Framework Extensions

Hadrien Grasland

LAL – Orsay

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

Task 3.6 – Framework Extensions

- Original task objectives (from AIDA2020 proposal):
  - Parallel algorithm scheduling for HEP frameworks
  - To be developed in a framework-independent way
  - Then integrated in Gaudi, Marlin, PandoraPFA
- However, parallel Gaudi algorithm scheduling work was completed before AIDA-2020 started
- Decided to refocus on another obstacle to Gaudi parallelization, namely detector condition handling

Why conditions?

- Condition data is a subset of detector state which…
  - …is **time-dependent**, but not **event-specific**
  - …is used during event processing
- Historically handled by experiments, as ~global variables
  - Not supported by core Gaudi → Much effort duplication
  - Common strategy: Update global state between events
  - Interacts badly with parallel event processing

Project status

- Done:
  - Devise a design which can work for any Gaudi user
  - Demonstrate and validate it through prototyping
- In progress:
  - Add some missing Gaudi infrastructure
- Pending:
  - Integrate condition handling prototype in Gaudi

Requirements

Main design constraints

- Condition data changes **rarely** (~1/1000 events at worst), but event time budget can be **tight** (30 MHz HLT in LHCb)
  - Design for fast condition **readout** and per-event **scheduling**
  - Aim for zero overhead when condition data doesn’t change
  - Some overhead on writes is okay in exchange for fast reads

Main design constraints

- Condition data changes **rarely** (~1/1000 events at worst), but event time budget can be **tight** (30 MHz HLT in LHCb)
  - Design for fast condition **readout** and per-event **scheduling**
  - Aim for zero overhead when condition data doesn't change
  - Some overhead on writes is okay in exchange for fast reads
- **Multiple detector states** in flight is a doubled edged sword
  - Use cases exist (e.g. shuffled events at IoV boundaries)
  - Makes migration harder (can't reuse old globals initially)
  - Can cause memory bloat if done in an uncontrolled fashion
  - Best tradeoff: Configurable bound on amount of states

Other constraints

- Reuse unchanged data to save RAM & bandwidth
- Hide condition storage implementation from user
- Overlap condition IO/derivation with event processing
- Track condition usage and discard unused state
- Provide cross-experiment interfaces and allow for experiment-specific implementations of IO & derivation

Framework concepts

Storage metaphor

- Manage full detector states as **ConditionSlots**
- **Slot IoV (interval of validity) =** Intersection of inner IoVs

- This design has many nice properties:
  - Maximal flexibility of RAM storage implementation
  - Intuitive bound on condition storage (“N slots at most”)
  - Event validity tracking is cheap (check slot-wide IoV only)
  - Condition usage tracking is cheap (use slot-wide refcount)
  - Data access can be cheap (slot ID = index in arrays of data)
  - Can still share unchanged data between condition slots

Data access

- Algorithms & al declare **Condition[Read|Write]Handles**
- Handle + EventContext → Access to the condition data

- Again, there are many benefits:
  - Consistent with event data access*
  - Track condition usage and pre-allocate storage
  - Tell who reads/writes what (for scheduling, error handling…)
  - Easily discriminate condition versions (via EventContext)

* Or at least should be… more on that later

Allocation & initialization

- Events get a ConditionSlot before processing starts
- This process may not be instantaneous (e.g. if all slots are busy, or if waiting for ongoing condition IO/derivation…)

- Use an asynchronous interface to avoid forced blocking:

```
ConditionSlotFuture setupConditions( const framework::TimePoint & eventTimestamp );
```

  - With a future, the event loop can choose to block or not
  - Blocking is fine as a first implementation
  - Can easily optimize later (e.g. reorder ready events first)
  - Good future impls also simplify condition IO & derivation

ConditionDB I/O

- Multiple ConditionDB interfaces, most restrictive is COOL
  - Cannot know what is the IoV of a condition before fetching it
  - Means we can only do IO for one ConditionSlot at a time
  - …and it has an influence on slot allocation (IoV needed then)

- Can be modeled via asynchronous condition IO services:

```
virtual cpp_next::future<void> startConditionIO( const framework::TimePoint   & eventTimestamp,
                                                 const ConditionSlotIteration & targetSlot ) = 0;
```

  - Future-based interface integrates trivially with slot allocation
  - Blocking IO threads, if any, are an implementation detail
  - Design is somewhat polemical on the Gaudi side, though…

Condition derivation

- Some conditions require post-processing (e.g. alignments)
- Everyone thinks this should have an Alg-like interface
  - ...which is the most important part, really
- Implementation, however, is quite polemical
  - Question is, should the Gaudi Scheduler be executing them?
  - Is it worth making this arcane code even more complex?
  - Can it efficiently handle two very different time scales?
  - Can it scale to large numbers of rarely changing conditions?

Running prototype

- All these concepts have passed the implementation test: https://gitlab.cern.ch/hgraslan/conditions-prototype

- Early performance numbers* are satisfactory:
  - Scheduling an event with “hot” conditions takes **~5.4 µs**
  - Reading a condition from an Alg takes **~10 ns**
  - Writing a condition takes ~0.3 µs
  - Scheduling condition IO takes ~(12.3 + 0.3 x N_{cond}) µs
  - Deriving conditions takes ~(1.0 + 0.1 x N_{alg} + 0.3 x $N_{cond,out}$) µs
  - **No synchronization** on reads, fine-grained locks elsewhere

* Test scenario : Batches of 10k events, 10k conditions in flight, on an Intel Xeon E5-1620 v3 @ 3.50GHz (4 cores + HT)

Gaudi integration

Down a rabbit hole…

- The condition handling design uses **reentrant data handles**
  - Basically a proxy to versioned data (event data, conditions...)
  - An EventContext is passed in to select the right version
  - Also tracks data dependencies: simple and effective
- As it turns out, these were not ready yet:
  - Reentrant data handles are currently heavily ATLAS-specific
  - ATLAS started to integrate them into Gaudi… then stopped
  - Basically, these are wanted, but no one is working on them
  - So I resumed this development effort

…and digging deeper

- Some needed Gaudi infrastructure was missing
  - Current framework interface leaks too much information
  - Properties (aka configurables) are not flexible enough yet
- I started working on it
  - Support for non-default-constructible Properties is merged
  - Less leaky framework interface redesign under discussion
  - Tracked @ https://gitlab.cern.ch/gaudi/Gaudi/merge_requests/462

Summary

- Multi-threaded Gaudi needs better condition support…
  - …which, in turn, require reentrant data handles…
    - …which, in turn, require other framework clean up and rework
- All of this is being worked on
  - From the depths of the framework, going upwards

Questions? Comments?