EPJ Web of Conferences **214**, 02006 (2019) https://doi.org/10.1051/epjconf/201921402006
CHEP 2018

AREUS

A Software Framework for ATLAS Readout Electronics Upgrade Simulation

Nico Madysa[1,*] on behalf of the ATLAS Liquid Argon Calorimeter Group

[1]Institute of Nuclear and Particle Physics, TU Dresden

Abstract. The design of readout electronics for the LAr calorimeters of the ATLAS detector to be operated at the future High-Luminosity LHC (HL-LHC) requires a detailed simulation of the full readout chain in order to find optimal solutions for the analog and digital processing of the detector signals. Due to the long duration of the LAr calorimeter pulses relative to the LHC bunch crossing time, out-of-time signal pileup needs to be taken into account. For this purpose, the simulation framework AREUS has been developed. It models analog-to-digital conversion, gain selection, and digital signal processing at bit precision, including digitization noise and detailed electronics effects. Trigger and object reconstruction algorithms are taken into account in the optimization process. The software implementation of AREUS, the concepts of its main functional blocks, as well as optimization considerations will be presented. Various approaches to introduce parallelism into AREUS will be compared against each other.

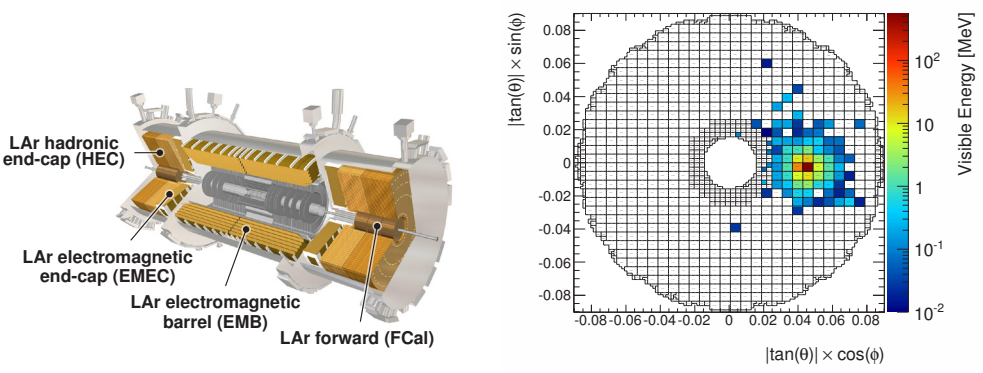
1 Introduction

In order to reach the ambitious goal of collecting data equivalent to $4\,\mathrm{ab}^{-1}$ of integrated luminosity until 2038, the LHC is scheduled to undergo two major upgrades[1]. These will be taken during two long shutdown phases: the *LS2* in 20192020 and the *LS3* in 20242026. The end of LS3 marks the beginning of the *High-Luminosity LHC* phase (HL-LHC). After this point, the peak luminosity is expected to reach a maximum of $75\,\mathrm{nb}^{-1}\,\mathrm{s}^{-1}$, a value about 4 times higher than today. At the same time, the number of protonproton interactions per bunch crossing is expected to increase by a factor of 4 to 5, up to a value of 200.

In order to keep up with the increased luminosity, the ATLAS detector[2] will undergo upgrades at the same time as the LHC: The *Phase-I* upgrade during LS2[3] and the *Phase-II* upgrade during LS3[4]. During the Phase-II upgrade, the full liquid argon (LAr) calorimeter readout will be replaced. This is necessary because the old electronics have only been qualified for a radiation exposure equivalent to $1\,\mathrm{ab}^{-1}$ of integrated luminosity. Furthermore, the new electronics are to supply more fine-grained information to the trigger. Finally, they must accommodate an increase in the trigger rate (from 100 kHz to 1 MHz) and a larger data buffer due to an increased over-all latency on the lowest-level trigger (from 2.5 µs to at least 10 µs).

The greatest challenge for the precise energy measurement of signals in the LAr calorimeter subsystem (shown in Figure 1a) is the increased number of interactions per bunch crossing. These bunch crossings occur once every 25 ns, whereas the electronic pulses caused by

*e-mail: nico.madysa@tu-dresden.de



(a) (b)

Figure 1. **(a)** Cutaway diagram of the ATLAS liquid argon calorimeters[5]. **(b)** Visualization of a Geant4 event, here in the FCal[6, p. 88]. The color of a cell hit indicates the deposited energy.

each calorimeter hit are roughly 450 ns long. A higher number of interactions means an increase of *pileup* effects, where pulses overlap and hinder an accurate energy reconstruction. Techniques that reduce pileup effects often exacerbate electronics noise and vice versa, so an optimum has to be found.

In order to minimize the combined impact of electronics noise and pileup effects, the LAr upgrade requires a detailed simulation of the new readout electronics. This simulation needs to be modular so that new filtering algorithms can be implemented and studied in a timely manner. It also needs to be flexible enough that simulated electronics can be replaced with data from real, prototype components as they become available over time.

This simulation is provided by the AREUS framework, which is described in Section 2. In Section 3, implementation details on AREUS' architecture are given. In Section 4, a custom smart pointer class as an example for the optimization considered in AREUS is presented. In Section 5, various approaches to apply parallelization to AREUS are considered and weighed against each other. Finally, a summary is given in Section 6.

2 The AREUS Framework

AREUS[7–9] is a program written in C++[10]. It directly depends on the Boost[11] and ROOT[12] libraries for common functionality and Git[13] for version control. It can be built using the GNU build system[14–16] or CMake[17]. Other dependencies are bundled into the source repository, either literally or as Git submodules:

- ExprTK[18] for parsing of mathematical expressions,
- Google Test[19] for unit testing,
- Splines[20] for interpolation between data points,
- VDT[21] for vectorized implementations of certain mathematical functions.

In addition, it provides a Python[22] and an ATHENA[23] package to automate certain tasks such as input file generation.

AREUS' input are files derived from detailed Geant4[24, 25] simulations of the full ATLAS detector. These files provide a series of *events*, where each event contains a list of

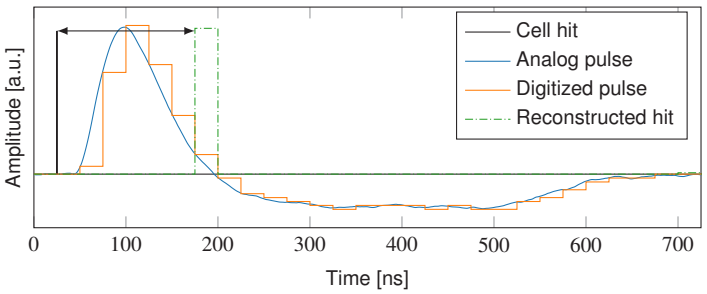


Figure 2. Simulation of the readout chain in AREUS. See the text for a description.

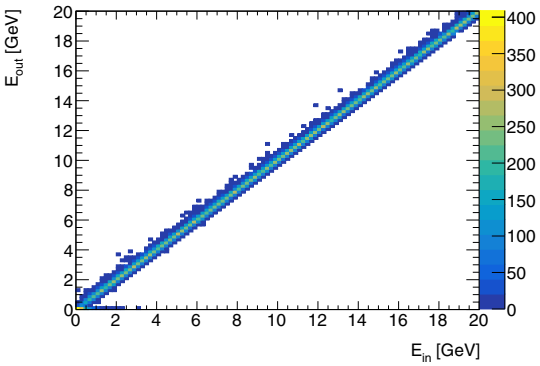


Figure 3. Exemplary output of AREUS, a correlation plot between E_{in}, the energy of the cell hits, and E_{out}, the corresponding reconstructed energy. This takes the filter-induced delay into account. The color represents the number of entries in each bin. The bins right at the X-axis from 0 to 3 GeV show undetected events.

detector cells that have been hit with a certain amount of energy (cf. Figure 1b). AREUS simulates each of the calorimeter cells separately and forwards each hit to the corresponding cell.

Figure 2 illustrates how AREUS processes each cell hit. It calculates the pulse induced by the hit, accounting for effects such as nonlinearity of the amplification stage or electronics noise that follows an arbitrary spectral density. This pulse is then overlaid with any pulses from previous hits in order to simulate pileup effects[1]. After this, the pulse is digitized both in time and amplitude, considering quantization noise and the ADC's finite range and resolution. The samples are subsequently sent to the digital filters, which can be chained by the user in an arbitrary manner. The result of this filter chain should be close to the original series of hits, except for a small, filter-induced delay. AREUS compares these two sequences on-line and produces a collection of diverse histograms detailing the performance of the filter chain. Two examples of these histograms are shown in Figure 3 and Figure 4. Figure 5 shows an analysis that aggregates the results of multiple AREUS runs and has been used in the technical design report for the LAr Phase-II upgrade[4, p. 32].

3 Implementation of AREUS

The modularity of AREUS is achieved through the Observer pattern[26, pp. 293–303] as well as a strict separation of concerns[27, p. 61]. The core library, called *Common*, contains only utility classes used by all other libraries as well as message classes used to interface the other

[1] To do so, cells need to retain their state between events for a duration at least as long as an electronics pulse: 450 ns, or equivalently 18 bunch crossings.

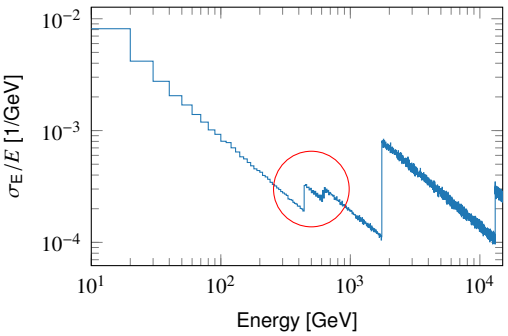


Figure 4. Exemplary output of AREUS. It shows the energy reconstruction's relative resolution over the energy. This graph was part of a qualitative study of dynamic gain selection, where pulses from different gain levels have slightly different shapes. The circled section shows a transition effect caused by sample mixing of pulses from different gain levels. This effect does not occur at 2 TeV because the gain switching there employs a different strategy.

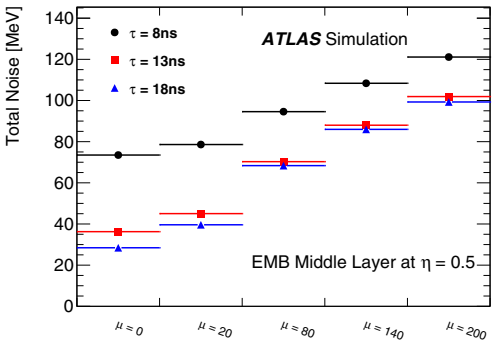


Figure 5. Exemplary analysis of a single calorimeter cell, done using AREUS[4, p. 32]. This shows the total noise due to both electronics noise and pileup effects. Each of the points represents an individual AREUS run. Two parameters have been varied: μ, the average number of proton–proton interactions as a measure of pileup effects; and τ, the RC time constant of the bandpass filter that is employed in the LAr analog electronics.

libraries (cf. Figure 6). The other libraries communicate with each other using the message classes from *Common* (cf. Figure 7). The advantage of this is twofold:

1. the libraries do not need to know about each other, only about *Common*;

2. the user can combine objects from these libraries to arbitrary chains (and even graphs) as long as the message types sent between any two connected objects match.

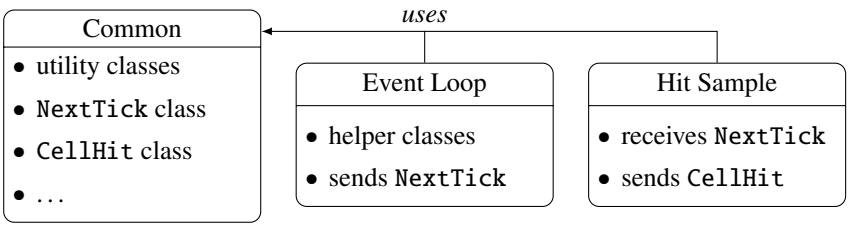


Figure 6. Visualization of AREUS' modular architecture. The core library *Common* contains utility classes and the message classes that other libraries use for communication. The other libraries only know about the message classes, but not about each other.

The second point is most important for the digital-filter chain, which may consist of many small and independent algorithms using the same message class. Figure 8 shows the exact procedure through which AREUS connects objects from different libraries.

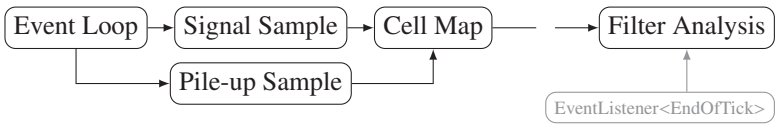


Figure 7. The Observer pattern as implemented in AREUS. Execution starts in a root object that loops over events. On each event, it notifies its observers, which recursively notify their observers. Pre- and post-processing of events is done via inheritance from the special class *EventListener*, which is backed by a global registry object.

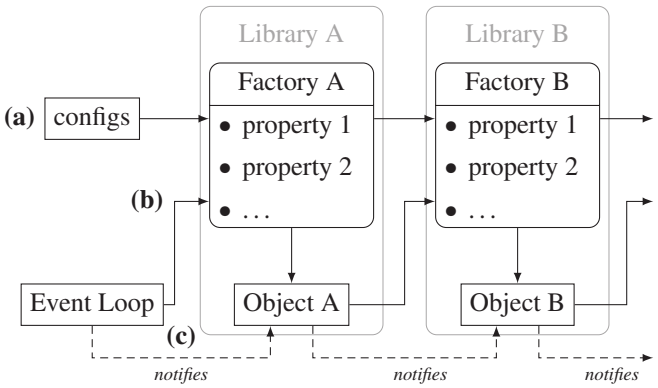


Figure 8. The three phases in which AREUS runs. **(a)** The configuration file is parsed and each module's factory reads its properties from it. A subject–observer chain between factories is built. **(b)** The event loop tells the factories to recursively produce objects. A subject–observer chain between the produced objects is built. **(c)** For each event, the event loop recursively notifies all observers.

4 AREUS Smart Pointers

Due to the inter-library communication in AREUS, many of its runtime objects do not have a clear hierarchy of ownership and reference counting has to be used throughout the code base. To this end, AREUS uses a custom smart-pointer system based on two pointer classes: `CSharedPtr` and `CPersistentPtr`. Though unconventional, this design decision has several advantages over using smart pointers from Boost or the standard library:

- AREUS' smart pointers are *intrusive*, i.e. the reference count is a part of the object that is being reference-counted. This choice avoids allocation of many small memory blocks and so decreases the chance of heap fragmentation.
- The reference counts are regular integers if AREUS is compiled for single-threaded usage (cf. Section 5 for details) and atomic integers otherwise. In contrast, the standard-library class `std::shared_ptr` always uses atomic integers[2]. Accessing atomic integers usually is slightly slower, though on some modern architectures, this additional cost may avoided via *hardware lock elision*[28].
- AREUS' smart pointers provide a runtime mechanism to guarantee data consistency. Whenever a `CPersistentPtr` starts pointing at an object, a flag inside the object is set that marks the object as immutable. Any further modification of such an object results in an assertion failure. As a result, objects are immutable as soon as code *assumes* they are immutable. Violations of this assumption are caught early and with a clear error message.

5 Parallelization Opportunities

By default, AREUS runs in single-threaded mode, taking a set of input files and producing a set of output files. On modern machines with multiple CPUs, this is often wasteful and makes run time longer than necessary. Single-CPU clock speeds have stopped increasing more than a decade ago and most gains in computing nowadays result from parallelization and vectorization[29, p. 9].

There are several methods to make use of more than one core per CPU with AREUS, each with its advantages and disadvantages:

Multiprocessing: This is the simplest method of parallelization and requires no changes to the program at all. It simply means to split the user's data into batches and run one instance of AREUS on each of them in parallel. When comparing different setups, one may also run multiple instances of AREUS on the same data, each using a different configuration file. One of the disadvantages of this method is that the user has to do additional data pre- and post-processing. Another one is that computing time is wasted if the run time of parallel AREUS configuration varies a lot – most cores will be idle while the user waits for the longest run to finish.

Vectorization: Another method is data-level parallelism, i.e. using a CPU's SIMD instructions (single instruction on multiple data), which are available on virtually all modern processors. This technique may speed up array-based algorithms by a factor between two and four. It can be used on a small scale and replace unvectorized algorithms without modifying the overarching architecture of the program. In certain situations, optimizing compilers can even apply it automatically (*auto-vectorization*). AREUS makes use of it where possible e.g. by using VDT[21] and by writing algorithms in a way that facilitates autovectorization. However, this method is inherently limited in scope and cannot, for example, parallelize independent portions of the simulation.

Task-level multithreading: AREUS' *Thread* library, which can be enabled with a compile-time flag, allows the user to parallelize the calorimeter cell simulation. This is possible because each of these simulations is independent of the others. However, no appreciable performance gain could be observed when introducing this library – possibly because the programs was not profiled sufficiently and the actual bottlenecks lie elsewhere. But even if there was a gain, the considerable number of users that simulate only a single cell per run would not profit from this optimization.

Stage-level multithreading: Another approach, that has yet to be investigated for AREUS, would go a step further and also parallelize the different stages of AREUS' processing chain. AREUS' architecture already is well-suited to such an approach due to its extensive use of shared ownership and message passing.

6 Conclusions

The High-Luminosity LHC is expected to start operation in 2026 and the electronics of ATLAS' liquid-argon calorimeters will be upgraded for this purpose in two phases. AREUS is a valuable tool that has been applied successfully in technical design studies for both the Phase-I upgrade[3, pp. 97–103] and the Phase-II upgrade[4, pp. 31–36]. It faithfully simulates pileup effects and allows developers to quickly implement and investigate digital-filter

[2]Technically, `std::shared_ptr` only uses atomic reference counters when compiled with the `-pthread` flag. However, this flag is mandatory when linking with ROOT, even though AREUS itself is fully single-threaded.

algorithms through its flexible and modular design. Its biggest challenge with respect to computing is handling parallelization in the face of strong data dependencies and a deep stack of function callbacks.

The most promising approach to improve simulation speed is data-level parallelism through algorithms that facilitate automatic vectorization by the compiler. Another, more radical approach is parallelizing the distinct simulation stages through multithreading.

This work was supported in part by the German Bundesministerium für Bildung und Forschung (BMBF) within the FIS research grant 05H15ODCA9.

References

[1] G. Apollinari, I. Béjar Alonso, O. Brüning, P. Fessia, M. Lamont, L. Rossi, L. Tavian, *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1*, CERN Yellow Reports: Monographs (CERN, Geneva, 2017), https://cds.cern.ch/record/2284929

[2] ATLAS Collaboration, JINST **3**, S08003 (2008)

[3] ATLAS Collaboration, Tech. Rep. CERN-LHCC-2013-017, ATLAS-TDR-022, CERN, Geneva (2013), https://cds.cern.ch/record/1602230

[4] ATLAS Collaboration, Tech. Rep. CERN-LHCC-2017-018, ATLAS-TDR-027, CERN, Geneva (2017), https://cds.cern.ch/record/2285582

[5] J. Pequenao, *Computer generated image of the atlas liquid argon* (2008), https://cds.cern.ch/record/1095928

[6] ATLAS Collaboration, Tech. Rep. CERN-LHCC-2015-020, LHCC-G-166, CERN, Geneva (2015), https://cds.cern.ch/record/2055248

[7] A. Ambler, J.P. Grohs, M. Hils, P. Horn, A. Jansen, T. Kwan, N. Madysa, S. Stärz, *Areus: Atlas readout electronics upgrade simulation*, version 2.6 (2013–2018), software, https://gitlab.cern.ch/AREUS/AREUS

[8] S. Stärz, *Energy Reconstruction and high-speed Data Transmission with FPGAs for the Upgrade of the ATLAS Liquid Argon Calorimeter at LHC* (TU Dresden, 2015), thesis presented on 19 May 2015, https://cds.cern.ch/record/2030122

[9] J.P. Grohs, *Simulation of the upgraded Phase-1 Trigger Readout Electronics of the Liquid-Argon Calorimeter of the ATLAS Detector at the LHC* (TU Dresden, 2015), thesis presented on 29 Feb 2016, https://cds.cern.ch/record/2135931

[10] ISO/IEC 14882:2014, *Programming language C++* (2014), https://isocpp.org/

[11] D. Abrahams, B. Dawes et al., *The Boost C++ libraries*, version 1.62 (1999–2018), software, https://www.boost.org/

[12] F. Rademakers et al., *Root*, version 6.12/02 (2018), software, https://doi.org/10.5281/zenodo.1292566

[13] J. Hamano, L. Torvalds et al., *Git – fast, scalable, distributed revision control system* (2005–2018), software, https://git-scm.com/

[14] D.J. MacKenzie, A. Demaille, *autoconf*, version 2.69 (2012), software, https://www.gnu.org/software/autoconf/

[15] T. Tromey, A. Duret-Lutz, *automake*, version 1.13.4 (2013), software, https://www.gnu.org/software/automake/

[16] J. Henstridge, M. van Beers, H. Pennington et al., *pkg-config*, version 0.27.1 (2012), software, https://www.freedesktop.org/wiki/Software/pkg-config/

[17] Kitware, Inc. et al., *CMake*, version 3.1 (2000–2018), software, https://cmake.org/

[18] A. Partow, *The C++ mathematical expression toolkit library* (2012–2018), software, http://www.partow.net/programming/exprtk/index.html

[19] The Google Test Project, *Google Test*, version 1.8 (2016), software, https://github.com/google/googletest

[20] E. Bertolazzi, *The Splines C++ library*, version 1.0 (2013), software, https://github.com/ebertolazzi/Splines

[21] D. Piparo, V. Innocente, T. Hauth, Journal of Physics: Conference Series **513**, 052027 (2014)

[22] Python Software Foundation, *Python language reference*, version 2.7 (1990–2018), software, https://www.python.org/

[23] P. Calafiura, W. Lavrijsen, C. Leggett, M. Marino, D. Quarrie, *The Athena Control Framework in Production, New Developments and Lessons Learned*, in *Computing in High Energy Physics and Nuclear Physics* (2005), p. 456, https://cds.cern.ch/record/865624

[24] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand et al., Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **506**, 250 (2003)

[25] J. Allison, K. Amako, J. Apostolakis, P. Arce, M. Asai, T. Aso, E. Bagli, A. Bagulya, S. Banerjee, G. Barrand et al., Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **835**, 186 (2016)

[26] J. Vlissides, R. Helm, R. Johnson, E. Gamma, *Design patterns: Elements of reusable object-oriented software* (Addison-Wesley, 1994), ISBN 0-201-63361-2

[27] E.W. Dijkstra, in *Selected writings on Computing: A Personal Perspective* (Springer-Verlag, New York, 1972), pp. 60–66, https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

[28] R.M. Yoo, C.J. Hughes, K. Lai, R. Rajwar, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (ACM, New York, 2013), pp. 19:1–19:11, https://doi.acm.org/10.1145/2503210.2503232

[29] A.A. Alves, Jr et al. (2017), arXiv:1712.06982 [physics.comp-ph]