The upgrade of the ATLAS High Level Trigger and Data Acquisition systems and their integration

Ricardo Abreu on behalf of the ATLAS TDAQ Collaboration [1]

***Abstract*—The Data Acquisition (DAQ) and High Level Trigger (HLT) systems that served the ATLAS experiment during LHC's first run are being upgraded in the first long LHC shutdown period, from 2013 to 2015. This contribution describes the elements that are vital for the new interaction between the two systems. The central architectural enhancement is the fusion of the once separate Level 2, Event Building (EB), and Event Filter steps. Through the factorization of previously disperse functionality and better exploitation of caching mechanisms, the inherent simplification carries with it an increase in performance. Flexibility to different running conditions is improved by an automatic balance of formerly separate tasks. Incremental EB is the principle of the new Data Collection, whereby the HLT farm avoids duplicate requests to the detector Read-Out System (ROS) by preserving and reusing previously obtained data. Moreover, requests are packed and fetched together to avoid redundant trips to the ROS. Anticipated EB is activated when a large enough portion of the event is requested, reinforcing this effect. A new HLT Processing Unit exploits current architecture trends with a multiprocessing approach that is based on process forking, thereby bypassing thread-safety concerns, while containing total memory usage through the Operating System's Copy-On-Write feature. HLT and DAQ releases are decoupled by a flexible interface that allows quick updates of the communication between both sides, thus providing increased operational maneuvering. Finally, additional data are recorded through Data Scouting. A method of previewing properties of events whose frequency would otherwise exclude them, this new feature will provide key intelligence for subsequent trigger adjustments.**

I. Introduction

ATLAS [2] is a particle physics experiment that relies on a general purpose detector for studying high-energy particle collisions at the Large Hadron Collider (LHC) [3]. After a successful first physics run, the world's most powerful particle accelerator is currently shutdown for planned maintenance and renovation efforts. During this period, both the LHC machine and the experiments are preparing for an increase in luminosity from $8 \times 10^{33}\mathrm{cm}^{-2}\mathrm{s}^{-1}$ to the nominal $10^{34}\mathrm{cm}^{-2}\mathrm{s}^{-1}$, and possibly beyond. Center-of-mass energy will also increase from a previous maximum of 8 TeV to the design value of about 13 TeV, while the previous bunch spacing of 50ns will be reduced to 25ns. These changes will mean a substantial increase in the rate of the data produced by the ATLAS detector in 2015, after the Long Shutdown 1 (LS1) [4] – as this period between 2013 and 2015 is known.

Only a small fraction of proton-proton interaction events can be recorded for *offline* analysis. The rest must be discarded by an *online* Trigger and Data Aquisition (TDAQ) system [5], to record event data with *offline*-manageable rates. This was already the case in Run 1, but the higher collision energy and luminosity will increase the L1 trigger rates by about a factor of five and thus require tighter filtering requirements. In Run 2, an upgraded ATLAS experiment will deal with these more demanding conditions not so much by scaling computing resources as through the improvement of the systems that are involved.

This document first presents the central elements that are affected by the HLT/DAQ upgrade, before describing in more detail five improvements to the two subsystems that fundamentally affect the way they integrate.

II. The ATLAS Trigger and Data Acquisition

As the name indicates, the TDAQ system is composed of two subsystems: the Trigger and the Data Acquisition (DAQ). The first of these – the ATLAS Trigger – is organized in levels. The first level of data selection is called simply Level 1 (L1) and uses custom hardware to process detector data with coarse granularity. After LS1, ATLAS will see on the order of 10^9 proton-proton interactions per second, but the first trigger level will only trigger event accept decisions with a rate of 100kHz. The data that passes the L1 trigger are then processed in more detail by the software algorithms of the High-Level Trigger (HLT), which allows 1kHz of events to be recorded. With the LS1 upgrade, the HLT system is considerably simplified, which is most apparent in the merger of the previous two software trigger levels – Level 2 (L2) and Event Filter (EF) – into one single HLT level.

Of equal importance to the Trigger, the Data Acquisition (DAQ) subsystem includes all the infrastructure that drives the data flow from the ATLAS detector to *offline* storage. This includes, among other things:

- the detector readout system – which in turn is composed by special readout drivers, links and buffers;
- HLT hosting infrastructure – farms of computers which harbors HLT processing;
- data collection components – to collect data from the ROS;
- data recording components – which buffer accepted events and record them for *offline* study;
- control and management infrastructure – to control and coordinate all TDAQ components;
- monitoring infrastructure – to monitor the operation of the system.

III. HLT/DAQ Integration

The DAQ and Trigger subsystems are complementary and meant to operate in harmony. Even though they can be

Ricardo Abreu is with CERN, PH/ADT Department, CH-1211 Geneva, Switzerland (e-mail: ricardo.abreu@cern.ch)

operated independently for development and testing purposes, their ultimate goal cannot be achieved without one another. In what concerns the DAQ and the HLT in particular, the DAQ provides the HLT Processing Units (HLTPUs) on which to run the trigger software, along with data collection, control, and monitoring services. On the other hand, the HLT employs its own software frameworks to provide *online* event analysis and selection, being responsible for the rate reduction that was mentioned before.

The interdependence of the two systems is materialized through a software interface that specifies what one side can expect from the other: the `hltinterface`. The DAQ and HLT play a role of both client and provider of different aspects of this interface. The `hltinterface` establishes, on one hand, how the DAQ can control the state of the HLT Event selection [6] (e.g. configured, connected, running), through a concrete Finite State Machine (FSM). This FSM establishes the single chain of states that the HLT is guaranteed to comply with. On the other hand, the HLT interacts with the outside world through the `hltinterface` to collect event data, relying on external components that orchestrate and distribute the filtering process. Furthermore, the `hltinterface` provides a means of histogram publishing and specifies the structures that can be used in the communication between both systems (e.g. to convey selection results).

The upgrade of the TDAQ system has far-reaching dimensions, but some of the changes affect the expectations that HLT and DAQ have from each other – that is, they modify the contract between them. These contract changes are flatly reflected on the `hltinterface`, which therefore provides a practical distinction between simple implementation updates and the fundamental behavior reforms this document focuses on.

As shown in Fig.1, the integration of the DAQ and the HLT is achieved mainly by the elements that are on the border between the two systems. These are, in the *online* world:

- the HLT Online Integration Framework - partly composed by software layers that wrap the event-selecting algorithms and the general (*offline*) libraries [7] they rely on, it also includes key components that are especially tailored for the *online* environment. These provide diverse infrastructure functionality, such as histogram publishing, error reporting, and configuration procedures.
- the HLTPU - an overloaded term that can designate both the physical computers and the primary *software* that hosts the HLT, interacting with the rest of the DAQ. The HLTPUs are organized in large processing farms which constitute a paramount segment of the DAQ system, providing its data filtering power.

As previously alluded to, these components can be developed and tested independently, relying on simplified replacements for the other side. A dummy algorithm that burns CPU time and follows a configured trigger distribution is enough to replace the HLT, as far as the DAQ system is concerned. Simulating a DAQ system that can interact with the HLT Online Integration Framework in a consistent way is more difficult. When run *online*, the HLT is hosted by HLTPU processes that depend on a multitude of remaining

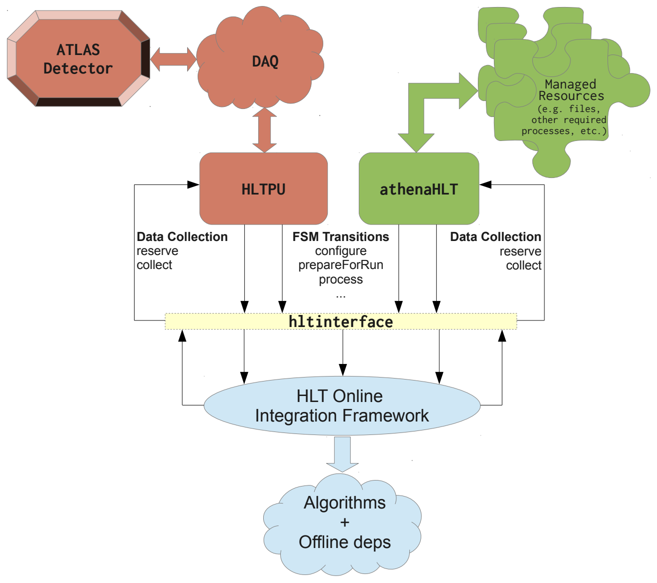
ATLAS Detector
DAQ
Managed Resources (e.g. files, other required processes, etc.)
HLTPU
athenaHLT
Data Collection reserve collect
FSM Transitions configure prepareForRun process ...
Data Collection reserve collect
hltinterface
HLT Online Integration Framework
Algorithms + Offline deps

Fig. 1. The integration between HLT and DAQ elements. The two sides interact through client/provider contracts that are established in the `hltinterface`. Because it respects the same specification, athenaHLT can replace the online components transparently.

programs and resources. The tool `athenaHLT` fills the gap they leave in the *offline* environment, providing the means to run the HLT by replacing the DAQ entirely. Fulfilling the requirements for the use-cases of convenient HLT software testing, event stream reprocessing, and event reconstruction, `athenaHLT` is an emulator of the *online* data flow that is crucial for several activities. The conjunction of `athenaHLT` with the HLT Online Integration Framework forms the full HLT Integration Framework.

IV. Five DAQ/HLT improvements

This section describes five ways in which the ATLAS HLT/DAQ relation is being upgraded for Run 2.

A. HLT merger

Arguably the most important update, the merger of L2 and EF trigger levels into a single HLT level was probably the change with the most impact on TDAQ. Its most direct effects were on DAQ/HLT integration, concretely: the development of a new HLTPU host software; the adaptation of the HLT Online Integration Framework; and the creation of the new tool `athenaHLT`, which corresponds to a reissue of the two previous tools `athenaMT` [8] – for L2 – and `athenaPT` [9] – for EF. Notice that these three software pieces correspond to the direct clients and providers of the `hltinterface`. That is not a coincidence: to merge the two levels we had to radically change what was expected of both sides. The fusion of the previous two specialized interfaces (for L2 and EF) into a uniform `hltinterface` can be seen as the precise reflex of our purpose of merging the two levels: the new `hltinterface` effectively translates it into code. At the interface level, it is a straightforward modification, but it seeds a much vaster unification effort on both sides.

The EF/L2 merger grants a marked simplification of both subsystem's architecture, rendering previous intermediary applications obsolete, thereby releasing computing resources, and dispensing previous code branching, improving the way those resources are used. It is a pervasive improvement with which we aim to reduce intricate component dependencies, communication complexity and networking stress. Given the overlap of L2 and EF functionality, their unification opens up extensive opportunities for factoring common code, discarding specializations, and removing repetitions, ensuing a streamlined, cleaner code base. In addition, the centralization of the high-level event selection procedure is exploited with data caching mechanisms that were not possible before (see IV-B).

We allocate all computing power to a continuous triggering scheme by withdrawing the statical separation between L2 and EF processing. That, in turn, creates a new and important dynamism in the system. No longer do we need to determine beforehand the best node counts for the computing farms of both levels. Nor are we exposed to underutilization of computing nodes anymore, even if changing conditions imply a shift in trigger processing balance. In the new system, every HLT node does all necessary processing for each event it is assigned to treat, going as far as needed, and starting from the beginning with the following event. With no transfer of unfinished processing between L2 and EF, there will be no occasion for work starvation in one level while the other bears too much load. If a rack of computers suddenly becomes inoperative during data-taking, the balance of the remaining resources is unaffected. If a dynamic change in trigger configuration [10] requires more processing at a specific stage, the system adapts. The distribution of the available computing power is not only optimal, it is dynamic and automatic, requiring no manual intervention.

B. Incremental event building

The main difference between the previous separate levels concerned the data collection procedure. In Run 1, L2 algorithms examined only certain regions of the detector to make their decision. Then, an explicit Event Building (EB) [11] step took place, between L2 and EF. Events that were selected in L2 would be assembled in an EB farm, where all the logically-related detector data would be retrieved and packaged in a full event data package: the *full event*. Some of these data would have already been requested in L2, but EB happened in a different farm, so the data had to be requested again. Fully built events were then sent to the EF. Algorithms pertaining to this later step could rely on local availability of the whole event to employ holistic analysis methods, without suffering the additional performance hindering of retrieving data remotely.

In the new HLT, these two types of algorithms are kept, and we maintain the approach of progressively going from coarser and faster selection to more detailed inspection of events. The first algorithms to screen incoming events still focus on small data fractions, leaving comprehensive inspection for the algorithms coming later. Moreover, an Event Building step is still performed, albeit in the same machine where the HLT is running. HLT and EB software runs in each node of the same HLT farm, communicating through shared memory. From an architectural point of view, the three steps are now unified.

The new HLT nevertheless follows L2 in what respects data collection. In other words, rather than receiving the whole event as input, the new `hltinterface` still offers a way to collect individual data portions. Furthermore, all algorithms now have to request data explicitly. But the data are cached and served locally if they were ever retrieved before. More thorough filtering is still left for last and an explicit EB request can be issued by a dedicated HLT service, at an arbitrary point of the algorithm chains, according to the configuration. In each individual data request from the HLT, a bit more of the event data is retrieved and added to the partially held event, until the EB step retrieves the data that was not yet fetched. For all purposes, the event is built in an incremental fashion.

In order to optimize the ROS request rate, the `hltinterface` now offers the possibility of reserving data beforehand. In Run 2, each HLT algorithm will use this approach in a preliminary step, to announce the detector regions it *knows* in advance it will need to examine. Rather than letting each request from the HLT translate into a request to the ROS, the DAQ receives these data-reservation cues through the `hltinterface`, which later allows it to pack requests together, fetching data with a minimal number of trips to the detector readout.

When they receive the data they requested, HLT algorithms are unaware of their immediate origin. They may block for more or less time waiting for them to be delivered, but the distinction between cached and remotely obtained data is semantically irrelevant to them. The data can therefore be held locally before the HLT ever asks for it. This creates an opportunity for an additional DAQ-side optimization, whereby the system will apply configurable rules to pre-fetch data in certain cases. For instance, a simple rule might direct the system to perform event building as soon as a threshold of total requested data was crossed (i.e. after a certain percentage of event data becomes cached). This anticipated EB may be employed to further reduce ROS request rates by trading extra request sizes for lower request multiplicities.

The tool `athenaHLT` also provides a data collection service to the HLT, following the exact same interface. But the data do not come from the detector in this case. Instead, when running *offline*, the HLT usually gets its data from a file. The tool `athenaHLT` fulfills the HLT's data requests with the event data contained in the input files, simulating caching and reservation procedures of the *online* system. The preliminary data reservation and EB requests have no effect on the performance of the *offline* run, but `athenaHLT` still keeps track of them, marking delivered data pieces as cached or not cached in the same way as the *online* system. That information can be used by trigger experts to test trigger menus and find the arrangements that most efficiently exploit the DAQ's caching features without ever actually needing to setup the DAQ.

C. A multiprocessing HLTPU

With recent CPU evolution maintaining relatively stable clock speeds and favoring the increase in the number of hardware threads, modern processors reward parallel over sequential processing. For this reason, sequential programs no longer benefit from any substantial free speed-up when moved to newer machines. Taking advantage of new hardware capabilities implies executing several processes or threads in parallel.

In Run 2, each machine of the HLT farm will still execute several event selection processes concurrently. This was already the case in Run 1, but the processes were completely independent then. That meant that each of them took a full chunk of memory. With the rise in the number of hardware threads largely outranking the increase in available memory, this approach was not scalable. Furthermore, while there are ongoing efforts to upgrade the ATLAS event analysis and selection software to make it multi-threaded, this is not currently the case. The new multi-processing HLTPU allows us to exploit the parallel character of today's machines by running multiple event selection processes at the same time, in a scalable way, thus bypassing thread-safety constraints.

The downside of multi-processing relatively to multi-threading approaches is that they require more memory: one full memory chunk per process. Threads, on the other hand, share most of the memory of the process they are part of. Because of this, the number of processes that can run simultaneously in a single machine often ends up being limited by available memory, rather then by the number of hardware threads. But most of the memory that is used by HLTPU processes contains information that needs to be read but not modified by the HLT and which derives from configuration and present conditions, being therefore common to simultaneous HLT instances. If the HLTPU processes were started independently, this information would be repeated in each one's memory. We avoid that by initially starting only one HLTPU instance in each node. Only after it has completed all configuration and initialization, shortly before the first event is processed, is the HLTPU forked into multiple trigger processes, to take advantage of the Operating System's Copy-On-Write (COW) feature [12]. We therefore guarantee that no copying takes place for memory pages that are read but not modified by forked processes. We effectively reduce the total memory footprint by making the HLTPU processes share large amounts of memory.

Almost all of the information that is used but not changed by the HLT software can be shared by all HLTPU processes, but there are exceptions. An extra step is needed to individualize the memory corresponding to those exceptions. Without it, there would be information left unchanged after the forking that should nevertheless be individual to each process. Unique application names that are used to identify histograms' origin are an example. File descriptors of open log files or network sockets are another. While some of these cases can be dealt with directly by the HLTPU, others concern information that is known only at the scope of the HLT. That is why a new transition was added to the `hltinterface`: `prepareWorker`.

```
procedure HLTPU::PREPAREFORRUN()
  comment: The HLTPU does its initial preparations:
  THIS.INITIALPREPARE()
  comment: It then lets the HLT do its own preparations:
  HLTINTERFACE.PREPAREFORRUN()
  comment: At this point, the forking takes place:
  $forkResult \leftarrow$ FORK()
  comment: Different things happen in mother and children
  if $forkResult = IN\_CHILD$
    then { comment: individualizations on the HLTPU:
           THIS.INDIVIDUALIZE()
           comment: individualizations the HLT handles:
           HLTINTERFACE.PREPAREWORKER()
    else { comment: In mother process
           THIS.ANYADDITIONALSETUP()
```

Fig. 2. The HLTPU's last transition before data taking starts

The corresponding method is called by the HLTPU after it forks, during its own `prepareForRun` transition. This gives the HLT a chance to do the necessary updates in child processes, before they start filtering data. Figure 2 shows the HLTPU's `prepareForRun` procedure conceptually in pseudocode.

When the run ends, the HLTPUs have to come back through a symmetric transition chain. The forked HLTPUs do not go all the way to the initial state though. Instead, they exit early, after the first comeback transition – `endRun`, the converse of `prepareForRun` – is completed. Here, they get the opportunity to cleanup. But the HLT again needs to be able to cleanly release its own resources. Another forking-related transition was therefore introduced into the `hltinterface` for that purpose: `finishWorker`, the converse of `prepareWorker`. The HLTPU child processes only quit after the HLT completes this extra transition. The mother process then returns through the rest of the transition chain alone.

In order to test the consistency of HLT elements in the multi-processing scenario, `athenaHLT` will also be able to fork in a similar fashion, mimicking the transition structure that is used in the *online* world.

D. Flexible communication

Profiting from the original evolution momentum, the manner in which the HLT and DAQ systems integrate was changed in further ways. One important improvement concerns HLT's access to required operational information that originates externally (e.g. configuration parameters). The HLT used to rely on *online* libraries to obtain such information, but that created additional dependencies on the *online* system. These dependencies had the negative side-effect of requiring special code branches in the HLT to handle *offline* runs in `athenaHLT`, and of forcing HLT project recompilations in response to otherwise unrelated software changes.

In Run 2, this problem will no longer be present, as the HLT now directly receives all required information from the DAQ side, through the HLTPU, by means of parameters in the FSM transitions. Furthermore, the type of these parameters was picked to be general enough that their contents are not limited at compile-time. The concrete type that we chose was the *Boost C++ Libraries' Property Tree* [13], abbreviated `ptree`. This type can convey arbitrary information that needs only be derived at run-time. The parameters' structure is that of a tree, which conveniently maps the organization of common configuration files, such as *JSon* or *XML*.

The information that is passed from the DAQ to the HLT does not have to be the same across different runs; nor even across different calls of the same transition method: the parameter's contents are entirely dynamic. But the information that is passed in by the DAQ still has to agree with what the HLT is expecting. The weakly-bound character of FSM parameters' type at compile-time reflects an option of dispensing compile-time checks for the consistency of provided and expected information. It is a trade-off whose benefits are judged to surpass that drawback, as it loosens project release limitations, leaving room for quick updates of the communication between the DAQ and the HLT in response to operational contingencies justifying sudden demands. If the new demands mean that the DAQ needs to pass new information to the HLT but this information already fits the HLT's expectations, it will already be handled by present code and mere configuration changes will usually suffice. Otherwise, the code has to be updated but, since the interfaces remain unchanged, this can be done with software patches, which can be delivered must faster than full software releases.

The new `athenaHLT` is consistent with this approach, deriving the appropriate ptree contents from its own configuration.

E. Data scouting

The new system will be able to collect additional data through a feature called *Data Scouting*. It is the job of the HLT to filter the prodigious amounts of data that are produced by the ATLAS detector. In practice, the decision of accepting or rejecting an event is taken in agreement with predefined thresholds, which are determined with certain output rate reduction targets in mind, leaving potentially relevant data out. Higher input rates only aggravate the issue, as they presuppose stricter filtering.

In Run 2, the HLT will "see" about one thousand more events than can be recorded. Some useful data is therefore discarded and lost forever. To mitigate this problem, the new event selection scheme affords a special place for events that do not quite fulfill the requirements for being fully accepted but that are still likely to be significant. In other words, the new system allows a new category of event that is neither completely rejected nor fully saved – the *data scouting event*. The detector data of these events are dropped and only the result of the HLT's decision process is recorded, including relevant calculations and footprints of the HLT's decision process. Since they are much smaller, the system can make room for a steady rate of data scouting events without deviating from data output rate targets.

To accept this new kind of event, the TDAQ system has to be able to record only part of what would otherwise be full events. Fortunately, this *partial Event Building* capability has long been in place for calibration events. The new system conserves and reuses the same partial EB logic for Data Scouting.

Most of the modifications that are required for Data Scouting are contained within the HLT, but some also reach the level of DAQ/HLT integration. In particular, the HLT now needs to be able to return multiple results to the DAQ, which in turn needs to handle them properly. The reason for this extension is that the HLT produces different results for data scouting and regular physics events. But some events can fulfill the requisites for both data scouting and regular physics triggers at the same time. In these cases, the HLT consequently outputs multiple HLT results.

Recorded data scouting events can be used as a preview method, providing key intelligence to aid the optimization of the filtering process in subsequent threshold and algorithm setups. They can also be employed in new physics searches that explore previously uncharted empirical territory.

V. Conclusion

When data taking resumes in LHC's Run 2, higher output rates from the ATLAS detector will place added pressure on the TDAQ system. This document exposed five enhancements of the HLT and DAQ systems that affect their interaction. The merger of the two previous HLT levels into a single trigger step provides a dynamic balance of HLT computing resources as well as new caching possibilities. These are explored by an incremental event building strategy that avoids redundant data requests from the detector readout. A multi-processing HLTPU takes advantage of the multiple hardware threads each node provides and employs COW to reduce memory usage. Flexible communication from the DAQ to the HLT is achieved with general parameters in the FSM transitions, allowing prompt responses and consequently minimizing data losses. Finally, the new Data Scouting feature provides a means of gathering information on events that would otherwise be lost. These five updates, taken together, are expected to contribute considerably towards maximizing the efficiency of the TDAQ system.

References

[1] ATLAS TDAQ Collaboration, *The ATLAS Trigger/DAQ Authorlist*, version 13.05, ATL-DAQ-PUB-2013-001, CERN, Geneva, 2011, http://cds.cern.ch/record/1553190.

[2] ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST, Vol. 3, p. S08003, 2008.

[3] L. Evans and Ph. Bryant, *LHC Machine*, JINST, vol. 3, p. S08001, 2008.

[4] F. Bordry *et al.*, *The First Long Shutdown (LS1) for the LHC*, International Particle Accelerator Conference (IPAC), 4th, 2013, p. 1355.

[5] ATLAS HLT/DAQ/DCS Group, *ATLAS High-Level Trigger, Data-Acquisition and Controls Technical Design Report*, CERN/LHCC/2003-022, ATLAS TDR 016, 2003, http://atlas-proj-hltdaqdcs-tdr.web.cern.ch/atlas-proj-hltdaqdcs-tdr.

[6] PESA Software Group (ed. M. Elsing), *Analysis and Conceptual Design of the HLT Selection Software*, ATL-DAQ-2002-013, 2002, http://cdsweb.cern.ch/record/685387.

[7] S. Armstrong *et al.*, *Studies for a Common Selection Software Environment in ATLAS: From Level-2 Trigger to the Offline Reconstruction*, IEEE Trans. Nucl. Sci. 51 (2004) no. 3, p. 915920.

[8] P. Pinto, A. dos Anjos, W. Wiedenmann, *AthenaMT and Level-2 Software Integration*, ATL-DH-EN-0009, 2005, https://edms.cern.ch/document/571749.

[9] M. Bosman *et al.*, *athenaPT (pt test) and Event Filter Software Integration*, ATL-DH-OR-0002, 2005, https://edms.cern.ch/document/581296/1.

[10] F. Winklmeier *et al.*, *Real-time Configuration Changes of the ATLAS High Level Trigger*, Real Time Conference (RT), 2010, 17th, IEEE-NPSS.

[11] W. Vandelli *et al.*, *The ATLAS Event Builder*, IEEE Trans. Nucl. Sci. 55 (2008) no. 6, Part 2, p. 35563562.

[12] A. Silverschatz *et al.*, "Chapter 10: Virtual Memory" in *Operating System Concepts*, John Wiley & Sons, Inc., International Edition, 6th edition, 2003.

[13] (2013, Oct. 28) "Chapter 22. Boost.PropertyTree" in *The Boost C++ Libraries BoostBook Documentation Subset* [Online], version 1.55.0. Available: http://www.boost.org.