

Managing Asynchronous Data in ATLAS's Concurrent Framework

C. Leggett^{1,2}, J. Baines³, T. Bold⁴, P. Calafiura², J. Cranshaw⁵, A. Dotti⁶, S. Farrell², P. van Gemmeren⁵, D. Malon⁵, G. Stewart⁷, S. Snyder⁸, V. Tsulaia², B. Wynne⁸ on behalf of the ATLAS Collaboration

²*Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley CA 94720, USA*

³*STFC Rutherford Appleton Laboratory, Harwell Oxford, Oxfordshire, UK*

⁴*AGH University of Science and Technology, Krakow, Poland*

⁵*SLAC National Accelerator Laboratory, Menlo Park, CA 94025, USA*

⁶*Argonne National Laboratory, 9700 S. Cass Ave, Argonne, IL 60439, USA*

⁷*SUPA – School of Physics and Astronomy, University of Glasgow, Glasgow, UK*

⁸*SUPA – School of Physics and Astronomy, University of Edinburgh, Edinburgh, UK*

E-mail: cleggett@lbl.gov

In order to be able to make effective use of emerging hardware, where the amount of memory available to any CPU is rapidly decreasing as the core count continues to rise, ATLAS has begun a migration to a concurrent, multi-threaded software framework, known as AthenaMT.

Significant progress has been made in implementing AthenaMT - we can currently run realistic Geant4 simulations on massively concurrent machines. The migration of realistic prototypes of reconstruction workflows is more difficult, given the large amount of legacy code and the complexity and challenges of reconstruction software. These types of workflows, however, are the types that will most benefit from the memory reduction features of a multi-threaded framework.

One of the challenges that we will report on in this paper is the re-design and implementation of several key asynchronous technologies whose behaviour is radically different in a concurrent environment than in a serial one, namely the management of Conditions data and the Detector Description, and the handling of asynchronous notifications (such as FileOpen). Since asynchronous data, such as Conditions or detector alignments, has a lifetime different than that of event data, it cannot be kept in the Event Store. However, multiple instances of the data need to be simultaneously accessible, such that concurrent events that are, for example, processing conditions data from different validity intervals can be executed concurrently in an efficient manner with low memory overhead, and without multi-threaded conflicts.

*38th International Conference on High Energy Physics
3-10 August 2016
Chicago, USA*

¹Speaker

1. Introduction

ATLAS's ^[1] framework (Athena^[2]) was designed to serially process 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^[3]. ATLAS has begun the process of migrating its software to the new framework (AthenaMT), focusing this year on making its core Services thread safe and able to process multiple concurrent events.

One of the challenges in this 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 loosely 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.

2. 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).

2.1 Processing Barrier

The first considered design was to use a processing barrier, such that only events where all Conditions objects were unchanged were processed concurrently. No new events would be

scheduled until all events within the same Conditions region had finished processing. Then the conditions store would be updated using the IOVSvc machinery, and new events could be scheduled. By utilizing this technique, very few changes would need to be made to the client code, and there is no extra memory usage, as there is only one instance of the conditions store. The majority of the work would be in making the scheduler aware of the Conditions boundaries, and doing the appropriate filling and draining of associated events. The fundamental problem with this method, however, 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. Another problem is that it assumes that all events are processed in sequence. If events are out of order near a Conditions boundary, then the processing barrier could be triggered multiple times, once again resulting in a significant loss of concurrency.

2.2 Multiple Conditions Stores

Another proposed design was to use 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.

2.3 Multi-Cache Conditions Store

The chosen solution is to implement an intersection of the two preceding designs, with a single conditions store that holds containers of condition data objects, where the elements in each container correspond to individual IOVs. Clients access Condition objects via smart references, called ConditionHandles, which implements the 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 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 Ill. 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.

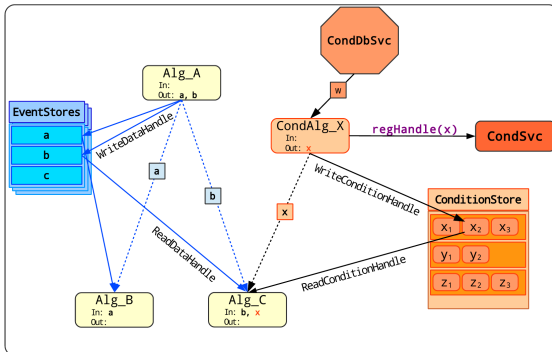


Illustration 1: ConditionHandles

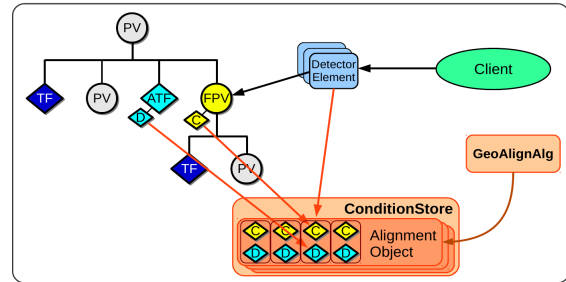


Illustration 2: Detector Geometry Alignments

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 let the framework do the majority of the heavy lifting. The use of collections of Condition Objects inside of a single ConditionStore allows us to minimize the total memory footprint.

3. Detector Geometry and Alignments

The detector geometry model used in ATLAS (GeoModel), is a hierarchical tree that is built from several components (see Ill. 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 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 Incidents

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. 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 (*ie* 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, 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.

5. Conclusions

For ATLAS, managing Asynchronous data in a concurrent environment has required a paradigm shift. No solution is fully transparent or plug and play, unless we choose to sacrifice concurrency and performance, or increase memory usage. Dealing with multiple threads as well as multiple concurrent events increases the complexity of the problem.

In spite of these difficulties, we have been able to minimize impact on client code via strategic modifications at the framework and Service level, leveraging core features of AthenaMT such as DataHandles and the Scheduler itself. In some cases, we have found it necessary to reduce the complexity of Services to fit their actual uses, resulting in simpler designs with no loss of performance or functionality.

New versions of all three aspects of the Asynchronous data and event infrastructure discussed in this paper have been implemented, and migration of client code is ongoing. We anticipate full operation of these services by the end of 2016.

References

- [1] ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, *JINST* **3** (2008) S08003
- [2] P. Calafiura, W. Lavrijsen, C. Leggett, M. Marino, D. Quarrie, *The Athena control framework in production, new developments and lessons learned*, *CHEP 2004 Conf. Proc.* **C04-09-27** (2005) pp 456-458
- [3] P. Calafiura, W. Lampl, C. Leggett, D. Malon, G. Stewart, B. Wynne, *Development of a Next Generation Concurrent Framework for the ATLAS Experiment*, *J. Phys. Conf. Ser.* **664** (2015) no.7, 072031