Future frameworks: ATLAS perspective

Scott Snyder

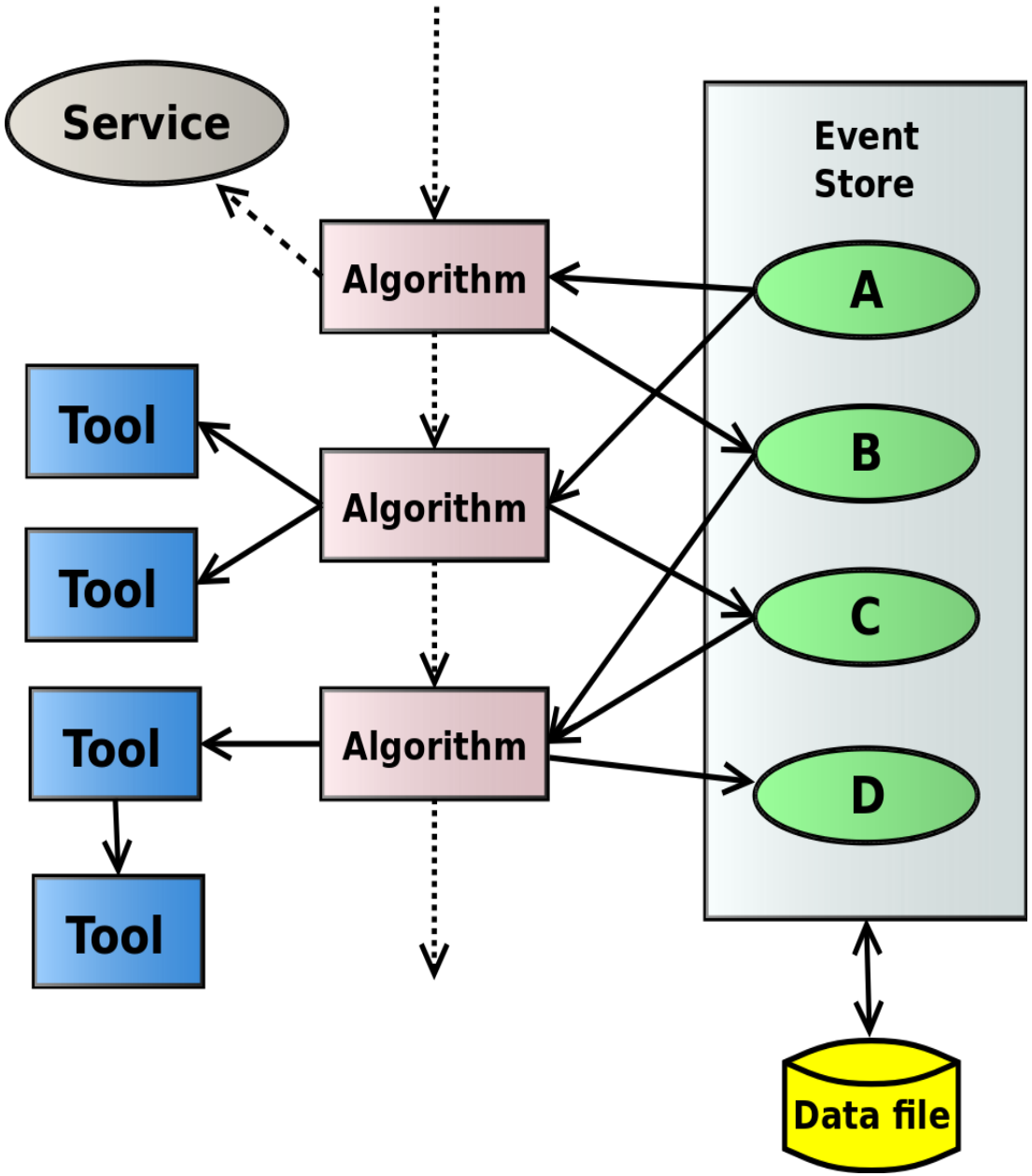
BNL

Mar 10, 2020

Framing frameworks

- Some thoughts about framework design.
- Mostly from the perspective of ATLAS.
- Brief tour of the ATLAS framework.
  - With some emphasis on what worked well, and what we might change if we started over.
- Considering "event" processing framework.
  - Event: Independent, relatively small chunk of data.
- ATLAS offline software (Athena) based on the GAUDI framework.
  - Originated from LHCb in the 90's.
  - Also used by numerous other experiments.
  - Shared development stagnated during Run 1. Has since been reinvigorated, but some effort was required to reconcile diverging developments.

Common framework design



Almost all HEP frameworks use:

"Whiteboard" pattern

Sequence of Algorithms communicating via event store.

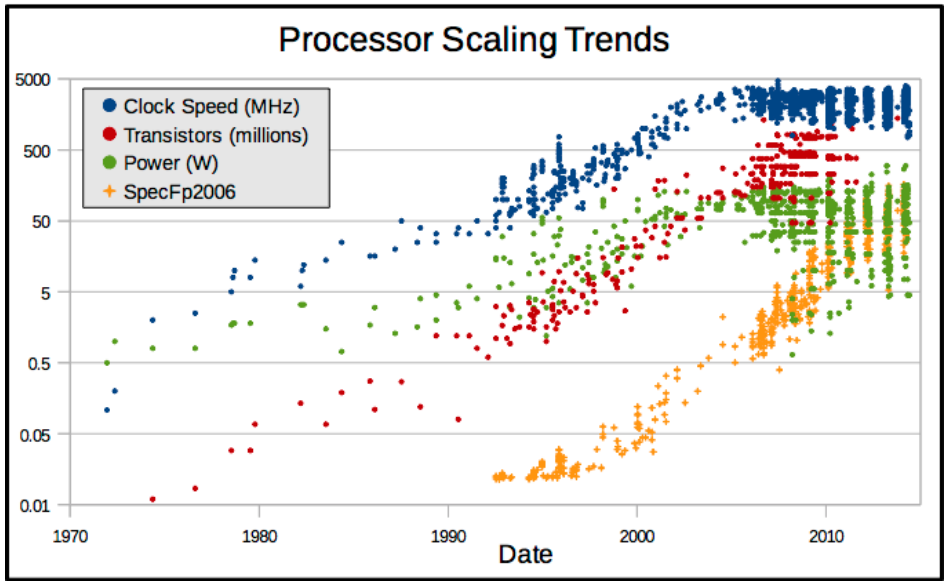
For good reason!

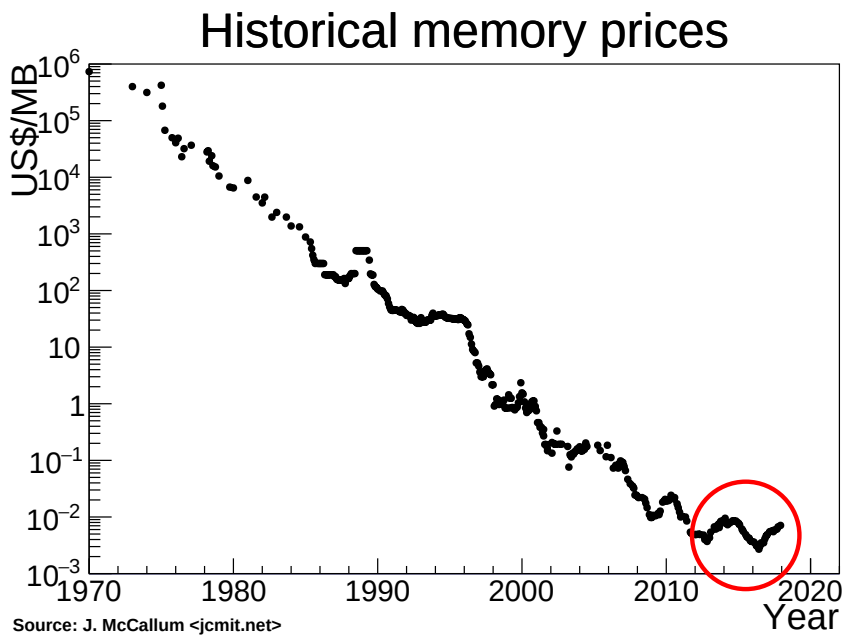
Through LHC run 1, usual practice was a fixed Algorithm sequence, set in job configuration.

Multi-processing/threading

Evolution during LHC Run 2

- Clock speeds have plateaued. Processors are getting wider vector units and more cores.
- Memory prices not been decreasing much recently.
- Expect ratio of memory to cores to decrease.
- Can't continue to keep cores occupied simply by running multiple jobs on one machine. Need to reduce memory required per core.

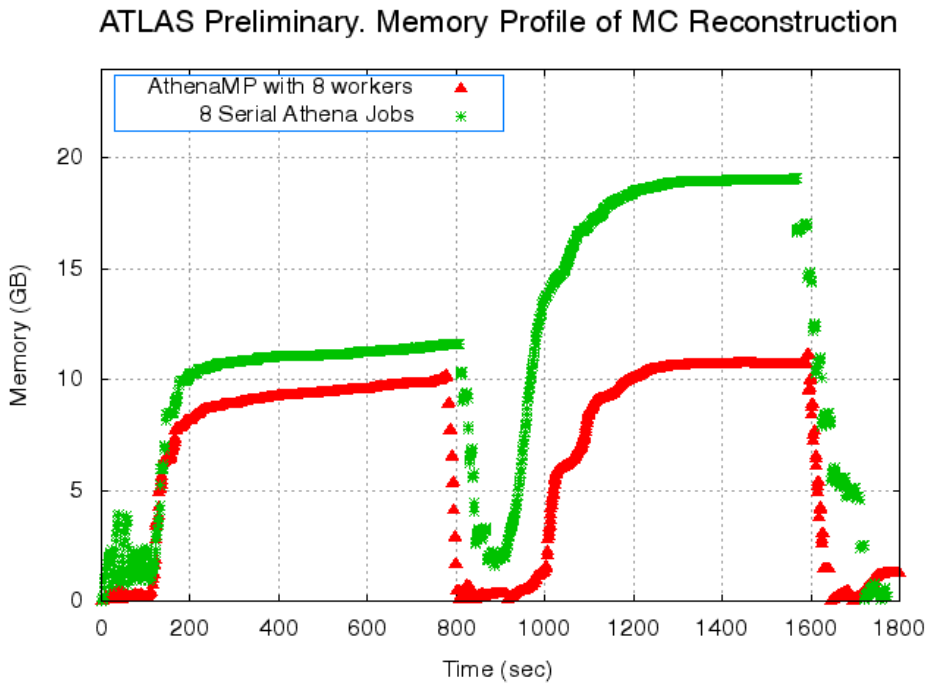




AthenaMP

For Run 2, ATLAS reduced memory requirements via *multiprocessing*.

- A job forks subprocesses to process events in parallel. Memory is shared automatically via copy-on-write.



Conceptually simple. Prototyped in an afternoon, but took a long time to make ready for production.

Also employed by other experiments (e.g., Belle II).

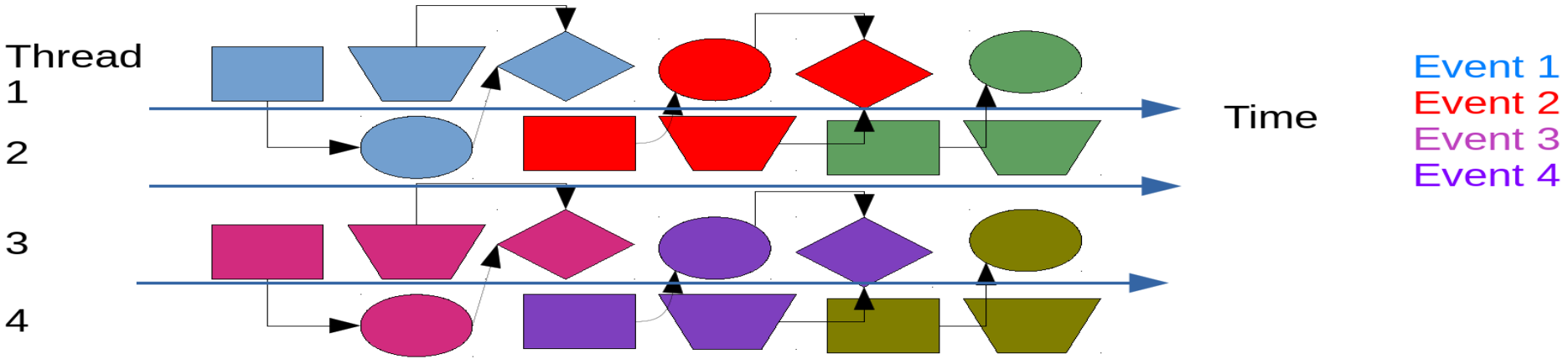
Multithreading

Can have inter-event parallelism, intra-event parallelism, or both.

Full multithreading implemented by CMS (Run 2) and ATLAS (Run 3). Others have implemented more restricted approaches.

Natural approach is to couple the whiteboard pattern with a task-based scheduler such as provided by TBB.

Algorithms declare their inputs and outputs. A scheduler chooses algorithms to run based on what data items are available. Allow multiple events in flight.



A major effort to convert an existing code base. But have generally seen very good memory scaling behavior.

Designing for multithreading

Planning from the beginning helps here!

Use `const` as much as possible.

- Don't modify data objects after creation.
- Avoid caches / mutable data in classes.
  - Think about how to handle things like statistics and histograms.
- C++ convention: `const` methods can be called simultaneously from multiple threads.
- Avoid `mutable` and `const_cast` — they may require explicit locking.

Avoid static state, even `thread_local`.

- Explicit is better than implicit.

ATLAS and CMS use custom static analysis tools to help identify problematic code.

Example: Algorithm interface

Old:

```
virtual StatusCode execute();
virtual StatusCode beginRun();
virtual StatusCode endRun();
```

- Assumes `begin/endRun` are well-ordered with respect to event processing.
- `execute()` is non-const.
- Current event is not passed to `execute()` — have to find it using (ultimately) a global static.

New:

```
virtual StatusCode execute (const EventContext&) const;
```

But still have many internal interfaces that don't pass the `EventContext`. Need to resort to reading a (thread-local) static global in such cases.

Functional interface

- Usual interface leaves it to the Algorithm to retrieve/record data from/to the event store ... which is largely boilerplate.
- Gaudi also supports a 'functional' interface for algorithms that factors this out.

```
class FunctionalAlg
  : public Functional::Transformer
      <OutputData (const Input1&,
                   const Input2&)>
{
   ...
   OutputData operator() (const Input1& in1,
                          const Input2& in2) const;
}
```

- Used by LHCb.
- Not seriously considered for ATLAS in Run 3 mainly due to time constraints, but may adopt some elements of this in the future.

Conditions

Calibrations, etc. depending on event number or time.

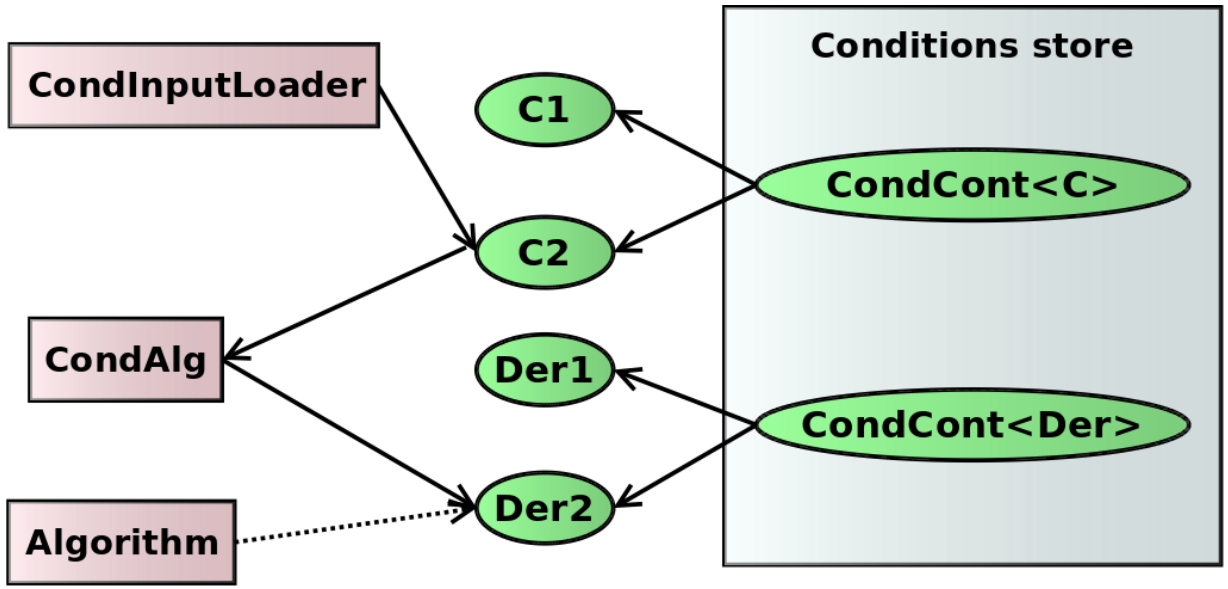
Different events may use different conditions versions.

Conditions store holds potentially multiple versions of conditions objects.

CondInputLoader algorithm loads needed conditions for each event.

To apply a transformation to conditions data, use a 'conditions algorithm' acting on data in the conditions store.

Algorithms access conditions data via handles.




```
ReadCondHandle<Der> h (key, ctx);
```

Scheduler knows about dependencies and schedules them accordingly.

Implicit dependencies

Have some implicit dependencies between algorithms that casuse difficulties with scheduling.

Example: Input data objects are not read until they are accessed for the first time. Done via a '`Converter`' object.

Some `Converter` objects have dependencies on, for example, conditions data. But the scheduler does not know about these dependencies.

Resolved by introducing an `Algorithm` that produces the data object, which can be scheduled as usual. Eventually, would like to migrate away from converters to using algorithms for all input.

Gaudi component model

Gaudi components:

- Inspired by Microsoft COM.
- Dynamically-loaded class.
- Identified by class / instance name,
- Ideally used via an abstract base class.
- Each component has a set of typed properties set during job configuration.
- Component types:
  - Algorithm: Event processor.
  - Helper class, owned by an Algorithm or another tool.
  - Global (singleton) service. Examples: error logging, random number management. Should be explicitly thread-safe.

Job configuration

- Job configuration can be done with a domain-specific language (may be based on XML) or with a general-purpose programming language (often Python).
- Gaudi may be configured with either a declarative, text-based format or with Python.
- Using a general-purpose programming language is attractive, but need to provide more structure on top of it.
- ATLAS made several attempts at this, none of which was fully taken to completion.
  - As a result, different parts of the job are configured in different ways.
- Now trying again for Run 3 with a new design.
  - Based on configuring components independently and deduplicating when merging.

Run 3 job configuration: simple example

```
def MyAlgoCfg(configFlags):
    result=ComponentAccumulator()
    isMC=configFlags.Input.isMC

    # Set up Calorimeter geometry.
    from CaloGeoAlgs.CaloGMConfig import CaloGMCfg
    result.merge(CaloGMCfg(configFlags))

    from MyComponents.MyToolCfg import MyToolCfg
    tool = MyToolCfg(configFlags)

    from MyComponents.MyComponentsConf import MyAlgo
    myalg = MyAlgorithm('myalg', isData = not isMC,
                        tool = tool.popPrivateTools())
    result.addEventAlgo(myalg)
    result.merge(tool)
    return result
```

Event data model

- Experiments tended to build complex data models with complicated inheritance schemes.
- Led to poor performance and complications with persistency.
- Trend now is to represent a collection of objects as a set of arrays of (relatively) simple types ('structure of arrays' (SOA)).
  - Removes much of the overhead from data access.
  - Better locality of reference / caching behavior.
  - Simpler persistency.
  - Possibility of performing vectorized operations.
  - Reducing reformatting needed for GPUs / accelerators.
  - Efficient interop. with numpy-based / ML tools, plus ROOT.
- Challenges:
  - Users should not have to deal with a SOA representation directly. Need to have an object-based interface on top.
  - How to represent variable-sized arrays?
  - References between objects?
  - 4-vector / matrix / other compound types?

ATLAS desires for Run 2

Several Run 1 types supported adding extra named pieces of data — "decorations" — to elements of containers.

- Originally done to be able to separate pieces of the structure for I/O.
- Several different, incompatible implementations.
- Can we unify this?

Data stored as, essentially, "array of structures."

- Poor locality of reference.
- "Structure of arrays" might be better.
- Can we still make it look like a collection of structures?

Make data easily and efficiently readable from ROOT.

- Avoid copies.
- Partial object reading / writing.
- User extensibility for analysis.

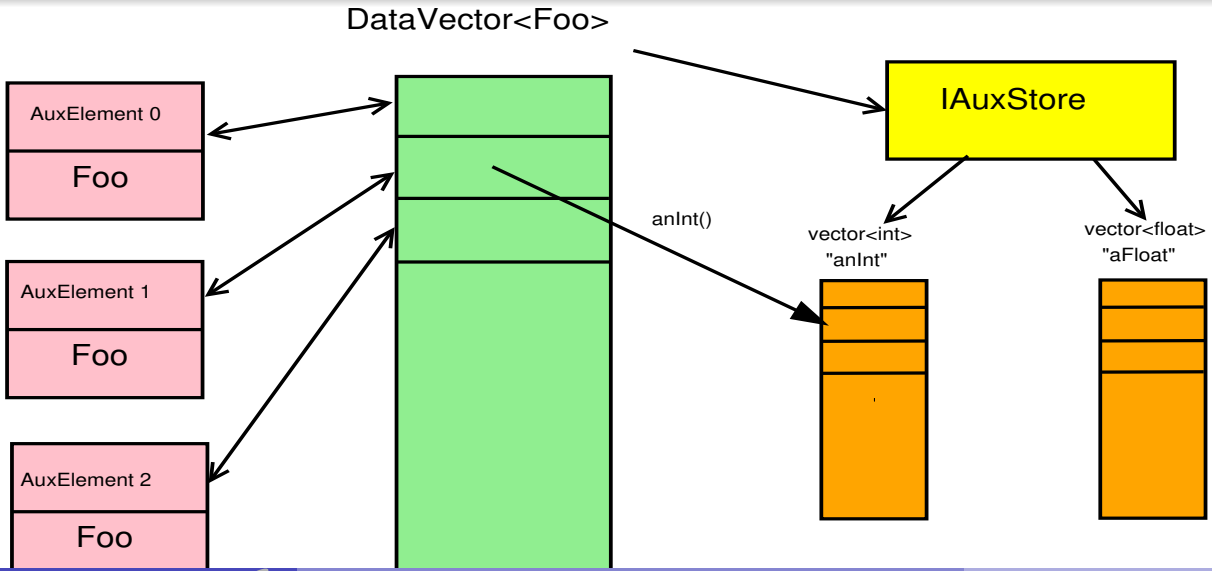
ATLAS Run 2 data model (xAOD)

Long time basis of ATLAS EDM: `DataVector<T>` acts like `std::vector<T*>`.

Wanted to preserve the interfaces people were familiar with, but have data stored as contiguous, segregated arrays.

`T` instances don't hold their data.

Store object holds data and accessed via an abstract interface, so multiple implementations are possible. xAOD only requires that each variable be in a contiguous array.



ATLAS Run 2 data model (xAOD)

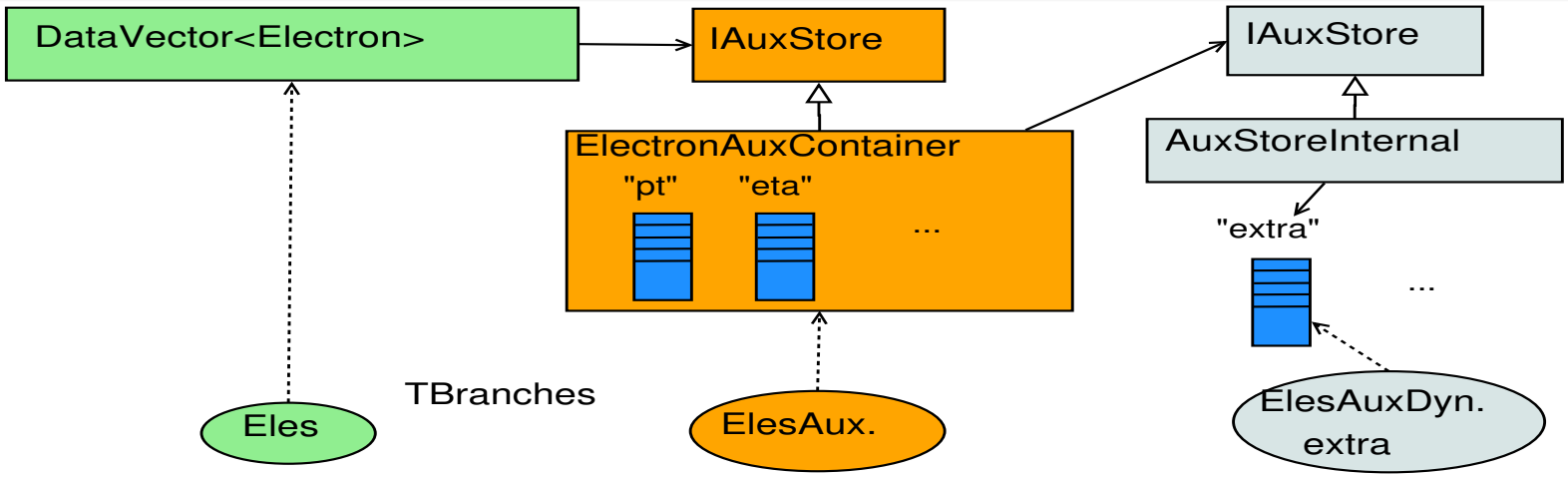
Most xAOD data kept in a 'static' store object.

- Has a bunch of `std::vector` members.
- Given to ROOT to save.

Static store can reference an additional 'dynamic' store.

- Contains extra variables.
- Storage managed dynamically, but also as `std::vector` instances.
- I/O system treats each dynamic variable individually.

Allows for decorations / shallow copies.



ATLAS EDM future improvements

Not all types have been converted to xAOD format; generally only those used for analysis. Many of the remaining types should probably be migrated eventually.

xAOD-style layout may also be useful for conditions data / detector description.

Preserving the existing `vector<T*>` interface aided migration, but prevents vectorization. More vectorizaton-friendly interfaces are available, but are more awkward and have not generally been used. More should be done to make this easier to use. Such interfaces would also be more friendly for GPU code.

Complex types used as variables, such as `std::vector`, are problematic.

Need to take more control of memory allocation.

Summary

Design for multithreading from the beginning.

- Task-based paridigm has worked well for HEP applications.
- Make dependencies explicit.
- Don't modify event data once it's been created.
- Declare `const` as much as possible.
- Avoid `const_cast` and `mutable`.
- Avoid static data.
- Minimize use of `thread_local`.

Even if you ultimately don't use threading, designing for threading will lead to more robust code.

Avoid complicated data models

- Consider using a 'SOA' representation.
- But with an object interface on top.
- Decide how to handle variable-length arrays and references between objects.
- Build in control of memory allocation from the start.