EPJ Web of Conferences **214**, 05031 (2019) https://doi.org/10.1051/epjconf/201921405031
CHEP 2018

Conditions DataHandling in the Multithreaded ATLAS Framework

Charles Leggett[1,*], *Illya* Shapoval[1,**], *Scott* Snyder[2,***], and *Vakho* Tsulaia[1,****] on behalf of the ATLAS Collaboration

[1]Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, US
[2]Brookhaven National Laboratory, P.O. Box 5000, Upton, NY 11973, US

Abstract. In preparation for Run 3 of the LHC, the ATLAS experiment is migrating its offline software to use a multithreaded framework, which will allow multiple events to be processed simultaneously. This implies that the handling of non-event, time-dependent (conditions) data, such as calibrations and geometry, must also be extended to allow for multiple versions of such data to exist simultaneously. This has now been implemented as part of the new ATLAS framework. The detector geometry is included in this scheme by having sets of time-dependent displacements on top of a static base geometry.

1 Introduction

During Run 1 and Run 2 at the LHC, ATLAS[1] utilized a serial event processing framework called Athena[2][3] and its multiprocess capable variant AthenaMP[4]. However it was determined that neither Athena nor AthenaMP would scale to the projected computational requirements or Run 3, and a new data flow driven, multithreaded implementation which enables concurrent processing of multiple independent events, was required. This framework has been called AthenaMT[5][6].

Asynchronous or time varying data, also commonly referred to as conditions data, are data whose lifetime can be longer than one event. Some data may remain the same for multiple runs, while some may sometimes change as often as every event. Managing this sort of data in a concurrent framework poses many challenges beyond those necessary for a serial event processing environment.

Multiple versions of the data that are referenced by different time points may be in use at the same time when the framework processes concurrent events, and the framework must be able to correctly return the appropriate version to a client in an efficient manner when requested. As a processing job progresses, multiple versions of the conditions data are loaded, necessitating some form of garbage collection to reduce memory consumption. Adding the complexities of multithreading to the equation only increases the difficulty of the task. In this paper we present the solution that ATLAS has implemented to manage its time varying data as implemented in the AthenaMT framework.

*e-mail: cgleggett@lbl.gov
**e-mail: ishapoval@lbl.gov
***e-mail: snyder@bnl.gov
****e-mail: vtsulaia@lbl.gov

2 Conditions Store

In Athena, the serial ATLAS framework, there is one instance of the event store, called StoreGate[7], and one instance of the conditions store, which is implemented as a special version of StoreGate. Algorithms, the main processing unit of the framework, access event data by means of smart references called DataHandles, where individual data objects are referenced via unique keys.

At the end of each event the event store is flushed, to prepare for the next event to be read in. It is unnecessary to do the same for the conditions store, as the data therein changes at points defined by its associated interval of validity (IOV), so the data only needs to be updated, usually being read from a database, when a new IOV is entered. Clients can register callback functions with a particular condition data object, which are triggered when a condition object is read in as it enters a new IOV. This is done to generate "derived" conditions data, which may in fact be dependent on multiple raw conditions objects. This derived data is also written to the conditions store.

This workflow for accessing conditions data 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.

In AthenaMT, the concurrent, multithreaded ATLAS framework, there is one instance of StoreGate created for each concurrent event. When an Algorithm or other client needs access to an object in the event store, it does so via the use of a DataHandle, and an EventContext object, which contains information about the current event, such as an event and run number. The object identifier key encoded in the DataHandle, along with the EventContext object, is sufficient to identify the appropriate object in the correct instance of the event store associated with the current event that the Algorithm is processing.

While the same mechanism could be used for conditions data, *i.e.* creating separate instances of the conditions store for each concurrent event, it would be grossly inefficient, from both a memory and processing point of view, as much of the data would be identical between all stores, and the callback functions that generate derived data would have to be executed multiple times.

After investigating a number of different designs, with two key requirements of minimizing changes to client code, and minimizing memory usage, the implementation that ATLAS chose was a single instance of multi-cache condition store, shared amongst all concurrent events (see Figure 1). Instead of holding individual condition objects, the store holds containers of them, where the elements in each container correspond to individual IOVs. This is implemented as a ConcurrentRangeMap templated in the type of the contained condition object, and indexed by the IOV, which allows for efficient lookup with no locking, and locked writing with concurrent reading.

Clients access condition objects via smart references, with a similar idiom to DataHandles, called CondHandles, which implement logic to determine which element in any condition container is appropriate for a given event. The callback functions from serial Athena which are used to populate the derived conditions objects, are migrated to fully- fledged Condition Algorithms, that are managed by the framework like any other Algorithm, but only executed on demand when the conditions objects they create need to be updated. The Algorithm Scheduler, which executes Algorithms in an order determined by their data dependencies, is aware of the IOV associated with each condition object, and will only trigger the execution of the associated Condition Algorithm when a new IOV is entered.

While some conditions data are derived, and created by Conditions Algorithms which can perform extensive post-processing, some are merely read from a conditions database

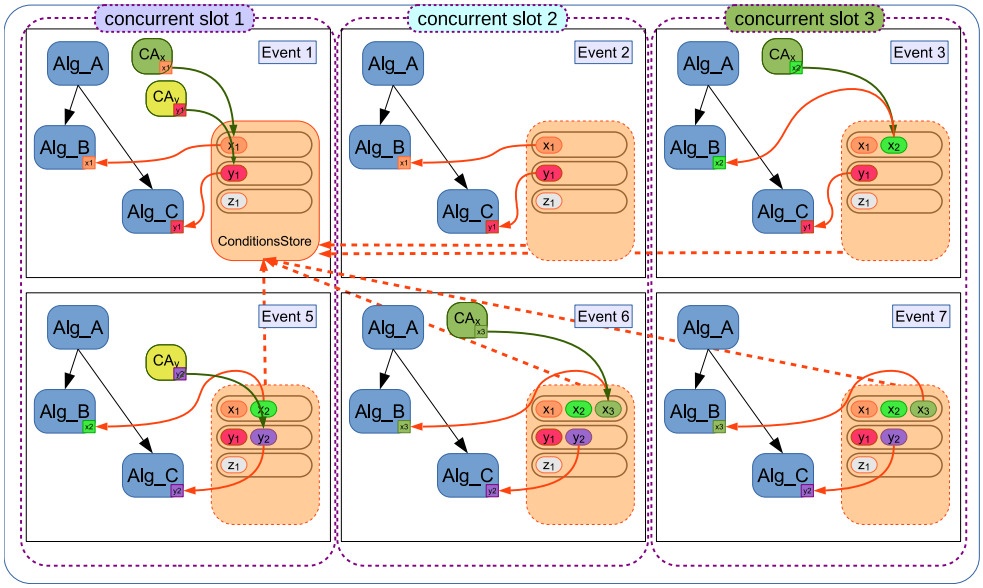

Figure 1. The ATLAS multi-cache Conditions Store

and placed directly into the conditions store. In order to facilitate this process, a special Algorithm called the CondInputLoader is configured with the list of database folders and keys from which the data is to be read. During initialization, this Algorithm uses special factory macros to automatically create the appropriate conditions containers in the conditions store. These containers are then automatically populated via appropriate database accesses on IOV boundaries when the CondInputLoader is executed by the Scheduler.

3 Condition Handles

One of the fundamental requirements for the client code needed for the migration to AthenaMT is that all access to event data must be done via DataHandles. DataHandles are declared as member variables of Algorithms, and provide two functions: to perform the recording (WriteHandles) and retrieval (ReadHandles) 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 only after the data they require has become available. We capitalized on the migration to DataHandles by requiring that all access to conditions data be done via related CondHandles. By using CondHandles 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 (via a declared ReadCondHandle) are executed.

Upon initialization, Condition Algorithms register themselves and the WriteCondHandles that they will create conditions data in the conditions store with a special Conditions Service (see Figure 2). This makes an association between the WriteCondHandle and the Condition Algorithm that creates it, which the Scheduler needs to know in order to trigger the execution of the Algorithm at the appropriate time.

When a CondHandle is initialized during the initialization phase of its parent Algorithm, it will look in the conditions store for its associated container, identified by a unique key, and creating it if necessary. This container holds a set of objects of the same type and their associated IOVs.

At the start of the event, the Scheduler queries the Condition Service to analyze 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 for execution. If an object is found to be valid, then the Scheduler knows it can be ignored. If all conditions objects associated with a Conditions Algorithm are valid for an event, then the Scheduler will not execute it.

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 conditions container that is associated with the WriteCondHandle. By the time a downstream Algorithm that needs to access conditions data via a ReadCondHandle is executed by the Scheduler, the data is guaranteed to be present. The CondHandle uses the information in the current EventContext (such as event and run numbers, lumi-block number or nanosecond time stamp) to identify which element in the container is the appropriate one, and returns its value.

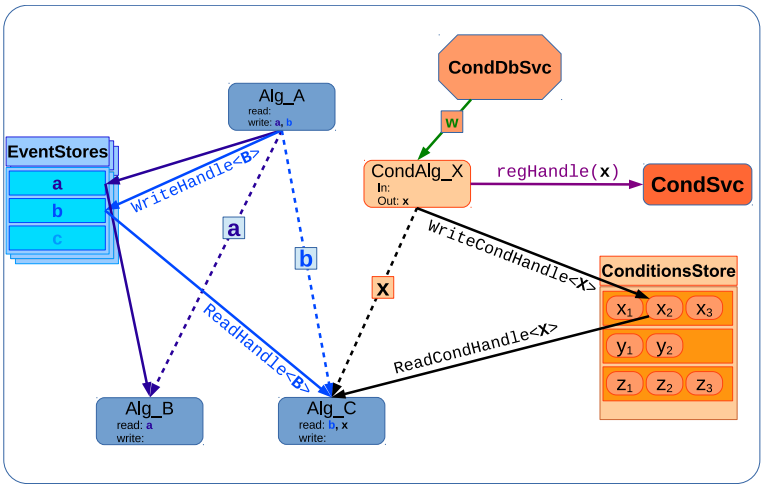

Figure 2. Accessing ConditionHandles in AthenaMT

4 Detector Description and Geometry

The detector geometry model used in ATLAS (GeoModel), is a hierarchical tree that is built from several components (see Figure 3): 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 conditions store. 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.

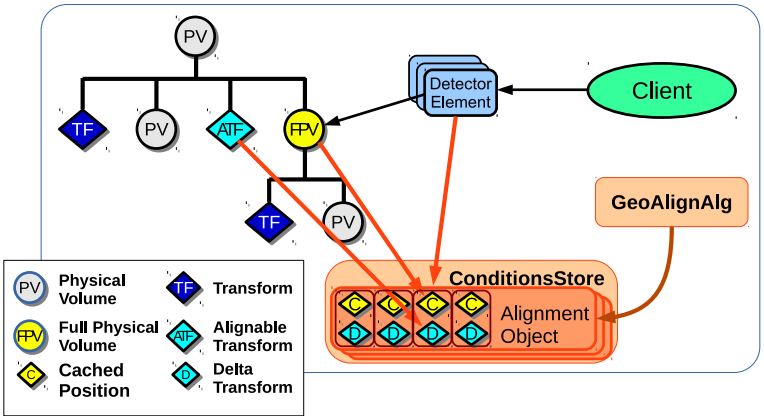

Figure 3. Concurrent implementation of the ATLAS Geometry Model and Detector Description

5 Garbage Collection

Since the concurrent implementation of the conditions store holds containers of conditions data, and new elements are added to the containers as new IOVs are entered, the size of the store will grow as the job progresses. Depending on how long the job runs, the number of conditions data needed, and how many IOV boundaries are crossed, this can result in significantly more memory consumption than was used in the serial implementation. However, not all container elements are actually needed at any one time - only the ones that are referenced by events that are currently being processed. This means that significant memory savings can be had through judicious use of garbage collection.

One complication is that events are not necessarily guaranteed to be processed in the same order that they were taken. This means that if conditions data is aggressively pruned during a reprocessing run as soon as they are no longer in use, a subsequent event which is in fact from a previous instant in time, may require reloading the just deleted data, and triggering a sequence of execution of Condition Algorithms. This is unwelcome as it results in unnecessary database access, as well as extra processing. Instead a certain delay in the removal of old objects can be beneficial.

This is implemented as follows (see Figure 4): whenever a new conditions object is created, the framework notes that conditions container should be examined for old conditions N_{delay} events later. The actual garbage collection is performed from the event loop, at the start of each event. First, the IOV keys for the current event are saved in a ring buffer with N_{event} entries. Each conditions container that was earlier scheduled for cleaning at this time is then

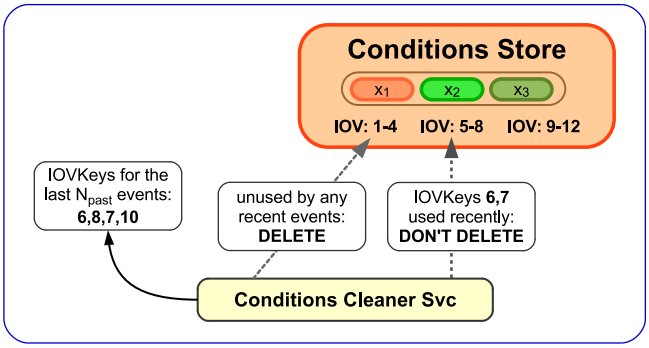

Figure 4. Garbage Collection in AthenaMT. Conditions Cleaner Service scans a conditions container N_{delay} events after a new object is added.

examined. The earliest conditions objects in these containers that do not match any event currently being processed or the keys for any of the past N_{event} saved in the ring buffer are then deleted. The parameters N_{delay} and N_{event} are preliminarily set to 100, but will be tuned based on further experience.

6 Migration Status

The framework and infrastructure components of the concurrent condition store handling in AthenaMT is feature complete, and is in the process of undergoing optimization. It supports both serial and concurrent processing environments.

The migration of clients to this environment has proved to be more challenging than initially anticipated. This is largely due to the construction of the callback functions which were used to update derived conditions in serial Athena. These were often implemented as components called AlgTools, which act as callable functions that can be shared between multiple parent Algorithms. They tended to do significant caching of event related data. Both of these implementation methodologies are not allowed in AthenaMT, as they result in thread and concurrent event unsafe behavior. Converting these into Conditions Algorithms has required re-writing significant amounts of the code, and redesigning interfaces.

In general, rewriting client code to read conditions data from the conditions store has been much more straightforward, as it usually only requires replacing direct access to the store with an equivalent ReadCondHandle. This aspect of the migration is largely complete.

Significant effort is now being directed to the client migration, and we hope to have the majority completed by the end of Q4 2018.

7 Conclusion

Designing a conditions handling mechanism that can efficiently manage multiple conditions data belonging to different concurrent events has been challenging. There is a constant trade off between processing and memory efficiency, and complexity. However, given the computational requirement and available resources for Run 3 at the LHC, it is essential to implement a design that minimizes the use of resources. We have done so with a shared multi-cache implementation that has a adjustable memory profile, where the aggressiveness of the garbage collection can be tuned to meet the specific job requirements.

The migration of client code has proved to be more time consuming than originally anticipated. This is due to the structure of the code which updates derived conditions, which was both thread and concurrent event processing hostile, and needed significant changes to operate in AthenaMT. The migration of clients is underway, and effort to do so has increased as the scale of the challenge has become apparent.

References

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

[2] G. Barrand *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).

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

[7] P. Calafiura, C. G. Leggett, D. R. Quarrie, H. Ma and S. Rajagopalan, "The StoreGate: A Data model for the Atlas software architecture," (*eConf C* **0303241**, MOJT008 2003) [cs/0306089 [cs-se]].