University of Wuppertal
School of Mathematics and Natural Science
Experimental Particle Physics

BERGISCHE
UNIVERSITÄT
WUPPERTAL

Wuppertal, 21.06.2018

Reading ATLAS xAODs with xAODdataSource
First Results and Plans

Umesh Worlikar

Introduction

About Me:
I am a Masters student of **Computer Simulation in Sceince [CSiS]** program
at Bergische University Wuppertal

Also working as a student assistant in Experimental Physics Dept. [Computing Team]
Project: PyJobTransforms Framework, a python interface layer around Athena framework

Current Work:
I am currently working on my thesis task, which initially began with an objective
to seek a generic solution for performance optimization of ATLAS analysis code.

This was further narrowed down to prototyping of **xAODdataSource**, a “Proof of Concept”, aimed
to bring performance benefits (and brevity) of ROOT’s **TDataFrame** to ATLAS analysis code.

Agenda:
1. Brief overview of xAODdataSource
2. Simple analysis use cases with TDataFrame and xAODdataSource
3. Initial performance test observations
4. Current status and plan

xAOD DataSource: Overview

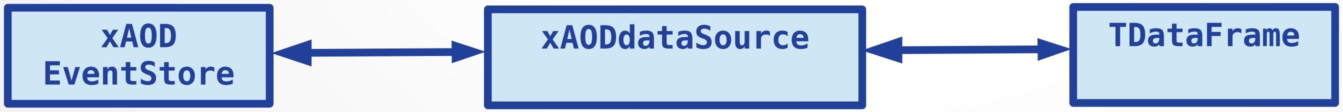
TDataFrame offers:

Implicite Parallelism → Hide complexity of multi-threading from Analysis User
Expressive interfaces → Simplified User Analysis Code

However, for complex data formats
Parallel access to nested data elements is not trivial

To support arbitrary data formats,
TDataFrame relies on experiment specific data sources for task decomposition and data access.

For ATLAS xAODs, the **xAODdataSource** serves this purpose.
It’s an adapter, that connects **TDataFrame** with **xAODs**



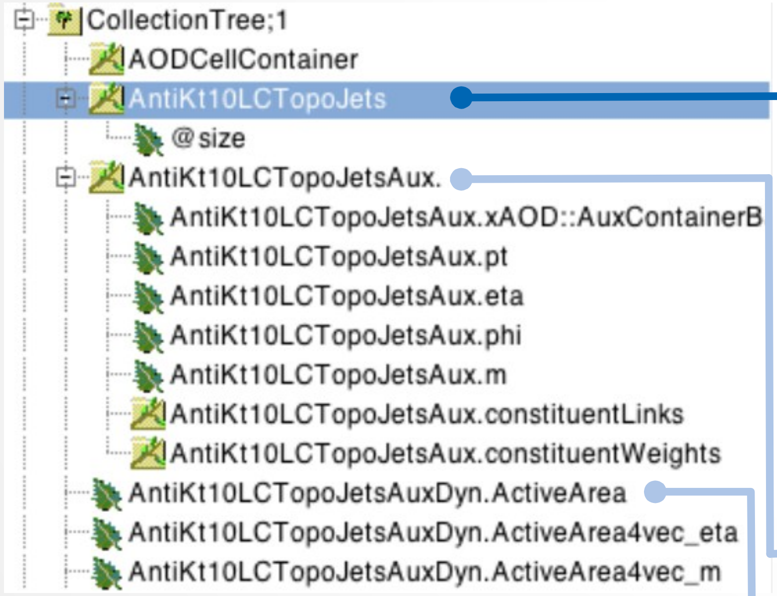
Reads persistent xAOD data from root files into transient objects

Distributes Events into Tasks
Initializes data access pointers for each event, when requested by TDataFrame

Executes event loop in parallel

xAOD Format: Overview

ATLAS xAOD format uses **Event Data Model**[1], a framework that maps physics objects into a scheme of interfaces and classes. Event data is stored in a **CollectionTree,** which contains **Interface Containers.** Each one associated with it's Auxiliary Store.



Interface Container

Provides consistent interface for all storage types
Hides complexity of storage technologies from user code

Standalone objects

Vector Containers

xAODdataSource uses these to access data elements through xAOD **EventStore**

Actual storage is handled by these under the hood

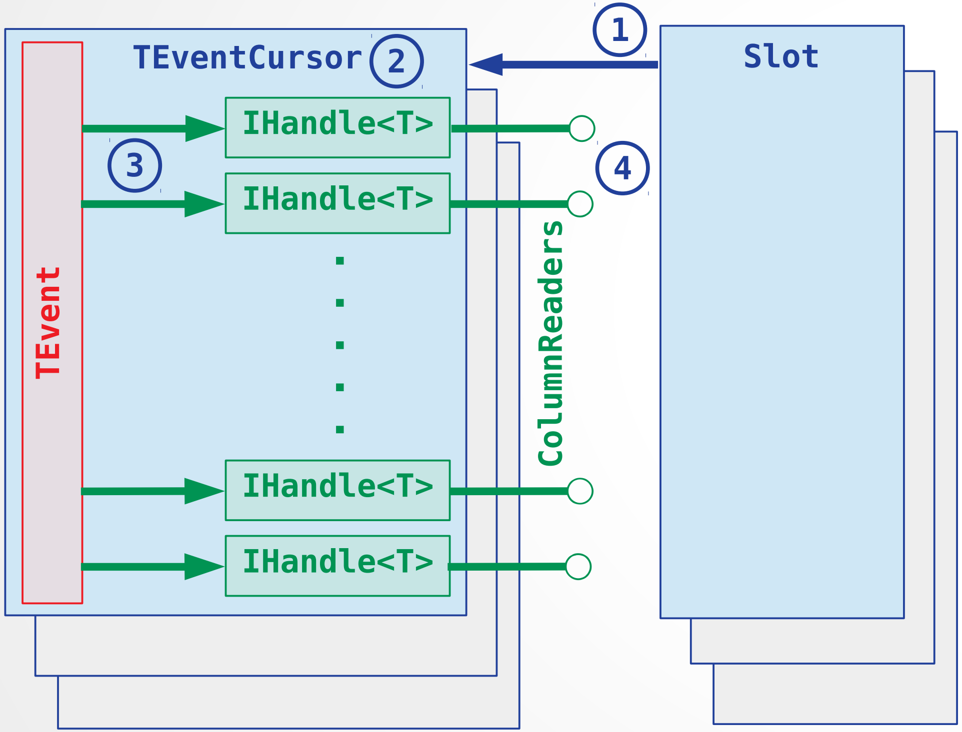
AuxStore — Compile time properties

AuxStoreDyn — Run time data derived properties/decorations

[1] General features of ATLAS Event Data Model

xAODdataSource: Overview

xAODdataSource's parallel-connection of **TDataFrame** with xAOD **EventStore**



During Initialization:
xAODdataSource creates multiple **TEventCursors** [one per slot]

Each **TEventCursor** holds:
→ One instance of xAOD::**TEvent**
→ A set of **IHandles**

Each Handle provides one **ColumnReader,** a pointer through which **TDataFrame** gets data access

When "updated", the Handle retrieves data from EventStore, thus ColumnReader gets its pointer initialized to expected memory location

During EventLoop:
1. TDataFrame concurrently calls **SetEntry** on all slots
2. Each TEventCursor calls **Update** on all its handles
3. Each handle retrieves data from EventStore
4. TDataFrame accesses data through ColumnReaders

Note: Handles are created on-demand
Number of Handles == Number of variables actually accessed by analysis code

Functional Tests

Functional Tests: Currently three basic use cases are considered

1. Read standalone xAOD objects, perform simple analysis tasks

2. Read xAOD vectors, perform simple analysis tasks

3. Read xAOD vectors, apply cut and filter some events

This validates:
Successful access to both basic categories of xAOD containers

Successful interaction with TDataFrame for common actions performed by analysis code

Note:
Although xAOD event store allows writing transient data into containers and make deep/shallow copies, current scope is initially set to test read-only workflow. More complex scenarios can be considered as future work.

Test Environment

```
Hardware       : Intel i7 Quad-Core CPU @3.6 GHz, 8GB RAM
               : Hyper Threading Disabled

OS             : Ubuntu 16.04.LTS
ROOT           : 6.13/02
AnalysisBase   : 21.2.31
xAODdataSource : 0.1.00

Data set       : data17_13TeV.00328263.physics_Main.deriv.DAOD_HIGG2D2.f836_m1824_p3213
               : Set of 50 compressed DxAOD files
               : Total size ~25GB
```

Analysis Example: Standalone xAOD objects

Use case 1: Read standalone xAOD object, Perform simple analysis tasks. [min, max and histogram]

```
int main(int argc, char* argv[]) {
   using namespace asg::msgUserCode;
   ANA_CHECK_SET_TYPE(int);   // Required since the return type is different than standard ATLAS StatusCode.

   AppUtils::TOptions options(argc, argv);
   options.interprete();
   if (options.isHelpRequested()) return 0;

   if (options.enableMT()) { ROOT::EnableImplicitMT(options.NThreads()); }

   ANA_CHECK(xAOD::Init()); // Set up the job for xAOD access.
   constexpr auto branchName = "EventInfo";
   constexpr auto runNumber = "RunNumber";
   constexpr auto eventNumber = "EventNumber";

   //Define transient columns mapping to xAOD functions EventInfo::runNumber(), EventInfo::eventNumber()
   auto xdf = xAOD::DataSource::make_xAODdataFrame(options.fileNamePattern(), AppUtils::DEFAULT_TREE_NAME);
   auto runNumberData = xdf.Define(runNumber, [](const EventInfo *ei) { return ei->runNumber(); }, {branchName});
   auto eventNumberData = xdf.Define(eventNumber, [](const EventInfo *ei) { return ei->eventNumber(); }, {branchName});

   //Apply lazy actions
   auto eventNumberMin = eventNumberData.Min({eventNumber});
   auto eventNumberMax = eventNumberData.Max({eventNumber});
   auto runNumberHist = runNumberData.Histo1D(runNumber);

   //Access data
   auto defaultSettings = std::cout.flags();
   ANA_MSG_INFO("EventInfo.eventNumber Range: [" << std::setprecision(16) << eventNumberMin.GetValue()
                                    << ", " << eventNumberMax.GetValue() << "]");

   ANA_MSG_INFO("Histogram EventInfo.runNumber:");
   ANA_MSG_INFO("Total entries: " << runNumberHist->GetEntries());
   ANA_MSG_INFO("Range: [" << runNumberHist->GetMinimum() << ", " << runNumberHist->GetMaximum() << "]");
   ANA_MSG_INFO("Mean: " << runNumberHist->GetMean() << " StdDev: " << runNumberHist->GetStdDev());
   std::cout.flags(defaultSettings);

   return 0;
}
```

Enable ImpliciteMT

Setup DataFrame's lazy actions

From xAOD user's perspective typical analysis code would look & feel like any other data format used within ROOT

Simplified code
Relatively easy configuration

Create DataFrame
Define columns of nested variables

Access data
Triggers event loop

Event loop is completely hidden from user

Analysis Example: Vector Containers

Use case 2: Read xAOD vector container, Create a histogram.

```
int main(int argc, char* argv[]) {
    using namespace asg::msgUserCode;
    ANA_CHECK_SET_TYPE(int);   // Required since the return type is different than standard ATLAS StatusCode.

    AppUtils::TOptions options(argc, argv);
    options.interprete();
    if (options.isHelpRequested()) return 0;
    if (options.enableMT()) { ROOT::EnableImplicitMT(options.NThreads()); }

    ANA_CHECK(xAOD::Init()); // Set up the job for xAOD access.
    auto xdf = xAOD::DataSource::make_xAODdataFrame(options.fileNamePattern(), AppUtils::DEFAULT_TREE_NAME);

    // Extract vector variable from AuxContainer
    const auto readPt = [](const xAOD::IParticleContainer* particles) {
        std::vector<double> ptVector{};
        for (const auto& p: *particles) {
            ptVector.push_back(p->pt());
        }
        return ptVector;
    };

    constexpr auto ptColumnName = "pt";
    auto ptData = xdf.Define(ptColumnName, readPt, {"Electrons"});    //Define a new column
    const TH1DModel& histModel = {"histElectronPt", "histElectronPt", 100, 0.0, 1500.0};
    auto histElectronPt = ptData.Histo1D(histModel, ptColumnName);

    // Access data
    auto defaultSettings = std::cout.flags();
    ANA_MSG_INFO("Histogram Electorn.pt:");
    ANA_MSG_INFO("Total entries: " << std::setprecision(16) << histElectronPt->GetEntries());
    ANA_MSG_INFO("Range: [" << histElectronPt->GetMinimum() << ", " << histElectronPt->GetMaximum() << "]");
    ANA_MSG_INFO("Mean: " << histElectronPt->GetMean() << " StdDev: " << histElectronPt->GetStdDev());
    std::cout.flags(defaultSettings);

    return 0;
}
```

Enable impliciteMT
Create DataFrame

Mimic a user defined transform
This collects a vector from
an xAOD vector container

Use it with DataFrame's
Define action to derive
another column

Analysis Example: Event Filter

Use case 3: Read xAOD vector container, Apply a cut and Filter some events

```
int main(int argc, char* argv[]) {
    using namespace asg::msgUserCode;
    ANA_CHECK_SET_TYPE(int);   // Required since the return type is different than standard ATLAS StatusCode.

    AppUtils::TOptions options(argc, argv);
    options.interprete();
    if (options.isHelpRequested()) return 0;
    if (options.enableMT()) { ROOT::EnableImplicitMT(options.NThreads()); }

    ANA_CHECK(xAOD::Init()); // Set up the job for xAOD access.

    auto xdf = xAOD::DataSource::make_xAODdataFrame(options.fileNamePattern(), AppUtils::DEFAULT_TREE_NAME);
    auto totalEvents = xdf.Count();

    //Define a limiting function for pt variable.
    constexpr auto ptThreshold = 10000.0;
    const auto ptCut = [](const xAOD::IParticleContainer* particles) {
        std::vector<double> ptVector{};
        for (const auto& p: *particles) {
            if (p->pt() >= ptThreshold) { ptVector.push_back(p->pt()); };
        }
        return ptVector;
    };

    //Define a transient column to hold filtered data.
    constexpr auto ptCutColumn = "ptCutVector";
    auto ptCutData = xdf.Define(ptCutColumn, ptCut, {"Electrons"});

    //Define a lazy event filter and use it to derive more analysis variables.
    auto filteredEvents = ptCutData.Filter("ptCutVector.size() > 2");   // String used to test jitting.
    auto filteredEventCount = filteredEvents.Count();
    auto ptCutMin = filteredEvents.Min(ptCutColumn);
    auto ptCutMax = filteredEvents.Max(ptCutColumn);

    auto defaultSettings = std::cout.flags();
    ANA_MSG_INFO("Total events: " << std::setprecision(16) << totalEvents.GetValue());
    ANA_MSG_INFO("Filtered events: " << filteredEventCount.GetValue());
    ANA_MSG_INFO("Range{Electron.pt >= minPt}: " << "[" << ptCutMin.GetValue() << ", " << ptCutMax.GetValue() << "]");
```

Enable impliciteMT
Create DataFrame

Mimic a user defined cut
Collects a slice of data from an xAOD vector container

Use the cut function to define a column

Define an event filter with arbitrary predicate

Access filtered data

Performance Tests: Setup

Performance Tests:
Using the same test environment, several test runs were conducted using three compiled test applications corresponding to three use cases described in last section

Data set : data17_13TeV.00328263.physics_Main.deriv.DAOD_HIGG2D2.f836_m1824_p3213
: Set of 50 compressed DxAOD files
: Total size ~25GB

Tests were executed in two different modes

1. WarmCache Mode:
A warmup iteration is executed as a dry-run, followed by actual sample of 20 test executions. Mean wall time was observed for each sample.

2. ColdCache Mode:
File system cache was cleared before each test execution.

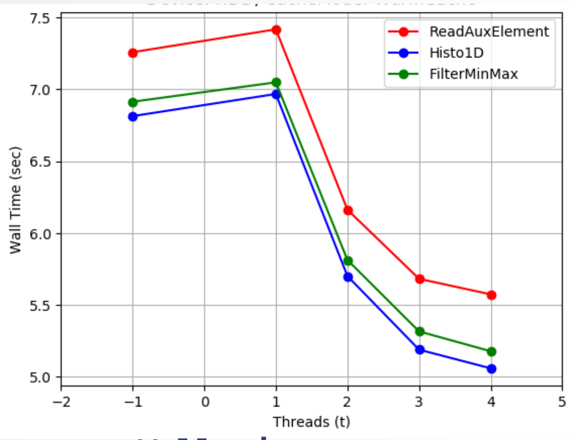
For both modes, all three applications were retested with data on spinning disk drive[HDD] and solid state drive[SSD].

The objective of these tests was to observe scalability relative to sequential performance.

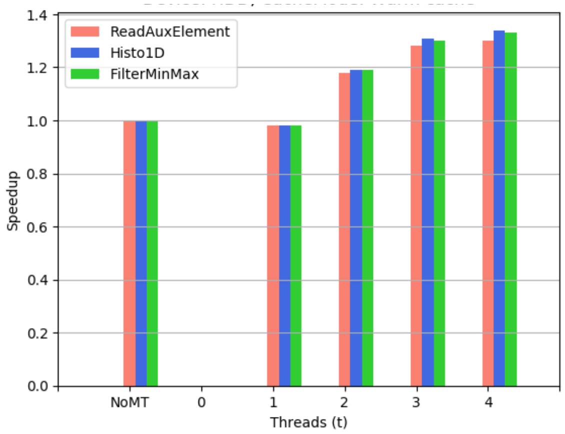
Cold cache mode was used to probe any possible effect of disk latency.

Test Observations: Warm Cache with HDD

Test Observations: Warm Cache, HDD



Wall time w.r.t. number of threads



Speedup relative to sequential execution

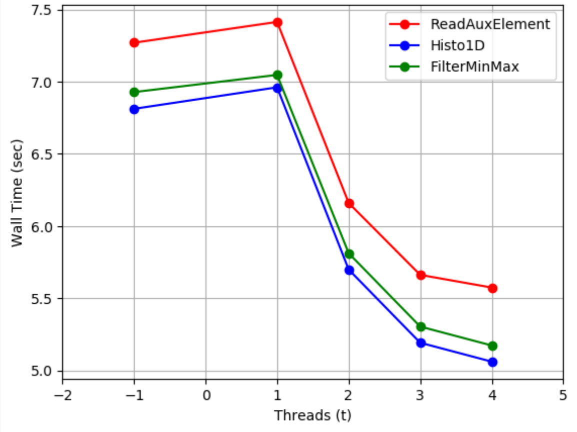
Speedup indicates possibility of performance bottlenecks

All three test cases exhibit similar scaling pattern

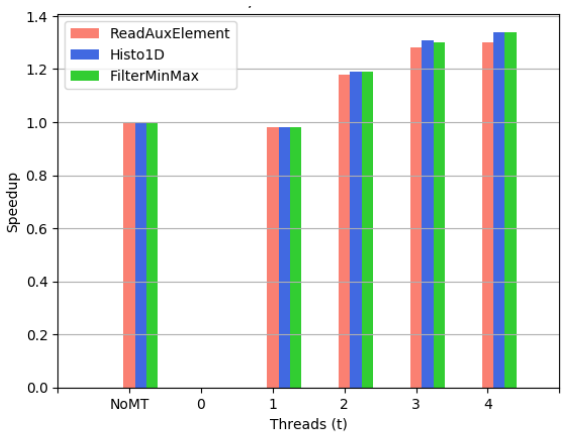
Potential overheads appear to be independent type of container or actions performed on them

Test Observations: Warm Cache with SSD

Test Observations: Warm Cache, SSD



Wall time w.r.t. number of threads

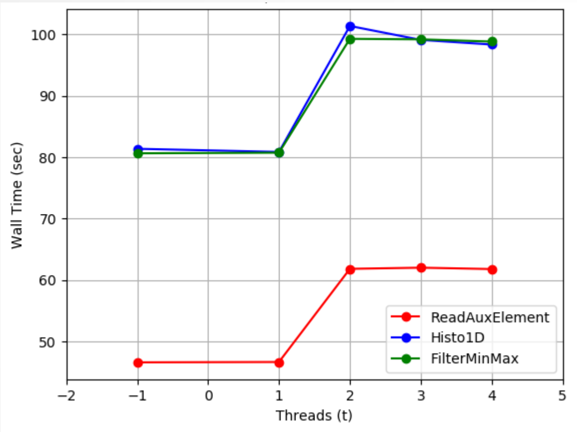


Speedup relative to sequential execution

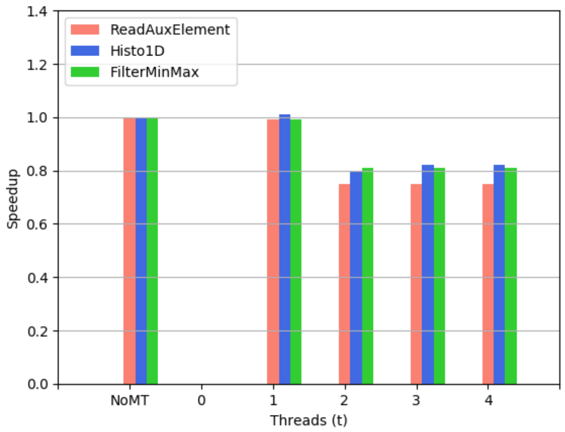
Almost same results as that of last observation on HDD. Type of disk shows no effect on speedup in this case.

Test Observations: Cold Cache with HDD

Test Observations: Cold Cache, HDD



Wall time w.r.t. number of threads



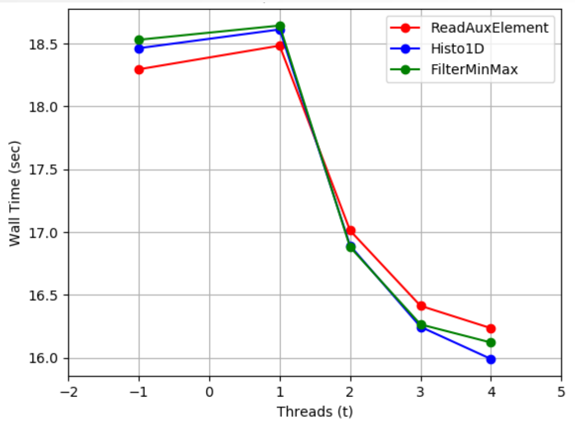
Speedup relative to sequential execution

For t > 1
Performance degrades,
Speedup stalls

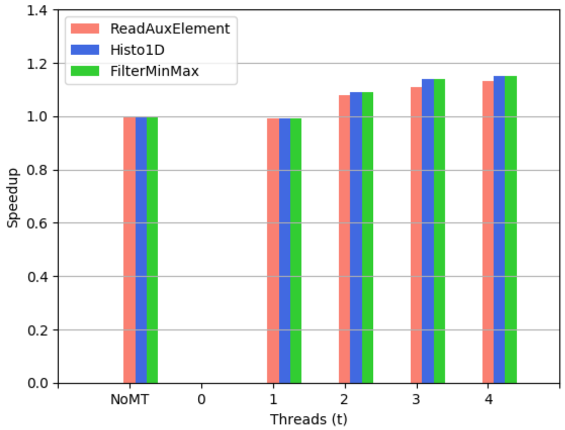
Wall time difference is significantly large between standalone and vector containers

Test Observations: Cold Cache with SSD

Test Observations: Cold Cache, SSD



Wall time w.r.t. number of threads



Speedup relative to sequential execution

Almost same speedup pattern as that of Warm Cache mode observed on both SSD & HDD.

Only difference is additional wall time, a consistent offset, which can be safely attributed to Cold Cache miss.

Remark:
In Cold Cache mode, spinning disk does affect performance.
However, even with better hardware, the pattern of sub-optimal speedup is still persistent.
At least for use cases with low IO traffic,
The effect of disk latency on speedup may be safely ruled out.

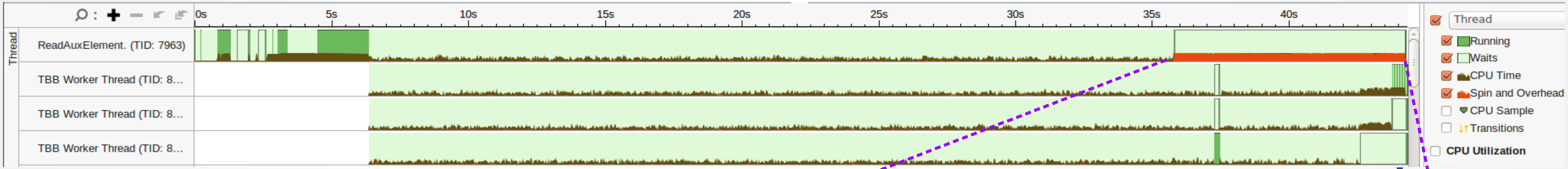
Profiling Observations

Preliminary Profiling Observations: [Work in progress]

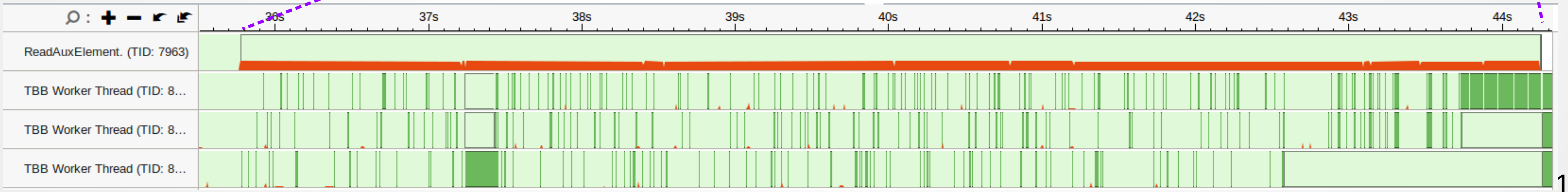
Concurrency timeline of Histo1D test case with 4 threads

<u>Two possible issues</u>

1. **Significant spin time**
   Workload imbalance



2. Low concurrency values throughout execution
Average number of simultaneously active threads < 2.3



Current Status

Currently identified problems:
Based on preliminary observations, two possible sources of bottlenecks are identified.

1. **Workload Imbalance:**
   XAODdataSource version [0.1.00] uses trivial uniform task distribution
   Number of Tasks == Number of Slots,
   Each Task has equal number of events.

   For effective task stealing, TDataFrame expects Number of Tasks >> Number of Slots
   With assistance from ROOT developers, I am currently working on solution for optimized workload distribution.

2. **Low Concurrency:**
   Initial profiling results, hint towards a mutex-lock in ROOT,
   However, at the moment it’s unclear whether it is apparent side effect of item 1 above.
   Worst, it could be a combination of load imbalance and lock contention.

Current Plan: Address item 1, then re-test and re-profile to reproduce item 2.

Conclusions

Conclusions:

1. **We have a “Proof of Concept” for xAODdataSource, which works!**
2. **Lot of scope for performance improvement**
3. **Limited Analysis work-flow supported at the moment**
   Community feedback would be of great help
   → Identification of more realistic use cases
   → Integration with existing Tools/Frameworks
   → Pre/post-EventLoop configurations/executions
   → Investigate effects of compression methods on performance

Questions?