Common condition handling

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.

Motivation

- Gaudi has no standard condition handling infrastructure
- Condition handling code historically experiment-specific
- …and based on thread-hostile global detector state

- We could keep fixing the experiment-specific code
- We could also standardize this important feature in Gaudi
  - More work in the short term, but less in the long term!

Requirements

ATLAS conditions

- HLT input ~75 kHz, with L2 time budget ~40ms/event
- ~300 raw conditions, ~10 of which are **short-lived**
  - Worst case: Calo noise bursts, every minute, last ~200ms
- RAM storage: ConditionStore (Global store of IoV arrays)
- ConditionDB based on COOL, may change in Run 3

- Requirements:
  - Short IoVs require keeping **multiple detector states in flight**
  - State storage must be **memory-efficient** (share reused data)
  - Condition IO & compute must **overlap** with event processing

LHCb conditions

- Run 3: 40 MHz SW trigger, ~3k HLT nodes → **~75 µs/event**
- **~10,000** raw conditions, usually valid for **one run** (~hours!)
- RAM storage managed by Gaudi today, DDCond tomorrow?
- ConditionDB historically used COOL, now Git-based

- Consequences:
  - Need **very fast event scheduling** (~µs) in the common case of unchanging conditions, **avoid O(N_{cond}) work per event**
  - Condition **readout** must be very fast and scalable
  - Must abstract away location of in-RAM condition state

Common requirements

- GaudiHive is all about RAM consumption
  - Must be able to bound amount of detector state in flight
  - Must discard unused conditions (but not too eagerly)
- IO should be a first-class citizen
  - Blocking IO in regular Algs is a Very Bad idea
  - Even dedicated IO Algs will only scale so far…
- Need an abstraction for condition derivation computations
  - e.g. computing detector alignments

Design proposal

ConditionSlot

- A **ConditionSlot** represents storage for a full detector state
- A fully filled slot has an **interval of validity (IoV)**, which is the intersection of the IoVs of its inner conditions

- How does this abstraction help us?
  - Largest granularity = maximal storage backend compatibility
  - Easy to bound RAM footprint (via amount of slots in flight)
  - Cheap usage tracking (by reference-counting entire slots)
  - Cheap condition reuse on new event (one IoV check per slot)
  - Can still share individual condition data blobs across slots

Slot allocation

- When a new event comes in, it is allocated a ConditionSlot
- This process may not be instantaneous:
  - All available condition slots may be used up
  - Related condition IO / computation may already be ongoing
- Blocking the event loop in this case should be avoided
  - We might be able to process other incoming events
- Modern* C++ approach: use an asynchronous interface

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

* Notably used in C++ Concurrency TS, Ste||ar HPX, just::thread, boost::thread, Qt Concurrent… Also imperfectly introduced in C++11/14/17, whose futures are hostile to asynchronous continuations and will thus compose poorly. The Concurrency TS aims to fix that.

Data access & dependencies

- Reentrant data handles are an **awesome** idea!
- They can solve so many different problems:
  - Tracking which conditions are used (for storage)
  - Tracking who reads/write each condition (for scheduling)
  - Discriminate between multiple versions of one condition
  - Efficiently access conditions (with suitable memory layout)
- So I enthusiastically designed around that concept

Condition IO caveats

- COOL peculiarity: Condition IoV is only known a posteriori
- This has unfortunate consequences on concurrency
  - ConditionSlot IoV is only known at the end of IO
  - We don’t know if two requests will have the same result
  - So we should only carry out IO for **one slot at a time**

- It’s not *that* bad (“real” IO is rare thanks to large caches)
- It means condition IO cannot be part of event processing
  - Need IO results to decide on ConditionSlot allocation!

Condition IO design

- See IO hackathon for more background discussion
- IO should probably be implemented as an async service
  - Maps system resources (e.g. network connections) well
  - Abstracts IO implementation (blocking/async, # IO threads)
  - Scales better to many conditions & IO requests
  - Composes trivially with asynchronous slot allocation
- Proposed interface:

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

Condition derivation

- After condition IO, some processing usually needed
  - e.g. computing an aligned detector geometry
- Everyone agrees that it should look like an Algorithm
  - Familiar abstraction, friendly to parallel processing
- Whether the Scheduler should handle it is debated
  - Must avoid $O(N_{cond})$ work in hot event processing path
  - Adds more complexity to an already complex Scheduler (very different time scales, broken design assumptions)
  - Many event scheduling features not needed for conditions

Conclusions

This actually works…

- Every part of the design presented above has been prototyped, tested and benchmarked
  - https://gitlab.cern.ch/hgraslan/conditions-prototype
- Some interesting prototype performance stats*
  - Scheduling an event with “hot” conditions takes **~5.4 µs**
  - Reading a condition takes **~10 ns**
  - Writing a condition takes ~0.3 µs
  - Scheduling condition IO takes $(12.3 + 0.3 \times N_{cond})$ µs
  - Deriving conditions takes $(1.0 + 0.1 \times N_{alg} + 0.3 \times N_{cond,out})$ µs
  - **No synchronization** on reads, otherwise fine-grained locking

* Tested with batches of 10k events, 10k raw conditions, on an Intel Xeon E5-1620 v3 @ 3.50GHz (4 cores + HT)

…but integration is stalled

- The design is strongly based on reentrant data handles
  - I think they are the right path, for so many reasons…
  - …but right now, they are **not** integrated in Gaudi

- Existing implementation: Athena's VarHandleKey
  - Identifies a transient store object (DataObjID + CLID)
  - VarHandleKey + EventContext => ReadHandle/WriteHandle
  - Problem: Previous integration attempt stalled due to interface incompatibilities between TES and StoreGate

Moving forward

- Some possible paths in order of decreasing preference
  1. Resolve incompatibilities and integrate reentrant handles
  2. Shrink the handle interface to reduce incompatibilities
  3. Build a TES-specific reentrant handle design
  4. Give up and just hack around non-reentrant handles

- Other issues to be resolved:
  - Interface compatibility with DD4hep / DDCond
  - Converge on IO & derivation scheduling

Questions? Comments?