AthenaMT: Upgrading the ATLAS Software Framework for the Many-Core World with Multi-Threading

Charles Leggett[1], John Baines[2], Tomasz Bold[3], Paolo Calafiura[1], Steven Farrell[1], Peter van Gemmeren[4], David Malon[4], Elmar Ritsch[5], Graeme Stewart[6], Scott Snyder[7], Vakhtang Tsulaia[1], Benjamin Wynne[8] on behalf of the ATLAS Collaboration

[1]Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, USA
[2]STFC Rutherford Appleton Laboratory, Harwell Oxford, Didcot OX110QX, United Kingdom
[3]AGH, University of Science and Technology al. Mickiewicza 30, PL-30-059 Cracow, Poland
[4]Argonne National Laboratory, 9700 S. Cass Ave, Argonne, IL 60439, USA
[5]European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland
[6]SUPA - School of Physics and Astronomy, University of Glasgow, Glasgow G12 8QQ, United Kingdom
[7]Brookhaven National Laboratory, P.O. Box 5000, Upton, NY 11973, USA
[8]SUPA - School of Physics and Astronomy, University of Edinburgh, Edinburgh EH9 3JZ, United Kingdom

E-mail: cgleggett@lbl.gov

ATL-SOFT-PROC-2017-019
24 January 2017

Abstract. ATLAS's current software framework, Gaudi/Athena, has been very successful for the experiment in LHC Runs 1 and 2. However, its single threaded design has been recognized for some time to be increasingly problematic as CPUs have increased core counts and decreased available memory per core. Even the multi-process version of Athena, AthenaMP, will not scale to the range of architectures we expect to use beyond Run2.

After concluding a rigorous requirements phase, where many design components were examined in detail, ATLAS has begun the migration to a new data-flow driven, multi-threaded framework, which enables the simultaneous processing of singleton, thread unsafe legacy Algorithms, cloned Algorithms that execute concurrently in their own threads with different Event contexts, and fully re-entrant, thread safe Algorithms.

In this paper we report on the process of modifying the framework to safely process multiple concurrent events in different threads, which entails significant changes in the underlying handling of features such as event and time dependent data, asynchronous callbacks, metadata, integration with the Online High Level Trigger for partial processing in certain regions of interest, concurrent I/O, as well as ensuring thread safety of core services. We also report on upgrading the framework to handle Algorithms that are fully re-entrant.

1. Introduction

ATLAS's[1] framework (Gaudi[2]/Athena[3]) was designed to process serially one event at a time. Limitations of existing and emerging computing technology, as well as the requirements of the ATLAS reconstruction environment, have forced us to examine concurrent, multi-threaded implementations[5].

In 2014, the ATLAS *Future Framework Requirements Task Force* created a report that summarized a list of requirements and recommendations for such a framework, and in 2015 ATLAS began the process of migrating its software to the new design[6]. For a large experiment like ATLAS, with a massive software code base, a complete rewrite of the software was deemed untenable, and instead, capitalizing on certain fundamental features of the GaudiHive[7] framework, which allow the localization of enforced thread safe code to only a limited set of components, an *evolutionary* migration strategy was developed.

The most difficult part of this migration is the requirement that all shared software Services, *i.e.* components that can be accessed concurrently by clients from multiple threads, must not only be thread safe, but must also be able to process requests from different events. These core Services must be fixed before any realistic migration of user analysis algorithms can commence. ATLAS has created an aggressive migration schedule in order to be ready in time for Run 3, and in 2016, we have focused our attention on the migration of core Services. We will detail our efforts in this paper.

2. Enabling Concurrency in Core Services

In order to function properly in AthenaMT, shared Services must be thread safe, and also able to process requests from various clients that may be executing in different concurrent events. These requirements have a broad range of impacts on the migration of the software. Some Services are already event agnostic, and can be made thread safe with simple mutexes or thread safe data structures.

Some Services need more intrusive modifications to be able to handle state information that is associated with multiple concurrent events. For example, the MagFieldSvc, which is used to calculate the value of the detector's magnetic field at any given location, was caching localized data internally, in order to speed up sequential requests which are usually for very similar physical locations. Not only was this thread unsafe, but this localization was broken if multiple clients queried the Service simultaneously, requesting information about very different regions. The Service was re-designed to carry a cache object along with each request, localizing the cache to the client, and not the Service, thus maintaining both thread safety and performance benefits.

Another example is the THistSvc, which is used to manage histograms and ntuples. Since the objects it manages are identified either by name, or a pointer to the object, multiple concurrent clients that updated these objects could interfere with each other. The Service was upgraded so that clients could either request a shared object, with an enforced locking policy that prevented simultaneous updates to the object, or their own copy of the object that would not be shared. At the end of the job, these copied objects could be automatically merged.

And some Services, such as those that manage Asynchronous Data, (*e.g.* the IOVSvc which manages detector Conditions or the GeoModelSvc which deals with detector alignments), need a complete redesign. These will be discussed in further detail below.

3. Concurrent Processing of Asynchronous Data

One of the challenges in the migration process has been the handling of Asynchronous Data, *i.e.* data which can have a lifetime of more than one event. The period of time for which any piece of such data is valid is referred to as an Interval of Validity (IOV). While we do have a solution for managing multiple concurrent Event Stores belonging to different events, Asynchronous data cannot be stored there, as the contents of the Event Store are erased at the end of each event, so a different solution must be found.

We can classify Asynchronous Data into two, somewhat interrelated, categories: Conditions, such as high voltages, calibrations, etc., and Detector Geometry and Alignments. Closely related to these are Asynchronous Callbacks (Incidents), which are functions that need to be executed

at non-predetermined intervals, such as in response to the opening of a file, or the signaling of the beginning of a new run.

3.1. Conditions

In serial Athena, Conditions were managed by the Interval of Validity Service (IOVSvc). At the start of a job, the IOVSvc is configured to manage a number of objects in an associated Conditions Database, which stores the value of each object for each IOV. At the start of each event, the IOVSvc examines the validity of each registered object. Objects that are no longer valid are re-read from the database, and any required post-processing of the data is performed by an associated callback function. The processed objects are then placed in a conditions store, from whence they can be retrieved by a user Algorithm.

This workflow fails when multiple events are processed concurrently. Since only a single instance of the conditions data can be held at any one time in the conditions store, if two events are processed concurrently, with associated conditions data from different IOVs, one will overwrite the other. Furthermore, neither the IOVSvc itself nor any of the conditions callback functions were designed to be thread safe, and since these are shared instances, threading problems are bound to occur. A major rewrite of the entire infrastructure is required.

Several different designs for the condition handling were examined, with two key requirements in mind: minimize changes to client code (as there is so much of it), and minimize memory usage (as an overall memory shortage is one of the main reasons we need to use a multi-threaded framework). Designs considered and discarded were the use of a processing barrier, and the use of multiple conditions stores. With a processing barrier the framework would only concurrently process events that had the same set of conditions, draining the event scheduler until all these events had finished processing, then loading a new set of conditions data for the next set of events, before resuming processing events concurrently. The problem with this design is that it assumes that conditions boundaries are infrequent, so that the loss of concurrency when the scheduler is drained and refilled is minimal. On ATLAS, however, Conditions changes can sometimes occur very rapidly, for example as frequently as once per event in the Muon subsystem. This would have the effect of serializing event processing, with complete loss of concurrency. Processing out of order events could also result in a loss of concurrency.

The other rejected design used multiple conditions stores, one per concurrent event, in the same manner as the Event Stores are duplicated for each concurrent event. The mechanism by which data is retrieved from the conditions store would be modified, such that clients would associate with the correct Store. Impact on client code would be small - only the conditions data retrieval syntax would need to be updated. However, beyond merely ensuring thread safety of the IOVSvc and the callback functions, there are two significant problems with this design: the memory usage would balloon, as objects would be duplicated between each Store instance; and also the execution of the callback functions that are used to process data would be duplicated, resulting in extra CPU overhead.

The chosen solution was to implement a single conditions store in the form of a multi-cache. Instead of holding individual Condition Objects, it holds containers of them, where the elements in each container correspond to individual IOVs. Clients access Condition Objects via smart references, called ConditionHandles, which implement logic to determine which element in any ConditionContainer is appropriate for a given event. The callback functions are migrated to fully- fledged Condition Algorithms, which are managed by the framework like any other Algorithm, but only executed on demand when the Conditions Objects they create need to be updated.

One of the fundamental changes in the client code needed for the migration to AthenaMT is that all access to event data must be done via smart references, called DataHandles. DataHandles are declared as member variables of Algorithms, and provide two fundamental

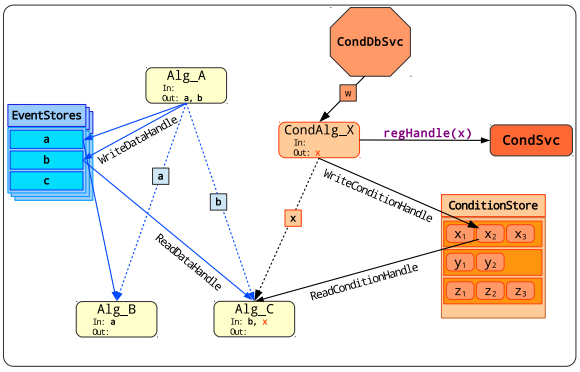

Figure 1. ConditionHandles

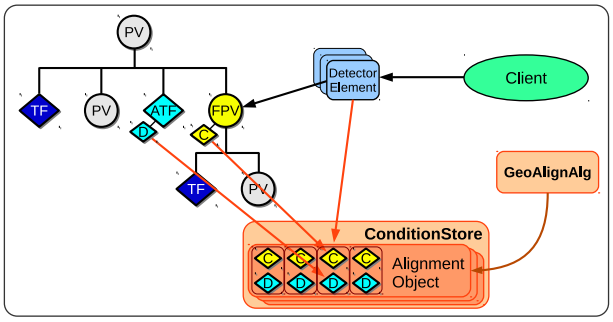

Figure 2. Detector Geometry Alignments

functions: to perform the recording and retrieval of event data, and to automatically declare the data dependencies of the Algorithms to the framework, so that the Algorithms can be executed by the Scheduler as the data becomes available. We capitalized on the migration to DataHandles by requiring that all access to Conditions data be done via related ConditionHandles. By using ConditionHandles in the Condition Algorithms to write data to the conditions store, the framework solves the problem of Algorithm ordering for us, ensuring that the Condition Algorithm is executed, and the updated Condition Objects are written to the store before any downstream Algorithm which needs to use them are executed.

When a ConditionHandle is initialized, it will look in the conditions store for its associated container, identified by a unique key. This container holds a set of objects of the same type and their associated IOVs. Upon dereferencing, the ConditionHandle will use the current event and run numbers to look inside this container, and determine what action needs to be taken. At the start of the event, the Condition Service analyzes the subset of the objects held in the condition store that have been registered with it at the start of the job by the Condition Algorithms, and determines which are valid or invalid for the current event. If an object is found to be invalid, the Condition Algorithm that produces that object will be scheduled. If an object is found to be valid, then the Scheduler will be informed that this object is present, and placed in the registry of existing objects. In this case the Condition Algorithm will not execute.

When a Condition Algorithm is executed, it queries the Conditions Database for data corresponding to the current event, as well as its associated IOV, creates the new object for which it is responsible, and adds a new entry in the ConditionContainer that is associated with a ConditionHandle (see Fig. 1). When a downstream Algorithm that needs to read a ConditionHandle from the store is executed, the data is guaranteed to be present. The ConditionHandle uses the current event number to identify which element in the container is the appropriate one, and returns its value.

By using basic features of the new framework, namely the use of ConditionHandles and data flow dependencies to automatically schedule Algorithms as needed, we are able to minimize changes to client code, and delegate the majority of the work to the framework. The use of collections of Condition Objects inside of a single conditions store allows us to minimize the total memory footprint.

3.2. Detector Geometry and Alignments

The detector geometry model used in ATLAS (GeoModel), is a hierarchical tree that is built from several components (see Fig. 2): a Physical Volume (PV) which are the basic building blocks; a Transform (TF) that is fixed at construction; and an Alignable Transform (ATF), which accounts for the movement of the detector component as a function of time, reading

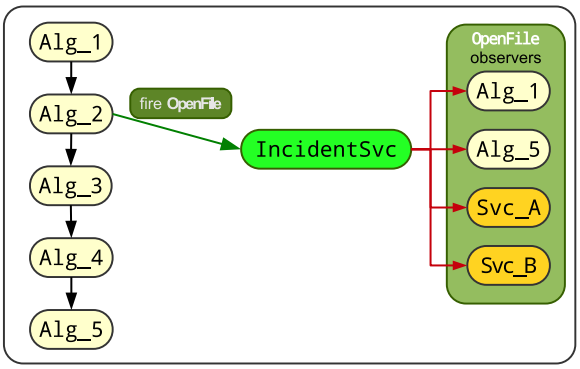

Figure 3. The IncidentSvc in Serial Athena

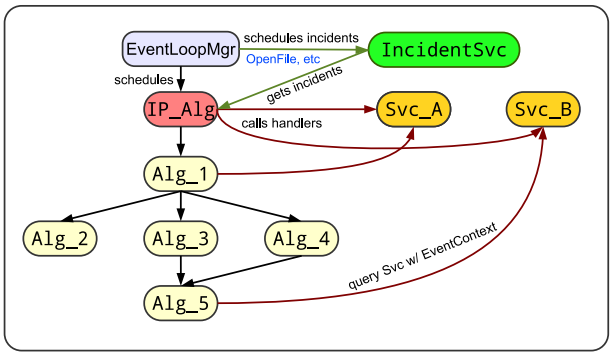

Figure 4. Handling Incidents in AthenaMT

Deltas (D) from a database. When a client requests the position of a Detector Element, the Full Physical Volume (FPV) is assembled, and the position is cached (C). As the detector alignment changes, new Deltas are read in by the ATF, and the cache held by the FPV is invalidated, until the position of the element is again requested, recomputed, and cached.

When multiple concurrent events are processed, this design will fail, as there is only a single shared instance of the GeoModel tree, and the ATF and FPV can only keep track of single Delta or cache at any one time. We can solve this problem in the same way as for the conditions. The time dependent information (*i.e.* the Deltas and cache) held by the GeoModel is decoupled from the static entries, and held in a new AlignmentObject located inside the ConditionStore. The ATF and FPV use ConditionHandles to access this data, and they are updated by a new GeoAlignAlg which is scheduled on demand by the framework. Clients of the DetectorElements are entirely blind to this change, and the only code that needs to be modified are base classes inside the GeoModel structure.

4. Asynchronous Callbacks

ATLAS uses the Incident Service to execute callback functions at certain well-defined times following the well-established observer patterns. Clients register interest in certain "Incidents" with the Service, such as BeginEvent, FileOpen, or EndMetaData. When components fire these Incidents, execution flow is passed to the IncidentSvc, which triggers the appropriate callback function in the registered observers (see Fig. 3). There are many issues with this design in AthenaMT, where there can be multiple instances of any Algorithm, executing simultaneously in different events. If a cloned Algorithm is an Incident observer, should all instances execute the callback? What if an instance is currently executing in a different thread? Fixing the design in a generic way looked to be an impossible task.

Instead, we did a study of exactly how Incidents were being fired and used, and discovered that the vast majority were fired outside the event execution loop (*i.e.* before or after all Algorithms are executed for one event), and being used to signal discrete state changes, such as BeginEvent. We realized that we could significantly limit the scope of the IncidentSvc without losing any functionality. Incidents instead became schedulable (see Fig. 4), where the IncidentSvc would add special IncidentAlgs at the beginning or end of the event processing loop, which would interact with event context aware Services to perform the same function as the old Incident callback functions. Clients would then interact with these Services, passing them the current event to extract the relevant information.

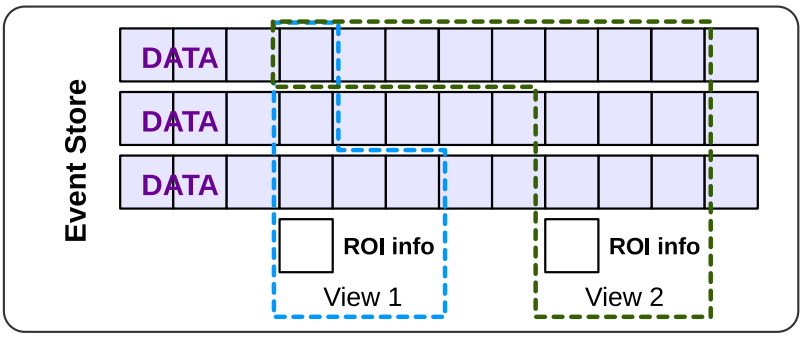

Figure 5. Structure of EventViews and ROIs in the EventStore

5. Event Views

One of the major requirements put forth by the *Future Framework Requirements Task Force*, was that the online and offline software be unified, and use the same code base. The High Level Trigger (HLT), used by the online, functions in a significantly different manner than offline reconstruction and analysis. In order to maximise performance and perform quick rejection, the HLT operates independently on geometrical Regions of Interest (ROI). Multiple filtering and hypothesis Algorithms are applied to each ROI in chains, with the ability to abort a chain if an unsatisfactory conclusion is reached. Each of these Algorithms sees only the data relevant to the ROI that it is operating upon, unlike the offline analysis Algorithms which operate on the entire detector. But since the same Algorithms are to be shared by the online and offline, the framework must be able to present the data to the Algorithms in a ROI or detector agnostic manner.

We were able to make use of the fact that Algorithms access Event data via smart DataHandles, and having the framework modify the data presented to the Algorithm internally within the DataHandle itself. This was done by implementing an EventView class, that can be used interchangeably with the whole Event Store. It has the same interface as the Event Store, but adds a Data Object that describes the corresponding ROI. In the case of online processing, each Event View is populated with the data corresponding to a single ROI, and the DataHandle of a particular Algorithm is updated to point to that View before execution by the framework.

By this mechanism, we are not only able to use the same algorithmic code for online and offline analysis, but it also offers us the possibility of increasing concurrency in the offline by doing sub-Algorithmic parallelism in various ROIs.

6. Re-Entrant Algorithms

One of the features of AthenaMT which frees Algorithm authors from having to implement thread safe code, is that any single instance of an Algorithm is guaranteed to process an event in its entirety in only a single thread at a time. Concurrent event processing is achieved by cloning Algorithms, *i.e.* multiple instances of the same Algorithm are created by the framework, so if the same Algorithm is scheduled simultaneously in different Events, each Event gets its own copy of the Algorithm. This frees Algorithm authors from having to worry about most forms of thread safety. Thread hostile behavior, however, such as the use of global statics, must still be avoided.

The downside of this cloning mechanism is that memory usage will increase as each Algorithm is copied. While the user can set the number of copies of each Algorithm during job configuration, and limit the cloning, this comes at the expense of limiting possible concurrency. There is always a trade off between performance and memory usage.

Because of this, we decided to add a new type of Algorithm - a fully re-entrant Algorithm. For these Algorithms, the framework will only create one instance, and the same instance may

be executed by the scheduler concurrently in multiple events. This eliminates the memory overhead of multiply cloned Algorithms. The re-entrant Algorithms must be made fully thread safe, and also stateless. In order to facilitate this design, the signature of the Algorithm's event process method had been made *const*, and the EventContext, a class which is used to provide information about the currently executing event, is explicitly passed into the method. Furthermore, the name of the base class of the Algorithm has been changed, to ensure that it is not used accidentally.

Having re-entrant Algorithms available gives us greater flexibility in designing Algorithms. While it is unlikely that the majority of existing Algorithms will be migrated to a re-entrant design, it is recommended to users that they first attempt to write new Algorithms in the re-entrant fashion. Older Algorithms will be made re-entrant on a case by case and as-needed basis, in all likelihood by experts who understand the complexity of re-entrant designs.

7. Conclusions

ATLAS has begun the migration of core framework elements to function in AthenaMT. While in some cases this is a relatively straight forward task, it often requires significant design changes beyond mere thread safety. Some redesigns were able to preserve the full functionality of the serial versions, but in other cases it became apparent that the serial version was completely incompatible with a concurrent environment. In these cases we examined the real use of the code, and limited the re-design to reproduce the actual use cases.

We have made design choices that minimized alterations to client code, due to its enormous volume. By leveraging on existing features of the framework, such as DataHandles, data dependency hierarchies of Algorithms, and the structure of the Scheduler itself, we have been able to vastly simplify the task. We anticipate on-schedule finalization of design, and implementation of essential core Services by the end of 2016, with full support of multi-threaded concurrency by the end of 2017. We already have production level ATLAS Geant4 simulations running in multi-threaded environment on the Knights Landing platform[8].

Changes to Algorithmic client code that use these framework elements are also underway. We believe that, for the most part, there is a relatively straight forward recipe for this migration, but it will nevertheless require a significant amount of manpower to effectuate. The broad migration of ATLAS's Algorithm client code to function in AthenaMT will take place in 2017.

References

[1] ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *JINST* **3**, S08003 (2008).

[2] G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank and G. Gracia *et al.*, "GAUDI - A software architecture and framework for building HEP data processing applications," *Comput. Phys. Commun.* **140**, 45 (2001).

[3] P. Calafiura, W. Lavrijsen, C. Leggett, M. Marino and D. Quarrie, "The Athena control framework in production, new developments and lessons learned," *CHEP 2004 Conf. Proc.* **C04-09-27** pp 456-458 (2005)

[4] Binet S et al., 2012 Multicore in production: Advantages and limits of the multiprocess approach in the ATLAS experiment *J. Phys.: Conf. Series* **368** 012018 (ACAT2011 proceedings)

[5] P. Calafiura, W. Lampl, C. Leggett, D. Malon, G. Stewart and B. Wynne, "Development of a Next Generation Concurrent Framework for the ATLAS Experiment," J. Phys. Conf. Ser. **664**, no. 7, 072031 (2015). doi:10.1088/1742-6596/664/7/072031

[6] G. A. Stewart *et al.*, "Multi-threaded software framework development for the ATLAS experiment," J. Phys. Conf. Ser. **762**, no. 1, 012024 (2016). doi:10.1088/1742-6596/762/1/012024

[7] M. Clemencic, B. Hegner, P. Mato and D. Piparo, "Introducing concurrency in the Gaudi data processing framework," *J. Phys. Conf. Ser.* **513**, 022013 (2014).

[8] S. Farrell, P. Calafiura, C. Leggett, V. Tsulaia, A. Dotti *et al.*, "Multi-threaded ATLAS Simulation on Intel Knights Landing Processors," *Proceedings of the CHEP 2016 conference J. Phys.: Conf. Ser.*