Gaudi Conditions Handling

Hadrien Grasland (LAL), Benedikt Hegner (CERN)
18.5.2017

Current Gaudi Status

- Gaudi has no standard condition support
- So each experiment wrote its own
- Multi-threaded event processing now breaks everything
- What should we do next?
  - Keep maintaining duplicate codebases?
  - Converge towards a common approach?
  - Share more code between users?

We designed and wrote (*) prototypes and proposed them to the Gaudi community

(*) current prototype entirely up to Hadrien Grasland. Thanks!

Existing Use Patterns

- **Overall, conditions change very slowly w.r.t. event data**
  - At one extreme, LHCb conditions are valid for 1 run (~hours)
  - At the other, ATLAS has noise bursts: ~200ms every minute
  - Still thousands of events between IoV changes on average!
    - Mostly true for highly skimmed derived data sets as well as they usually don't use lower-level conditions

- **Event processing requirements vary between experiments**
  - LHCb: *~10k raw conditions, very long IoVs*, 40 MHz HLT on ~3k nodes → HLT node budget ~75 μs/event
  - ATLAS: *~300 raw conditions, ~10 of them can vary rapidly* (IoV < 1 minute). HLT node budget ~100ms/ev

Important Optimizations

- **LHCb: Take a fast path when conditions do not change**
  - As before, reuse previous raw & derived condition data
  - Avoid checking individual condition validity for every event
  - Minimize condition readout overhead in event processing
  - Drain & restart on changes
- **ATLAS: Keep multiple detector states in flight**
  - Do not duplicate rarely changing state (common case)
  - Handle out-of-order events on IoV boundaries efficiently
  - Process "new" conditions in parallel with "old" events
- **Diverse requirements, but compatible with each other!**

Requirements for a New Implementation

1. Support concurrent event processing
2. Keep RAM usage under control
3. Accommodate diverse storage backends
4. Allow efficient condition IO & computations
5. Easy to use, error-proof, and scalable
6. Experiment-agnostic, but reasonably compatible

Requirement:
Support for Concurrency / Memory Control

- **Multithreaded Gaudi is mainly about RAM usage**
  - Condition state should not grow indefinitely
  - Expose the ability of slice-based backends to set clear bounds on condition storage size
  - Transient storage for a detector state is called a *ConditionSlot* (similar to EventSlot)

- **Framework interface to conditions plays a key role here**
  - Limit number of ConditionSlots in flight
  - Allows backend to track condition usage and perform smart garbage collection
  - Allow for storage optimizations (sharing, lazy GC...)

Requirement: Efficient RAM Storage Backends

- **Anything that maps condition identifiers to condition data**
- **Many implementations exist or are being developed**
  - DetectorStore & public Tool members (alas...)
  - ATLAS ConditionStore (~ DetectorStore w/ vectors of data)
  - DDCond (condition storage for DD4Hep)
  - Needed to write another for the prototype...

- **Convergence on a single storage backend is desired, but unlikely**

- **Framework interface should be backend-agnostic**

Storage Interface Proposal

- Condition storage backends are interfaced through the *TransientConditionStorageSvc* concept:
  - Communicate implementation limits:

```
static size_t max_capacity();
```

  - Set up storage (capacity in ConditionSlots, 0=unbounded):

```
TransientConditionStorageSvc( const size_t capacity );
```

  - Query storage usage at runtime:

```
size_t availableStorage();
```

  - Track condition dataflow (see next slide)
  - Allocate/reuse condition storage for an incoming event:
    - Using a future allows delayed allocation (when storage is full)
    - C++11 futures aren't enough, need Concurrency TS (Boost, HPX...)
    - ConditionSlot liberation is automated through RAII

```
ConditionSlotFuture allocateSlot( const detail::TimePoint & eventTimestamp );
```

Dataflow Tracking - Condition Handles

- We need to track some condition usage metadata
  - For the backend to manage condition data correctly
  - For the scheduler to know data dependencies
- Condition users also need a way to access conditions
- We already know that problem from event data and handles

```
template< typename T >
ConditionReadHandle<T> registerInput( const detail::ConditionUserID & client,
                                      const detail::ConditionID     & targetID );

template< typename T >
ConditionWriteHandle<T> registerOutput( const detail::ConditionUserID & client,
                                        const detail::ConditionID     & targetID,
                                        const ConditionKind             targetKind );
```

Accessing Conditions Data

- Condition handles are a proxy to condition data
- Each condition user must request its handles separately
  ⇒ handles are movable, but not copyable
- Write handles allow producers to write condition data:

```
void put( const ConditionSlot    & slot,
          const ConditionData<T> & value ) const;
```

- Read handles allow consumers to read it later on:

```
const ConditionData<T> & get( const ConditionSlot & slot ) const;
```

- This interface allows powerful backend optimizations:
  - Write handles can also support moving data in
  - Reads can be implemented without synchronization

How does the prototype perform?

- **Condition handle prototype is reasonably fast**
  - *Writing* a condition takes *0.3 µs*
  - *Reading* a condition takes *10 ns*
  - Algorithm independent of N_{cond}, tested for 10K conditions
- **Easily *outperforming ATLAS' StoreGate*, used for event data:**
  - SG's algorithmic complexity is roughly $O(\log(N_{keys}))$
  - With 50 keys, writing ("record") takes 2.2 µs (*7.3x slower*)
  - ...and reading ("retrieve") takes 0.83 µs (*83x slower*)

Requirement:
Efficient Condition I/O and Computation

- **Gaudi scheduler was mostly designed for CPU-bound work**

- **Ongoing debate regarding how IO should be integrated**
  - ***IO tasks*** modeled as blocking Algs on extra OS threads?
    - Pros: Code reuse, familiar concepts, minimal scheduler rework
    - Cons: Inefficient, fragile, thread-unsafe by default, hard to use
  - ***IO resources*** modeled as asynchronous services?
    - Pros: No wasted RAM & context switches, thread-safe by default, global request awareness, this is where standard C++ is going, decoupling of concerns
    - Cons: Integration with algorithm scheduling is more difficult

- **Prototype interface can accommodate both designs**

I/O service proposal

- Models an IO resource (file, database…)
- On initialization, user specifies requested conditions

```
ConditionIOSvcBase( ConditionSvc            &  conditionService,
                    detail::ConditionIDSet && expectedOutputs );
```

- Service implementation registers appropriate handles

```
template< typename T >
ConditionWriteHandle<T> registerOutput( const detail::ConditionID & outputID );
```

- Framework then invokes IO services asynchronously

```
cpp_next::future<void> startConditionIO( const detail::TimePoint     & eventTimestamp,
                                         const ConditionSlotIteration & targetSlot ) final override;
```

ConditionAlg

- **After condition IO, post-processing is usually needed**
  - "Derived" conditions, such as alignments
- **For such tasks, an Alg-like abstraction makes sense**
  - Need a condition-aware variant: doesn't run for every event!
  - How much scheduling infrastructure should be shared?
- **For reasons outlined before, we think IO Algs are a mistake**
  - Support is feasible, probably better to drop them

Low-level ConditionAlg Interface

- **Algs register to the Scheduler during initialization**

```
ConditionAlgBase( ConditionSvc        & conditionService,
                  detail::IScheduler & scheduler );
```

- **They are implemented using handles**

```
template< typename T >
const ConditionReadHandle<T> & registerInput( const detail::ConditionID & inputID );
```

- **They compute conditions on Scheduler request**

```
#ifdef ALLOW_IO_IN_ALGORITHMS
   // Register a condition output (an algorithm may have raw outputs if it is allowed to carry out I
   template< typename T >
   const ConditionWriteHandle<T> & registerOutput( const detail::ConditionID & outputID,
                                                   const ConditionKind         outputKind );
#else
   // Register a derived condition output (only option if IO is not allowed)
   template< typename T >
   const ConditionWriteHandle<T> & registerOutput( const detail::ConditionID & outputID );
#endif

   virtual void execute( const ConditionSlot     & slot
#ifdef ALLOW_IO_IN_ALGORITHMS
                       , const detail::TimePoint & eventTimestamp
#endif
                       ) const = 0;
```

Requirement: Ease of Use
Functional ConditionAlgs

- **Implementing a ConditionAlg requires some boilerplate**
  - Register inputs and outputs during initialization
  - Read input conditions on execute()
  - Compute IoV of output (~ intersection of input IoVs)
  - Write output conditions down

- **Like in event processing, we can automate this work**

- **Prototype features Transformer + MultiTransformer demo**

ConditionTransformer

- **Base class template follows Transformer's conventions**

```
template< typename Result,
          typename... Args >
class ConditionTransformer< Result(Args...) > : public ConditionAlgBase
```

- **Constructor receives inputs/output identifiers**

```
using ArgsIDs = std::array< ConditionIDRef, sizeof...(Args) >;
ConditionTransformer(       ConditionSvc          &  conditionService,
                            detail::IScheduler    &  scheduler,
                      const detail::ConditionID   &  resultID,
                            ArgsIDs               && argsIDs );
```

- **User only needs to implement condition derivation functor**

```
virtual Result operator()( const Args & ... args ) const = 0;
```

- **Caveat: *Only suitable for condition derivation***
  - Design assumptions break down for IO

More Performance Figures

- **Prototype performance**
  - Scheduling an event with full condition reuse: 5.4 μs
  - Regenerating full condition dataset: (12.3 + 0.3 x N_{cond}) μs
  - ConditionTransformer overhead: (1.0 + 0.1 x N_{alg}) μs
  - Reading a condition: 10 ns

- **Benchmarking configuration**
  - GCC 6.2 / Linux 4.9 / Intel Xeon E5-1620 v3 @ 3.50GHz
  - N_{event} = 10000 and N_{cond} = 10000
  - Analysis through affine performance model

Overall Entry Pointer - ConditionSvc

- **At the end, we need a simple framework entry point**
- **Initialize it with a TransientConditionStorageSvc**

```
ConditionSvc( TransientConditionStorageSvc & transientStore );
```

- **Request asynchronous condition setup for each event**
  - Condition setup = Storage allocation + IO
  - Future-based interface provides flexibility
    - Non-blocking polling
    - Blocking wait for availability
    - Attach asynchronous continuation

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

- **Will also need experiment hook for timestamp extraction**

Requirement: Compatibility

- **Started from ATLAS' condition handling design**
  - Abstracted RAM storage away
  - Added support for condition garbage collection
  - Removed various implementation detail leaks
  - Used a more performance-oriented interface where sensible
    - ConditionHandle more tightly integrated with storage backend
    - IO concurrency is resource-based rather than request-based
- **Interface could probably wrap ATLAS infrastructure**
  - Biggest pain point would be IO algorithms
- **A common interface would allow a common CondDBSvc**

Conclusions

- **What's done**
  - Requirements analysis
  - High-level interface design
  - Full-featured prototype outside of Gaudi
  - Early performance analysis
- **What's next**
  - Refine interface design
  - Examine remaining experiment edge cases
  - Integrate into Gaudi & experiments
  - Improve documentation & tests (requires interface freeze)

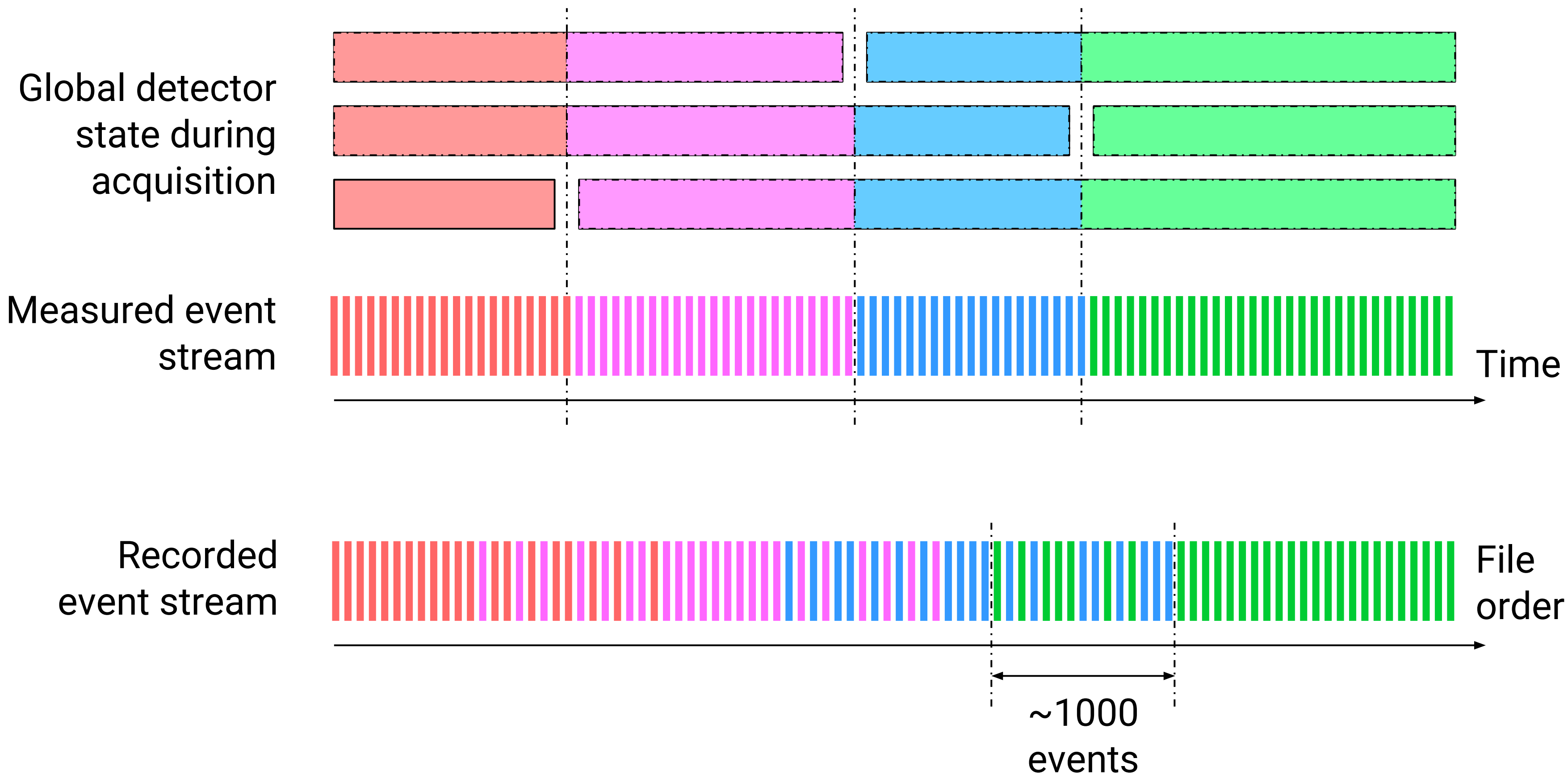
Questions? Comments?

Prototype code @ https://gitlab.cern.ch/hgraslan/conditions-prototype

Requirement: Scalability

- **In theory**
  - Condition readout is sync-free (zero mutexes/atomics)
  - Condition insertion locks a mutex briefly at the end
  - Slot allocation is mutex-protected, but has many fast paths
- **In practice**
  - Test scenario: Condition IO taking 24 ms, followed by "map" derivation taking 32 ms/condition. $N_{cond} = 16$, $N_{event} = 128$.
  - Derivation-only scenario: 8220 ms on a 4-core/8-thread CPU (7.97x speedup vs ideal sequential execution)
  - With IO: 8401 ms (8.16x sequential, due to latency hiding)

Effect of Out-of-Order Processing



Benedikt Hegner

A Bit of Terminology