Framework extensions:

Standardizing Gaudi 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.

Task 3.6 – Framework Extensions

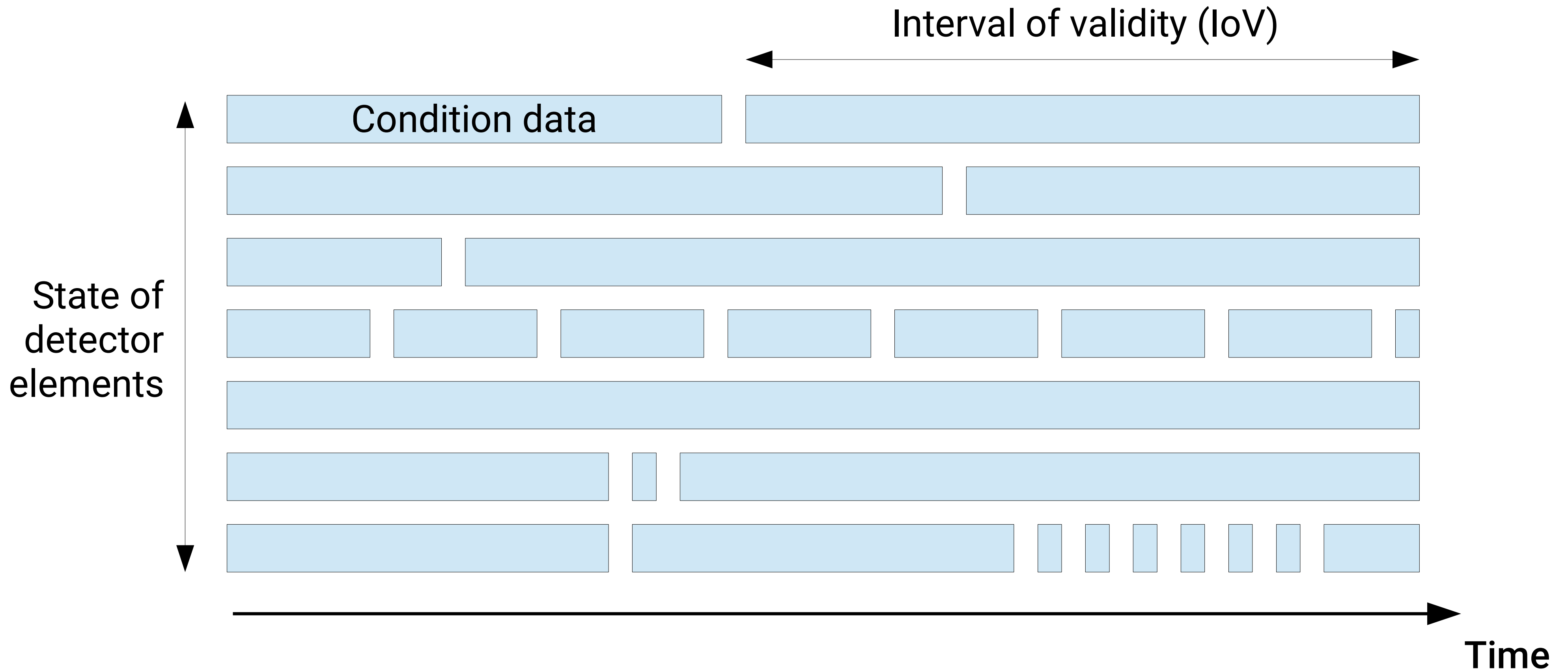
- Original task objectives (from AIDA2020 proposal):
  - Parallel algorithm scheduling for HEP frameworks
  - To be developed in a framework-independent way
  - Then integrated in Gaudi, Marlin, PandoraPFA
- However, parallel Gaudi algorithm scheduling work was completed before AIDA-2020 started
- Decided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

Some context

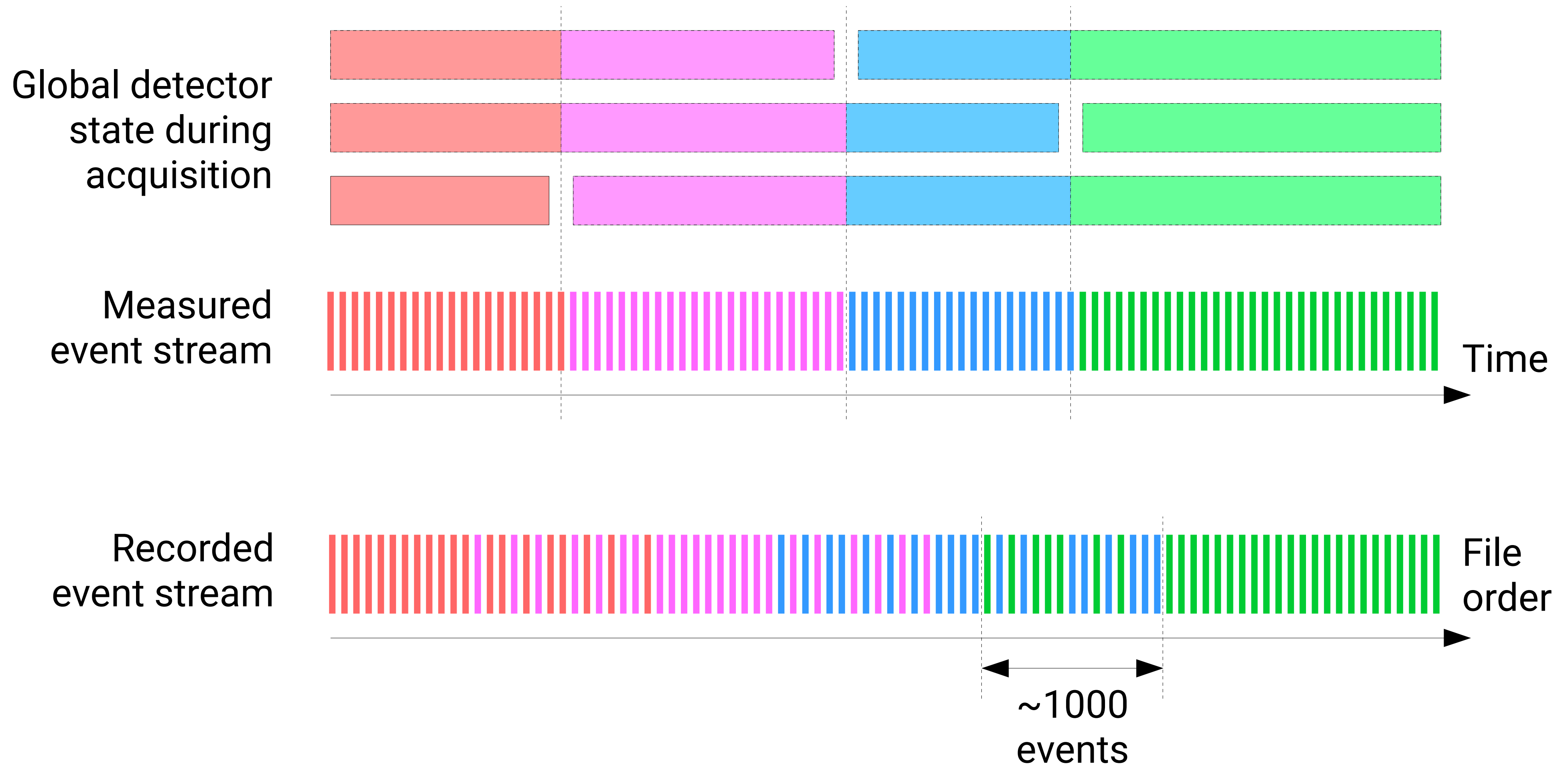
- Multi-processing is becoming too memory-intensive:
  - CPU core counts are exploding, but RAM prices vary slowly
  - Main memory bandwidth is becoming a major bottleneck
  - Importance of caches & scratchpads is increasing
- Result: many experiments must move to multi-threading

- Frameworks need to adapt to concurrent event processing
- One issue: time-dependent detector state, aka conditions
  - Detector state was historically modeled as a singleton...

A bit of terminology



Effect of out-of-order processing



Current Gaudi status

- Gaudi currently has no standard condition support
- Each experiment historically built its own mechanism
- These are all broken by concurrent event processing

- This is an opportunity to…
  - Move towards common abstractions & interfaces
  - Give new Gaudi users a pre-built solution
  - Share more code between historical users

- In this talk, I'll present a prototype implementation, whose integration into Gaudi is being actively discussed

Requirements

- Allow concurrent event processing (!)
- Support typical usage patterns efficiently
- Provide a common interface to diverse storage backends
- Keep RAM usage under control
- Enable efficient IO & computation patterns
- Maximize usability, scalability, resilience to errors
- Optimize compatibility with experiment-specific efforts

Condition usage 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!
- Event processing requirements vary between experiment
  - 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
- 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!

Storage backends

- In-RAM condition storage is a surprisingly popular problem
  - Traditional: Detector state singletons (e.g. Detector Store)
  - ATLAS: Detector state singleton with versioned contents
  - DDCond: Multiple “time slices” of detector state
  - Prototype: Conceptually similar to DDCond
- Some heterogeneity is likely to remain around
  - Unless ATLAS suddenly decides to move to DD4Hep
- Backend-specific framework code should be minimized

Constraining RAM usage

- Major goal of multi-threading: **keep RAM usage low!**
  - Do not let condition state grow indefinitely
  - Expose the ability of slice-based backends to set clear bounds on condition storage size
  - Condition storage can be abstracted as **ConditionSlots**
- Where framework interface plays a role:
  - Set a limit on the amount of ConditionSlots in flight
  - Track condition usage & perform garbage collection
  - Be high-level enough to allow storage optimizations

Transient storage interface

- Setup storage for N condition slots (0 = impl-defined):

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

- Allocate condition storage for an event (may be delayed):

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

- Declare condition dependencies and associated metadata:

```
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 );
```

- Access condition data through smart pointers (“handles”):

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

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

Efficient condition IO

- 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
    - Cons: Integration with algorithm scheduling is more difficult
- Prototype interface can accommodate both designs

Efficient computations

- Algorithm based parallel condition derivation is supported
- Ongoing debate regarding how it should be scheduled
  - Extend **event processing** infrastructure for conditions?
    - Pros: Code reuse, scheduler aware of full processing graph
    - Cons: Mixing frequently and rarely changing data likely to hurt event loop performance, worrying scheduler complexity, race conditions
  - Use **dedicated** infrastructure for some condition work?
    - Pros: Simpler building blocks, maximal event loop performance
    - Cons: Effort duplication, integration with event loop is more limited
- Prototype interface can accommodate both designs

Prototype performance

- Tested with batches of 10k events, 10k raw conditions, on an Intel Xeon E5-1620 v3 @ 3.50GHz (4 cores + HT)
- Current overhead of individual operations:
  - Writing/reading transient storage: 0.3 µs / **10 ns**
  - Scheduling an event w/ full condition reuse: 5.4 µs
  - Regenerating raw conditions: $(12.3 + 0.3 \times N_{cond})$ µs
  - Deriving conditions $(1.0 + 0.1 \times N_{alg} + 0.3 \times N_{cond,out})$ µs
- Multi-threaded scalability:
  - **Reads are sync-free**, other ops use fine-grained locking
  - Works concurrently (**7.97x** speedup + IO latency hiding)

Conclusions

- Prototype achieves its intended goals, and then some:
  - Simple storage abstractions, suitable for all Gaudi users
  - Event loop performance is sufficient for LHCb
  - Concurrent condition IO/derivation, as required by ATLAS
  - Passed independent design review from B. Hegner
- Integration in Gaudi has begun, waiting for community consensus on various matters:
  - Standard Gaudi abstraction for reentrant storage access
  - Choice of mechanism(s) for asynchronous IO
  - Distribution of condition handling responsibility

Questions? Comments?

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