Upgrade of ATLAS data quality monitoring for multithreaded reconstruction

Tomasz Bold, Walter Lampl, Rohin Narayan, Peter Onyisi, Piotr Sarna, for the ATLAS Collaboration

CHEP, 12 July 2018
Scope

- “Monitoring” here = “making histograms” (& other ROOT objects)
  - Online (control room) → status of detectors
  - Offline (post-reconstruction) → status of detectors, calibrations, reco software
- Specifically covering changes in “AthenaMonitoring”
  - i.e., monitoring that uses the Athena framework to make histograms and runs as part of the reconstruction workflow
- Not covering visualization, automated extraction of DQM decisions, persistency ...

online apps: see talk by Serguei Kolos
Athena Execution

- A sequence of *algorithms* is run in an order determined by Python configuration
- Algorithms can execute *tools* and interact with *services*
  - e.g., THistSvc service to persist TH1, etc. into ROOT files
- Event data read from/written to whiteboard
Current Implementation

• AthenaMonitoring is part of the standard Athena codebase
  − provides base classes for Athena tools that generate histograms
  − generic Athena algorithms manage monitoring tools
  − provides top-level configuration of monitoring depending on reconstruction settings

• Runs in several modes
  − offline: either directly in reconstruction or as an afterburner
  − online: specialized Athena jobs that sample & reconstruct raw events as they are being recorded

• DQM is memory-intensive
  − a typical single-core offline reconstruction job: ~ 80k histograms, ~ 1 GB in memory
DQ Code Anatomy

- User code subclasses a parent monitoring tool class
- User code is responsible for:
  - initial memory allocation of histogram
  - histogram binning & properties (declared in C++)
  - holding raw pointers to histograms & filling them
- Common code is responsible for:
  - applying event filters (trigger, standard cleaning, ...)
  - storing the histograms in appropriate locations in output ROOT files
  - rebooking histograms when necessary (e.g. for time-dependence)

Generic Manager

User Tool 1
User Tool 2
User Tool 3

Athena “Algorithm”

instance of standard AthenaMonManager class
directly scheduled in reconstruction

Athena “Tools”

instances of user subclasses of ManagedMonitorToolBase
executed by parent algorithm

Monitoring algos in a typical Athena reco job

Athena MonManager
JetTagMonManager
RpcLvl1RawMonManager
RpcLvl1SLRawMonManager
CscRdoBsRawMonManager
MdtRawMonManager
RpcRawMonManager
RpcTrackMonManager
RpcLvl1RawEfficiencyMonManager
TgcRawMonManager
TgcLvl1RawMonManager
CscPrdRawMonManager
CscClusterEsdRawMonManager
CscSegmEsdMonManager
MdtVsRpcRawMonManager
MdtVsTgcRawMonManager
MuonTrackMonManager
LucidMonManager
Challenges in Existing Framework

- User code declares histograms & other objects
  - C++ declaration → **changing histogram binning** often requires new Athena releases!
  - user code needs to know **underlying technology**, e.g. TH1F vs TH1D vs TProfile vs non-ROOT
  - Adding new types of objects to the framework that aren’t subclasses of TH1 / TGraph / TTree (e.g. TEfficiency) is **hard**: ROOT object memory, I/O semantics are wildly variable

- User code has raw pointers to histograms, controls memory by default
  - strongly limits what central management code can do, e.g. implementation of **rebooking of time-dependent histograms** is very fragile
  - transitions to new technologies have uneven adoption
The Multithreaded Future

- LHC Run 3: Athena → AthenaMT
  - multiple threads, multiple events in flight
  - reduce memory/core by reducing # of needed algorithm instances in a job
- Algorithms need not be scheduled by user
  - algorithms & tools declare dependencies via the data they consume/produce
  - scheduler builds dependency graph, determines order of execution

- For thread-safety, algorithms can be
  - legacy – only one instance, use serialized;
  - cloneable – multiple instances, with separate memory;
  - re-entrant – single instance, simultaneous use by multiple threads

See talk by Scott Snyder
Concurrency & Multithreading

- Strictly speaking, algorithm concurrency is not really needed for monitoring code in AthenaMT
  - many relatively small pieces of code, produce no event data so no interdependencies → can execute multiple monitoring algorithms in parallel
  - do not want to clone monitoring algorithms – this clones their histograms, with huge memory cost!
  - re-entrancy is necessary only for massive parallelism, requires thread-safe histograms

- Monitoring algorithms could see events out of order
  - upsets assumptions of strict time-ordering
Conceptual Design

- Move management, filling of histograms to centralized code
  - user code declares variables to be monitored
  - central code fills and manages histograms
  - central code configured to fill histograms with Python scripts
  - underlying technologies can be changed, adapted without changing user code

- Adopts a solution proposed for monitoring in the High Level Trigger upgrade
Practical Example

**C++:** definition of monitored variables

```cpp
using namespace Monitored;

auto et = MonitoredScalar::declare<float>("Et", 0);
auto ex = MonitoredScalar::declare<float>("Ex", 0);
auto ey = MonitoredScalar::declare<float>("Ey", 0);
auto phi = MonitoredScalar::declare<float>("Phi", 0);
auto sumet = MonitoredScalar::declare<float>("SumEt", 0);
auto sourceIndex = MonitoredScalar::declare<int>("SourceIndex", 0);  
```

C++: retrieval of monitored information into variables

```cpp
auto monTool = m_sourcesMonTools[ m_sourcesToolMap[xaod_key] ];

ex = (*xMissEt)[xaod_subkey]->mpx() / CLHEP::GeV;
ey = (*xMissEt)[xaod_subkey]->mpy() / CLHEP::GeV;
et = (*xMissEt)[xaod_subkey]->met() / CLHEP::GeV;

if (et > 0.1) {
  phi = (*xMissEt)[xaod_subkey]->phi();
  sumet = (*xMissEt)[xaod_subkey]->sumet() / CLHEP::GeV;
  monTool.sourceIndex = Monitored::save( monTool, et, ex, ey, phi, sumet );
}
```

**Python:** definition of histograms to book

```python
from AthenaMonitoring.GenericMonitoringTool import GenericMonitoringTool, defineHistogram

def srcMonTool( src, etRange, nEtBins=100, nPhiBins=100, doProfiles=False ):
  mon = GenericMonitoringTool(src)
  mon.HistPath = "Srcs"
  mon.ExplicitBooking = True

  mon.Histograms = [
      defineHistogram( "Et;Et;" + src, title="Et Distribution (%s);MET Et (GeV);Events" % src, xbins = nEtBins, xmin = 0.0, xmax = etRange ),
      defineHistogram( "Ex;Ex;" + src, title="Ex Distribution (%s);MET Et (GeV);Events" % src, xbins = nEtBins, xmin = etRange, xmax = etRange ),
      defineHistogram( "Ey;Ey;" + src, title="Ey Distribution (%s);MET Et (GeV);Events" % src, xbins = nEtBins, xmin = -etRange, xmax = etRange ),
      defineHistogram( "Phi;Phi;" + src, title="Phi Distribution (%s);MET Phi (radian);Events" % src, xbins = nPhiBins, xmin = -math.pi, xmax = math.pi ),
      defineHistogram( "SumEt;SumEt;" + src, title="SumEt Distribution (%s);SumEt (GeV);Events" % src, xbins = nEtBins, xmin = 0.0, xmax = etRange*10 ),
      defineHistogram( "SumEt;metVsSumEt;" + src, title="MET Vs SumEt Distribution (%s);SumEt (GeV);MET Et (GeV)" % src, xbins = nEtBins, xmin = -etRange*10, xmax = etRange*10 ),
      defineHistogram( "Phi, Et, metVsPhi;" + src, title="MET Vs MetPhi Distribution (%s);MET Phi (radian);MET Et (GeV)" % src, xbins = nPhiBins, xmin = -etRange*10, xmax = etRange*10 ),
  ]
  return mon
```

**Configure one histogram filling tool per MET source**

- **1D histogram**
  - `defineHistogram("Et;Et;" + src, title="Et Distribution (%s);MET Et (GeV);Events" % src, xbins = nEtBins, xmin = 0.0, xmax = etRange )`

- **2D histogram**
  - `defineHistogram("SumEt;metVsSumEt;" + src, title="MET Vs SumEt Distribution (%s);SumEt (GeV);MET Et (GeV)" % src, xbins = nEtBins, xmin = -etRange*10, xmax = etRange*10 )`

**Retrieve data**

- `ex = (*xMissEt)[xaod_subkey]->mpx() / CLHEP::GeV;`

**Take snapshot**

- `monTool.sourceIndex = Monitored::save( monTool, et, ex, ey, phi, sumet );`
Lessons Learned

Discoveries about a 10+ year old codebase...

- Many features of the current system are unused
  - will remove and simplify

- Users use many ROOT features
  - things like alphanumeric bin labels, varying bin widths, etc. are easy enough to implement
  - need to be able to make all standard ROOT object types (TH*, TProfile, TGraph, TTree ...)

- Logic for quantities of interest can be complex
  - example: “distribution of mean per-event occupancy over the many modules of this subdetector”
  - not possible to express as a purely per-event fill operation; requires a postprocessing step after accumulation – map/reduce

\[ \text{e.g. “fill”-length histograms, annotations for parent trigger chain} \]
“No-build alternative”

- "No-build": what if we tried to change as little as possible? Development work is costly...

- Make existing monitoring algorithms “legacy” - only one instance of each in an AthenaMT job, access serialized
  
  fine in principle as long as access to ROOT files is thread-safe

- Not truly “no-build”:
  - all AthenaMonitoring code needs some modification to enable the AthenaMT scheduler to work,
  - the ATLAS-developed sparse histogram solution is not thread-safe and not planned to be modified

- Strongest motivation to change: allow us to replace backends without user intervention
  - e.g. migration to ROOT 7 histograms can be handled centrally

- “No-build” and “build” are not exclusive
Summary

- ATLAS framework for data quality monitoring in Run 1 and 2 has worked well, but is inflexible, fragile, and requires a lot of users
- Transition to multithreaded AthenaMT framework in Run 3 → good opportunity to move to a more flexible system
  - separate extraction of data features from histogramming code
  - ease introduction of new features by centralizing implementation
  - prototype code in place
  - Fallback to “Run 2-style” monitoring is possible but user code needs to adapt to AthenaMT regardless
- Requires an audit of how people currently use the monitoring
  - attempt to address all practical use cases

Monitoring birds-of-a-feather session: 1 pm in Room 7.2