Standardizing Gaudi conditions

Design, prototype and perspectives

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.

Previously...

- Gaudi did not provide condition support
- So each experiment wrote its own
- Multi-threaded event processing broke everything
- What should we do next?
  - Keep maintaining duplicate codebases?
  - Converge towards a common approach?

Requirements

- Support concurrent event processing
- Accommodate diverse storage backends
- Keep RAM usage in check
- Allow efficient condition IO & computations
- Easy to use, error-proof, and scalable
- Experiment-agnostic, but reasonably compatible

Requirements

- Support concurrent event processing
- Accommodate diverse storage backends
- Keep RAM usage in check
- Allow efficient condition IO & computations
- Easy to use, error-proof, and scalable
- Experiment-agnostic, but reasonably compatible

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 unlikely
- Framework interface should be backend-agnostic

Memory usage control

- Multithreaded Gaudi is mainly about **RAM usage**
  - Condition state should not grow indefinitely
  - Most storage backends can bound amount of detector states in flight, we should expose this
  - For consistency with EventSlots, I propose calling storage for a detector state a **ConditionSlot**
- Framework interface to conditions plays a key role here:
  - Allows backend to track condition usage
  - Enables storage optimizations (sharing, lazy GC...)

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:

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

Dataflow tracking

- We need to track some condition usage metadata
  - For the backend to manage condition data correctly
  - For the Gaudi Scheduler to know data dependencies
- Condition users also need a way to access conditions
- We can achieve both goals with a single interface:

```
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 );
```

Condition access

- Condition handles are a proxy to condition data
- Each condition user must request its handles separately
  - ...so 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

Performance?

- Condition handle prototype is reasonably fast[1]:
  - Writing a condition takes 0.3 μs
  - Reading a condition takes 10 ns
  - Algorithm independent of N_{cond}, tested for 10K conditions
- Withstands comparison to 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)

[1] More performance numbers available on request

Requirements

- Support concurrent event processing
- Accommodate diverse storage backends
- Keep RAM usage in check
- Allow efficient condition IO & computations
- Easy to use, error-proof, and scalable
- Experiment-agnostic, but reasonably compatible

IO in Gaudi

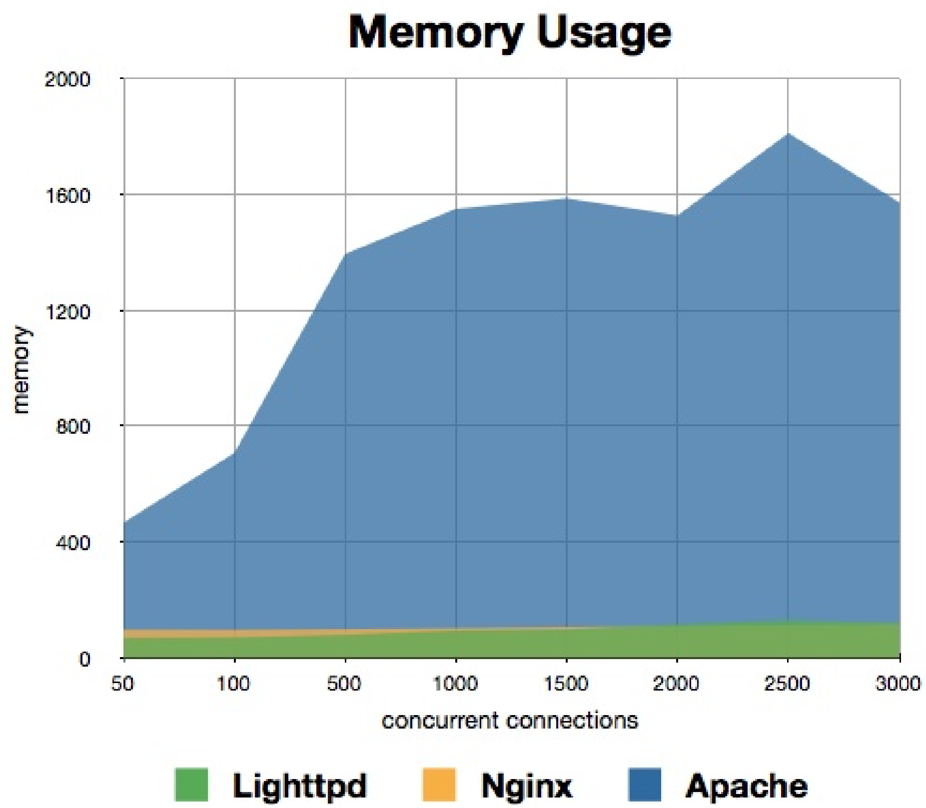
- TBB-based Alg scheduling model well-suited for compute:
  - Keep one worker thread per CPU core/thread
  - Feed worker threads with tasks whose inputs are ready
  - Do not let any task use a blocking construct, ever
- What should be done about (blocking) IO workloads?
  - Extend the Alg abstraction to support these use cases?
  - Use another abstraction for IO purposes?

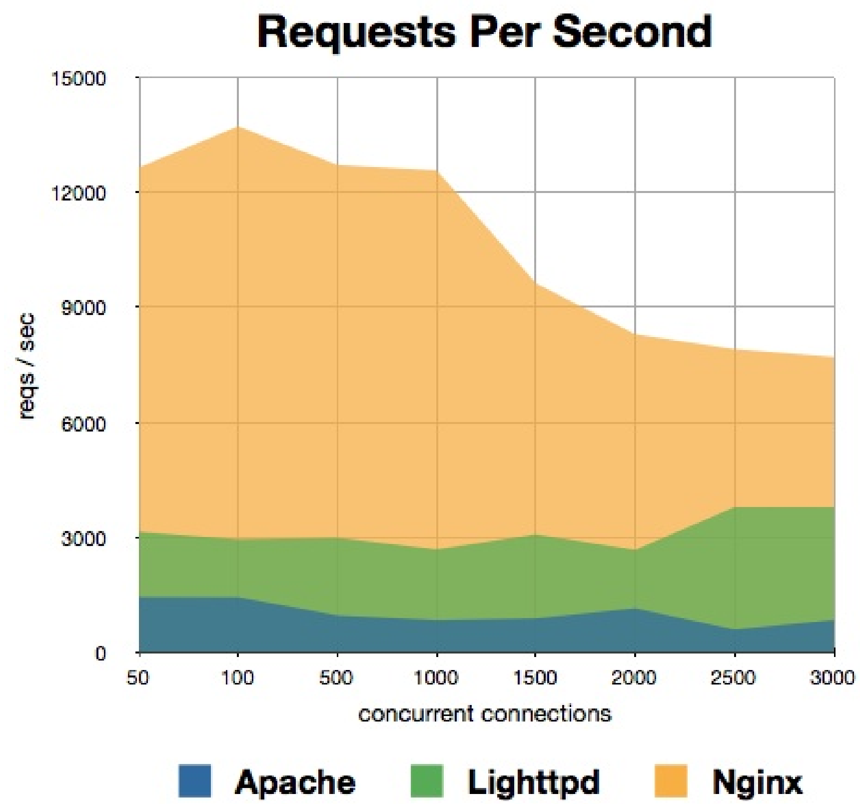
Option 1: IO Algs

- Reminder: Compute threads should **never** block
  - Whenever a compute Alg blocks, a CPU goes idle
  - If all CPUs go idle, it can lead to a deadlock
- Use another thread pool for blocking tasks?
  - User picks *large enough* amount of IO threads
  - User tags offending Algs as IO-bound
  - User writes IO Algs with sensible IO patterns in mind
  - Scheduler runs IO-tagged Algs on the IO thread pool

IO threads require care

Feedback from Apache[2]: 1 thread/IO request is a **bad** idea





[2] https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison

Managing IO threads

- Most IO resources are sequential and thread-unsafe
  - File descriptors (and abstractions thereof)
  - Network connexions
- There's no benefit in accessing these concurrently
  - At best it serializes, at worst it blows up
- Proper IO thread management is resource-dependent
  - 1 thread/resource is a good starting point
  - But remember that software/hardware mapping is $N \longrightarrow M$
- Should the Gaudi scheduler really care about all this?

Option 2: IO Services

- Serial entities are normally modeled as **Services**
- IO resources should be no exception!
- Benefits of modeling IO resources as services:
  - Can drop fragile “IO thread count” tuning parameter
    - If ever needed, should probably be resource-specific
  - Specialized code allows resource-specific optimizations
    - Grouping requests in batches, looking ahead…
  - Gaudi Scheduler is kept simple, focused on compute
- For optimal efficiency, IO services should be asynchronous
  - Allows processing unrelated IO in parallel

IO 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 );

#ifdef ALLOW_IO_IN_ALGORITHMS
   // Register a condition output (an algorithm may have raw outputs if it is allowed to carry out IO)
   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
```

- They compute conditions on Scheduler request:

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

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

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:

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

  - Condition setup = Storage allocation + IO
  - Future-based interface provides flexibility
    - Non-blocking polling
    - Blocking wait for availability
    - Attach asynchronous continuation
- Will also need experiment hook for timestamp extraction

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

Usability

- The proposed interface was designed for ease of use
  - Correct code is easy, incorrect code is hard
- Some examples of these principles at work:
  - Automatic ConditionSlot liberation
  - Type-safe ConditionHandles
  - Zero/Multiple condition writers is a run-time error
  - Zero readers is also a run-time error (more debatable)
  - IOSvc/ConditionAlg bases make it easy to register handles
  - Functional ConditionAlgs make it even easier

More performance

- Prototype performance as of 2017-02-01:
  - Scheduling an event with full condition reuse: 5.4 µs
  - Regenerating full condition dataset: $(12.3 + 0.3 \times N_{cond})$ µs
  - ConditionTransformer overhead: $(1.0 + 0.1 \times N_{alg})$ 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

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)

Compatibility tradeoffs

- Started from ATLAS' condition handling design
  - Abstracted RAM storage away (needed for DD4Hep)
  - 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!