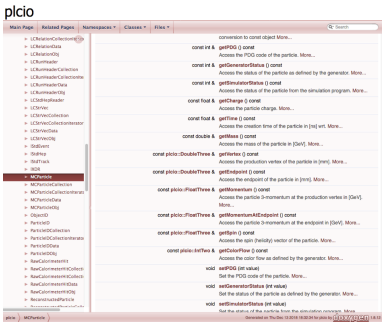


Figure 2. Documentation for the pLCIO classes that implement the complete LCIO event data model. While the interface to access the EDM data classes and their attributes is almost identical to the original LCIO, retrieving collections and handling of meta data is not.

3.5 Other developments

Apart from the recent developments presented in the last sections, a few additional features have been implemented in PODIO:

- code base compatible with C++ 14 and C++ 17
- implemented the support for I/O of *std::array*
- implemented code generation for ASCII streaming with `operator<<(...)` for all EDM classes and collections (see Listing 1)

```
std::ostream& operator<<(std::ostream& o, const ConstMCParticle& v);
std::ostream& operator<<(std::ostream& o, const MCParticleCollection& v);
```

Listing 1. Example of streaming operators for individual objects and complete collections that print the data items in a tabular format.

4 Performance Benchmarks

One of the main motivations to use PODs for PODIO, apart from a simplified EDM, was the expected performance improvement that would be possible when reading data back from a file. The default implementation of the I/O layer is based on auto-generated ROOT dictionaries and uses a columnar-data layout, where every POD member gets stored in an individual branch. This data layout has obvious advantages when reading individual data members of small subset of the total available event data. In cases, like central reconstruction jobs, where always the complete event is read, this might be a disadvantage. The SIO I/O layer, described in section 3.2, stores complete events in one record using the array-of-structs data layout. In order to compare the I/O performance of the different data storage methods, we use a small program that writes and reads Monte Carlo generator events, using the *MCParticle* class from LCIO and pLCIO respectively, with the ROOT and SIO based I/O layers in PODIO and for comparison with the original LCIO library. The files used, contain 17061 generated $t\bar{t}$-events that are written and read completely. The test is carried out on a Mac-book with a solid state disk and on a standard Ubuntu-PC with a classical spinning disk. The result is shown in Figure 3. The writing of the events with ROOT takes 75% of the time of SIO on the Mac (SSD) and 99% on Ubuntu (spinning disk). LCIO is about 20% faster than ROOT, having manually written streamer functions. For reading, ROOT takes more time, 150% and 186% compared to SIO on Mac and Ubuntu respectively. LCIO needs 148% and 132% of the time of SIO for reading. This clearly shows the advantage of the array-of-structs layout for reading full events, as both use the same I/O system but a different data layout. That ROOT is significant slower, is expected due to the columnar layout. Finally, ROOT files sizes are about 76% of

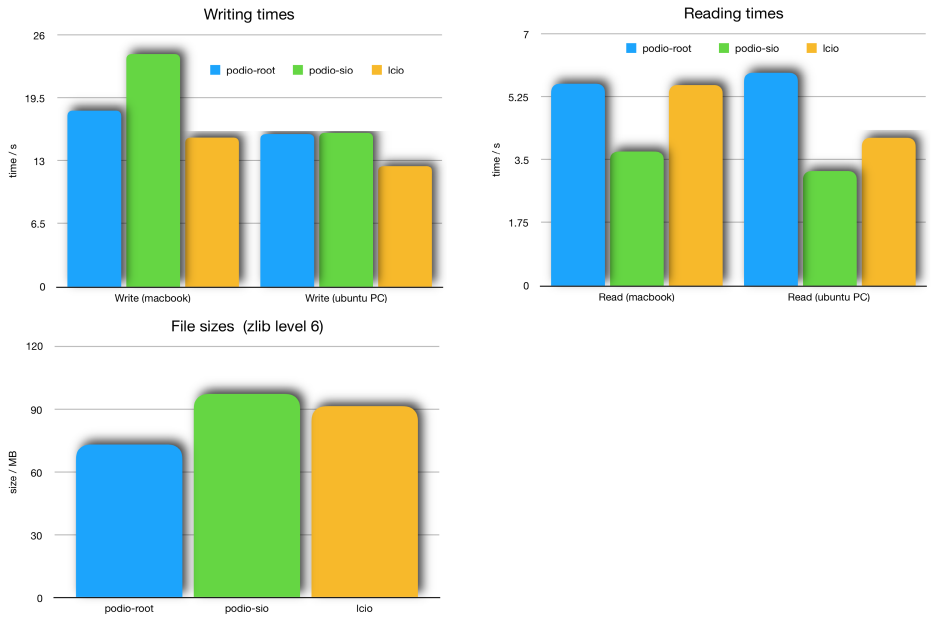


Figure 3. Performance comparison for writing, reading and the file size for the two PODIO I/O layer implementations based on ROOT and SIO and the original LCIO for comparison. The files used, contain 17061 generated $t\bar{t}$-events that are written and read completely.

those created with SIO. Also this is expected, as the compression, chosen to be the same for ROOT and SIO, can achieve better performance with a columnar data layout. The improvements in the reading speed of SIO compared to ROOT justify further studies to investigate if different settings of the splitting level for branches can improve the reading performance or if a dedicated POD storage layout could be implemented.

5 Conclusion

The PODIO event data model toolkit originally developed for FCC and the linear colliders with all of HEP in mind, has been further improved. New features, like the ability to store vector members and alternative I/O layers like HDF5 and SIO, have been implemented. In particular the SIO implementation demonstrates the advantage of using PODs and the array-of-structs data layout. Future plans for PODIO include the generalization of using alternative I/O implementations and the investigation of using, optionally, a struct-of-array layout that might be better suited to the default ROOT persistency. PODIO has been chosen as the solution for the new EDM4hep package that aims at providing the standard EDM for all future collider studies for the coming years. The PODIO developers are open for supporting more HEP projects in the future and volunteer contributions at [11].

6 Acknowledgements

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168. We thank for the constant efforts of the HEP Software Foundation in fostering more collaboration across experiments and collaborations.

References

[1] W. Lavrijsen, "Python in the Cling World," J. Phys. Conf. Ser. **664** (2015) no.6, 062029.
[2] R. Brun and F. Rademakers, "ROOT: An object oriented data analysis framework," Nucl. Instrum. Meth. A **389** (1997) 81.
[3] https://www.hdfgroup.org/solutions/hdf5
[4] https://github.com/iLCSoft/SIO
[5] F. Gaede, T. Behnke, N. Graf and T. Johnson, "LCIO: A Persistency framework for linear collider simulation studies", eConf C **0303241** (2003) TUKT001
[6] F. Gaede, "Marlin and LCCD: Software tools for the ILC", NIM **559**, 177-180 (2006)
[7] R. Ete, J. Benda, F. Gaede, H. Grasland "MarlinMT - parallelising the Marlin framework" These proceedings
[8] A. Sailer, P. Mato Vila, G. Ganis, G.A. Stewart "Towards a Turnkey Software Stack for HEP Experiments" These proceedings.
[9] https://github.com/key4hep/EDM4hep
[10] https://github.com/HEP-FCC/fcc-edm
[11] https://github.com/AIDASoft/podio