Gaudi/Athena Multithreaded Scheduling

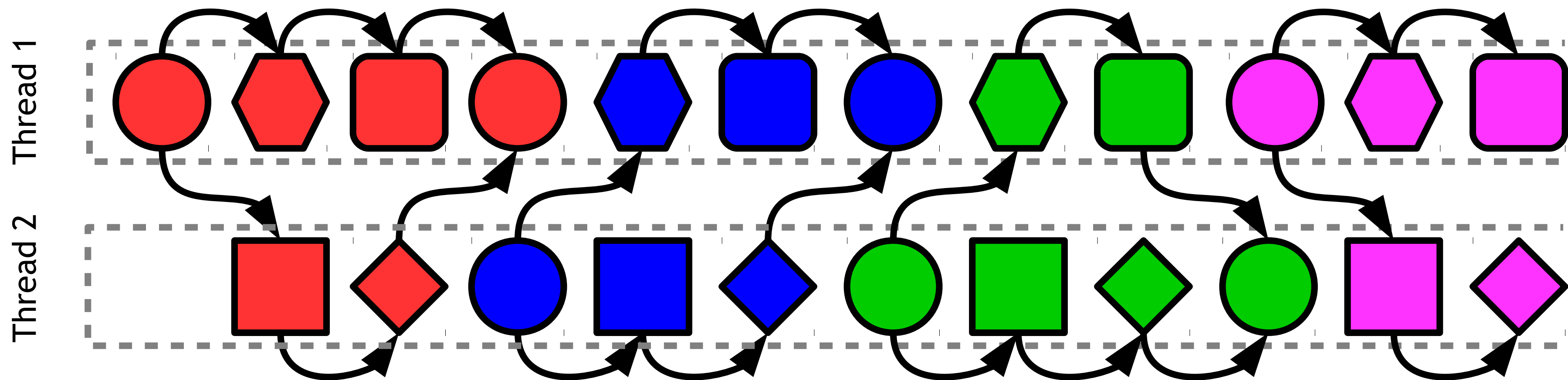
B. Wynne

01/07/20

Introduction

The ATLAS software framework Athena(MT) is built on top of the multi-experiment framework Gaudi

In particular, the (ongoing) migration to multithreaded operation relies on the Gaudi AvalancheScheduler
- Single-threaded execution expected within algorithms
- Parallel execution of multiple independent algorithms
- Multiple events processed simultaneously, sharing thread pool and algorithm instances
- The same premise as the GaudiHive prototype, but since extended with a number of features requested by ATLAS



Dependencies between algorithms determine what the scheduler will run next
- The PrecedenceSvc evaluates our dependency graph

AvalancheScheduler

The PrecedenceSvc uses the states of algorithms in an event to determine which algorithms should be executed next

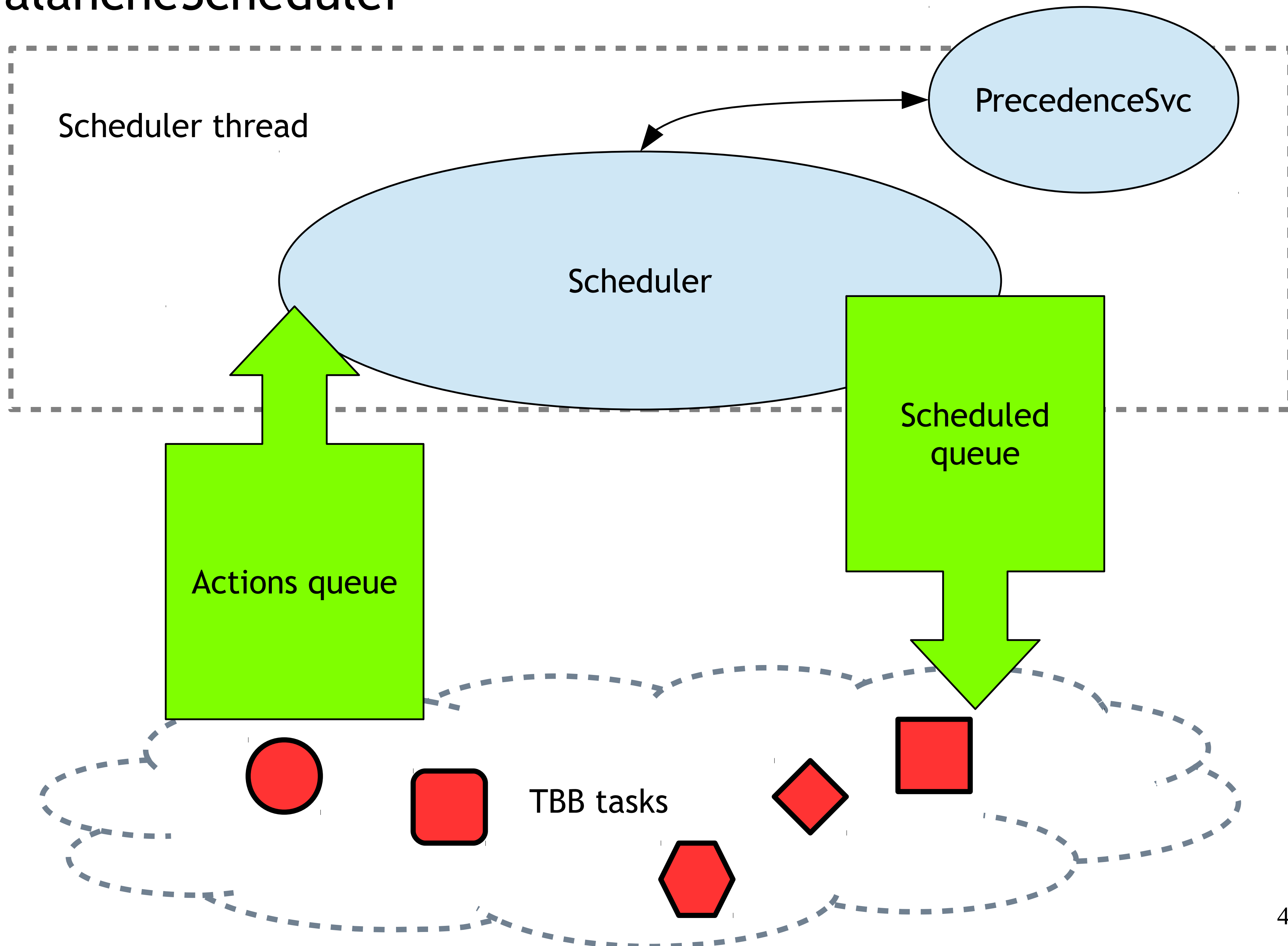
The AvalancheScheduler executes algorithms, and stores their states.
It runs in a dedicated thread, using two thread-safe queues to communicate with the rest of the framework

Anything that will modify the state stored by the scheduler is performed by pushing a lambda function to the actions queue
- Starting and ending an event
- Handling the result of algorithm execution
- HLT regional reconstruction in sub-events/views

Once the actions queue is emptied, the scheduler will query the PrecedenceSvc for the next set of algorithms to execute given the state of an event

All algorithms that can be executed are added to the scheduled queue
- The queue is ordered by algorithm priority
- Each algorithm has a corresponding event context to run in
- TBB tasks pop the front of the queue, execute the algorithm, and push the result to the scheduler actions queue

AvalancheScheduler



Data dependencies

Most dependencies between algorithms are data dependencies: the output of Algorithm A becomes the input to Algorithm B

Algorithms use smart pointers – DataHandles – to record and retrieve data from the EventStore
- DataHandles are digested at configuration time to build the dependency graph
- Multiple different stores can all be accessed with this mechanism

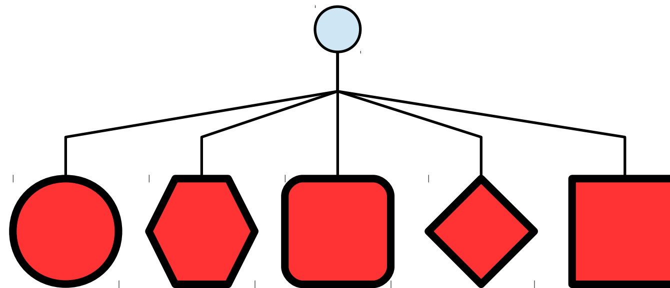
The scheduler does not inspect the contents of the EventStore, it only stores the execution states of algorithms
- If an algorithm has run for a particular event, assume that the outputs of that algorithm are now available
- A data object with multiple possible creators is available if any one of them has finished execution

It is possible to access data without declaring a dependency
- Some algorithms test to see if a data object is available, and adjust their behaviour accordingly
- Legacy access methods still in the process of migration

Control flow

If data dependencies determine which algorithms **can** run, control flow allows us to impose extra rules on what **may** run

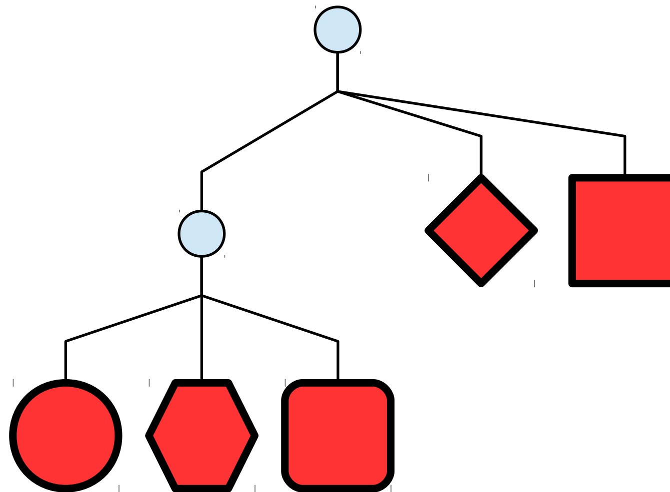
The basic units of control flow are sequences
- Sequences have one or more algorithms as children



Control flow

If data dependencies determine which algorithms **can** run, control flow allows us to impose extra rules on what **may** run

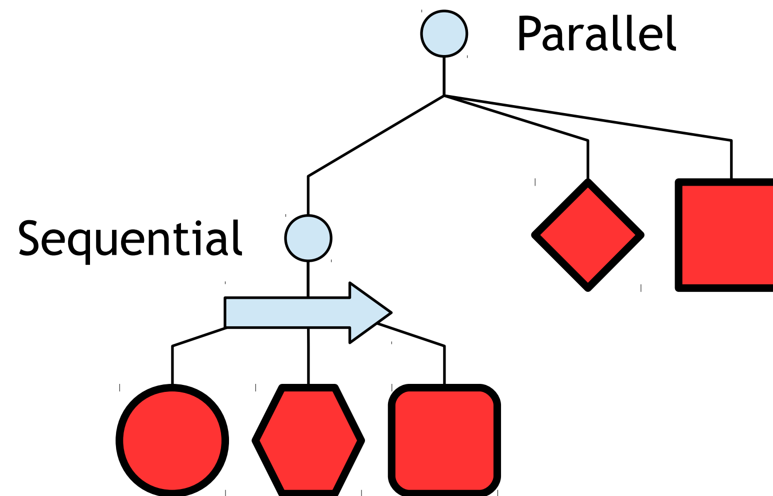
The basic units of control flow are sequences
- Sequences have one or more algorithms as children
- Sequences can be nested, with a single “top” sequence for a whole event



Control flow

If data dependencies determine which algorithms **can** run, control flow allows us to impose extra rules on what **may** run

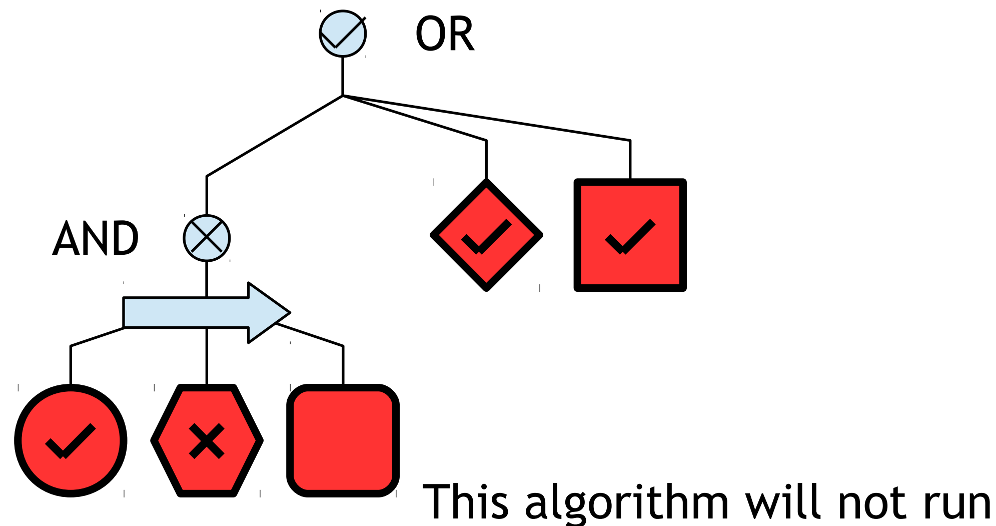
The basic units of control flow are sequences
- Sequences have one or more algorithms as children
- Sequences can be nested, with a single “top” sequence for a whole event
- Children can be run sequentially or in parallel



Control flow

If data dependencies determine which algorithms **can** run, control flow allows us to impose extra rules on what **may** run

The basic units of control flow are sequences
 - Sequences have one or more algorithms as children
 - Sequences can be nested, with a single “top” sequence for a whole event
 - Children can be run sequentially or in parallel
 - Algorithms and sequences return PASS or FAIL decisions, with parent sequences returning logical operations of their childrens’ decisions

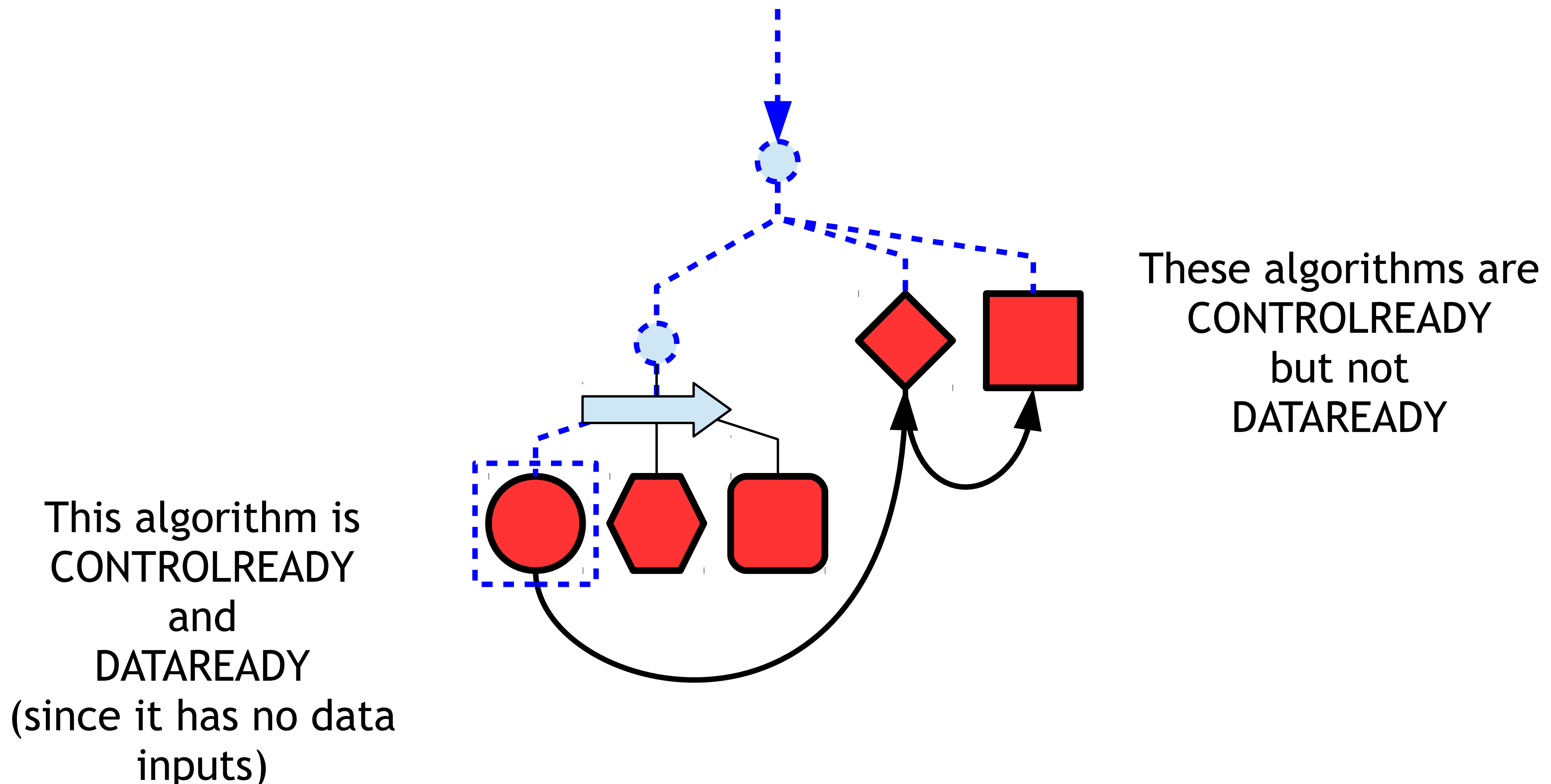


PrecedenceSvc

On top of all of the control flow rules, we must also apply data dependencies

The combination of the two is called the PrecedenceRulesGraph, which is evaluated by the PrecedenceSvc

At the start of an event, the PrecedenceSvc traverses the graph starting at the root node, to identify all algorithms that can be scheduled



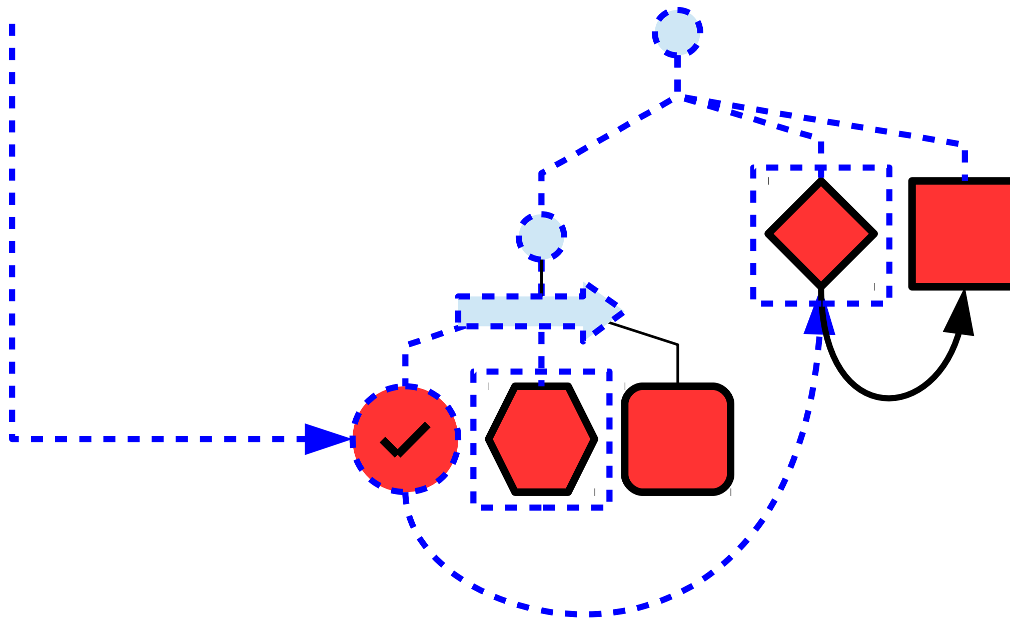
PrecedenceSvc

On top of all of the control flow rules, we must also apply data dependencies

The combination of the two is called the PrecedenceRulesGraph, which is evaluated by the PrecedenceSvc

At the start of an event, the PrecedenceSvc traverses the graph starting at the root node, to identify all algorithms that can be scheduled

All subsequent updates start at the algorithm that last executed, to avoid re-evaluating the whole tree

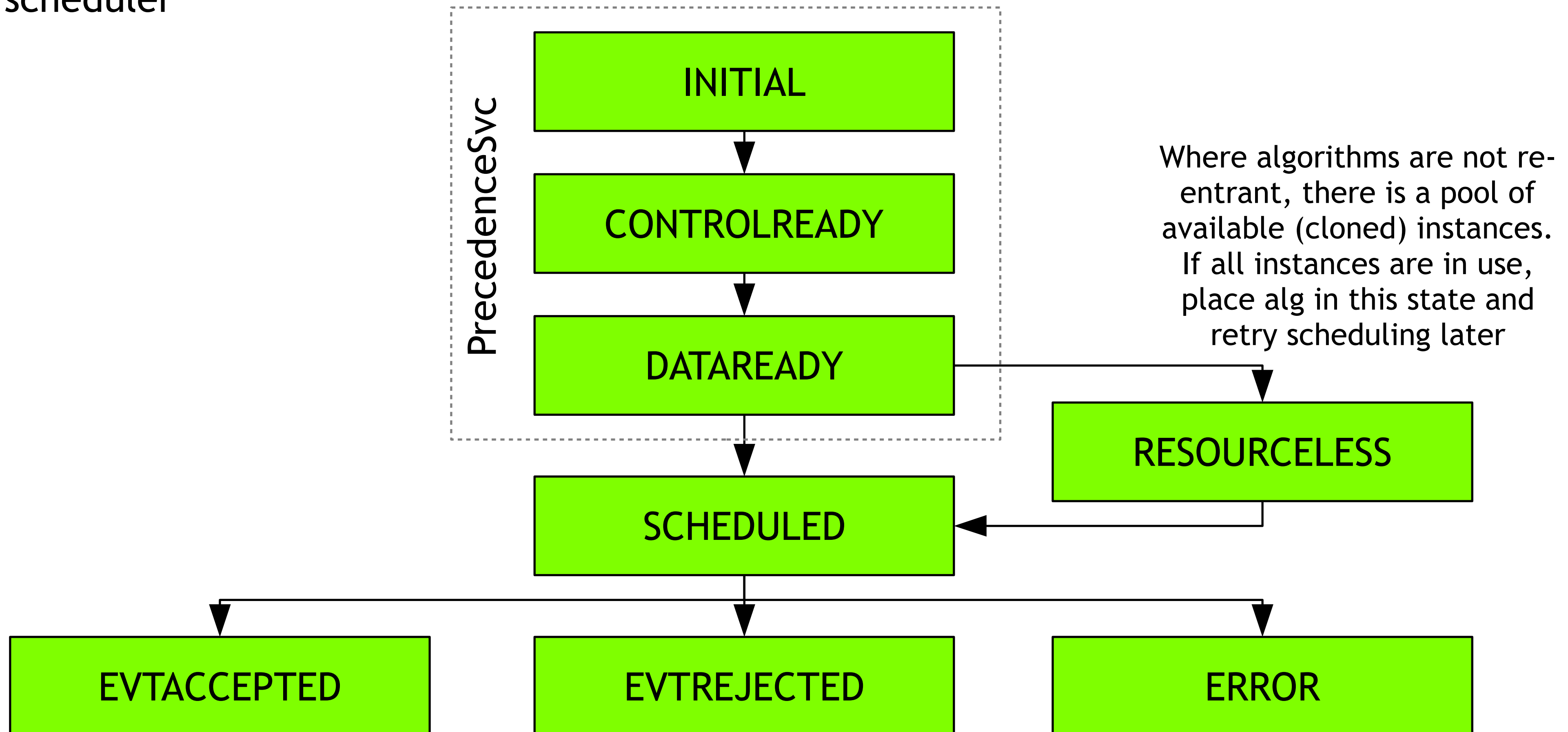


Graph-walking visitors perform the updates, e.g. DataReadyPromoter follows data dependencies, and Supervisor follows control flow

Algorithm states

Each algorithm in each concurrent event has a state value stored by the AvalancheScheduler

Algorithm state transitions are performed by the PrecedenceSvc, or by the scheduler

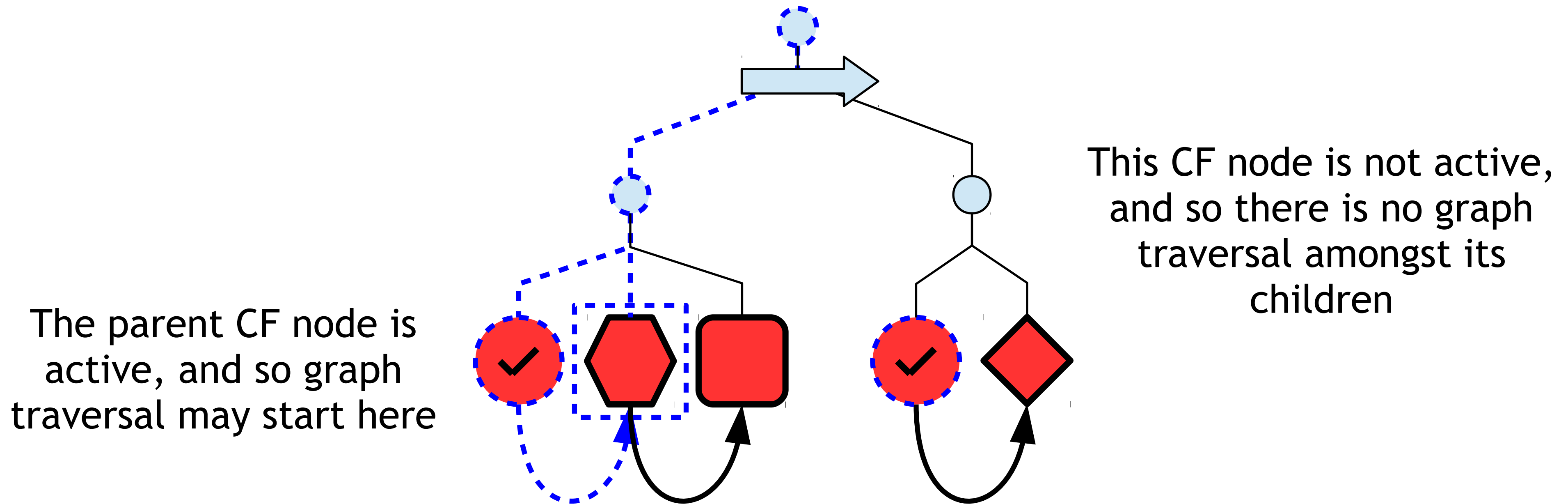


Results of execution are pass or fail decisions, or an error

Subtleties – duplicate algorithm CF

There might be several different reasons to run a specific algorithm (e.g. multiple muon triggers with different thresholds)

When traversing the precedence graph after that algorithm has run, we must test to ensure we don’t enter a CF node that should not be available

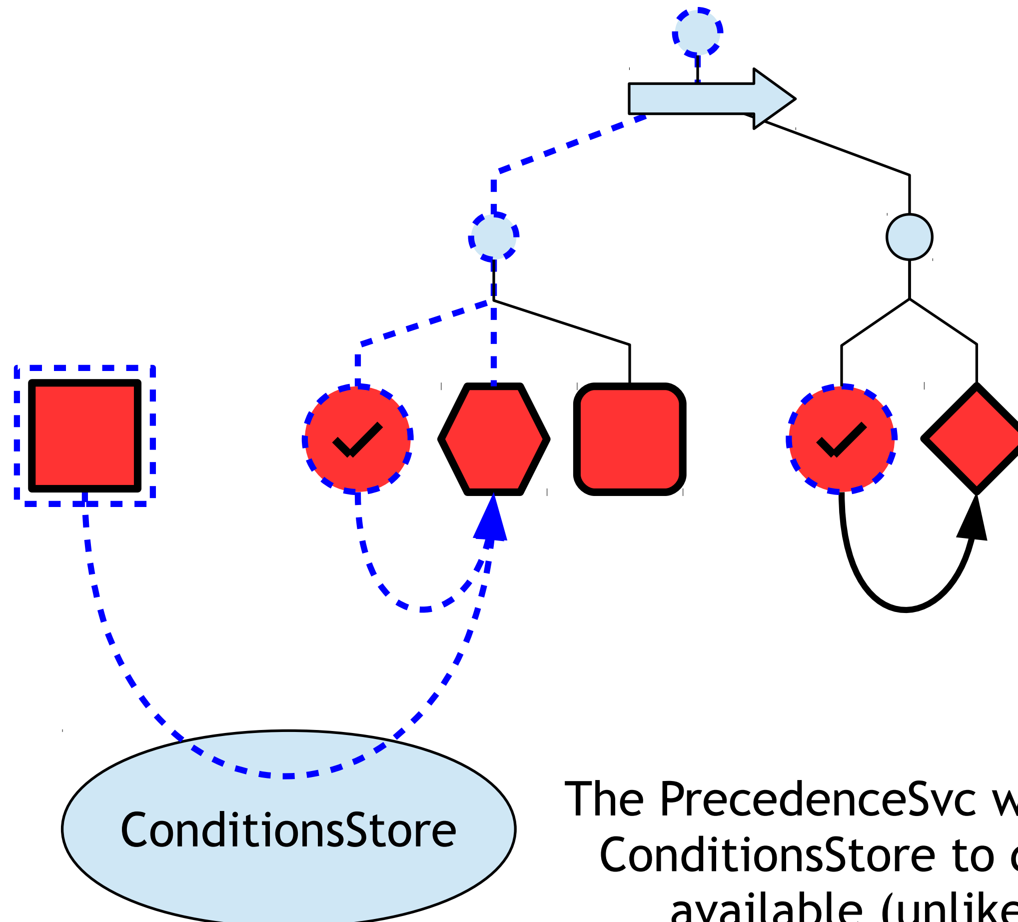


A graph visitor is sent back towards the root node from the point where we enter the graph, to ensure that there is a path via active nodes

Subtleties – conditions algs

Conditions data has a lifespan longer than a single event, and is expected always to be available if needed

Algorithms that produce conditions data are executed on-demand if the data does not already exist for a particular interval of validity
- They are not attached to the CF graph, since they should not run every event

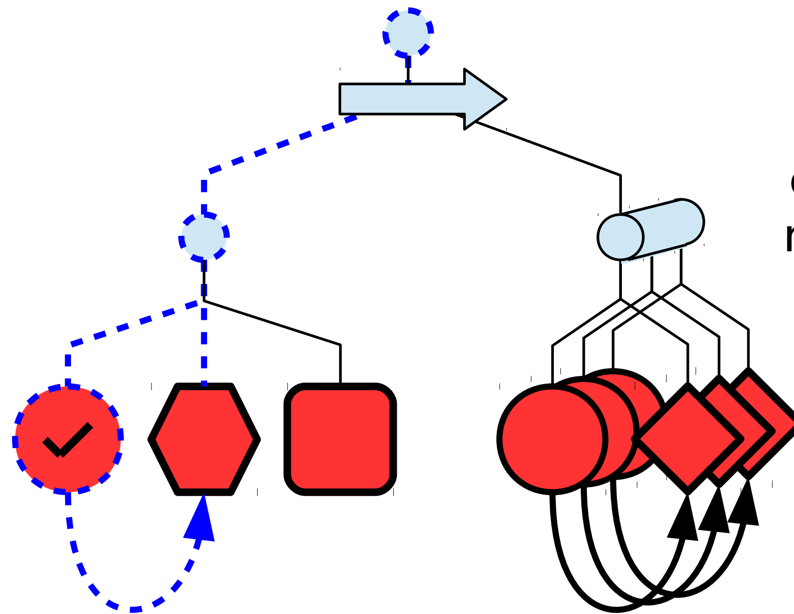


The PrecedenceSvc will directly query the ConditionsStore to determine if data is available (unlike for event data)

Subtleties – sub-event processing

Offline reconstruction assumes all algorithms executed once per event, on the whole event

The ATLAS HLT uses regional reconstruction, with multiple (or zero) executions of some algorithms on subsections of event data



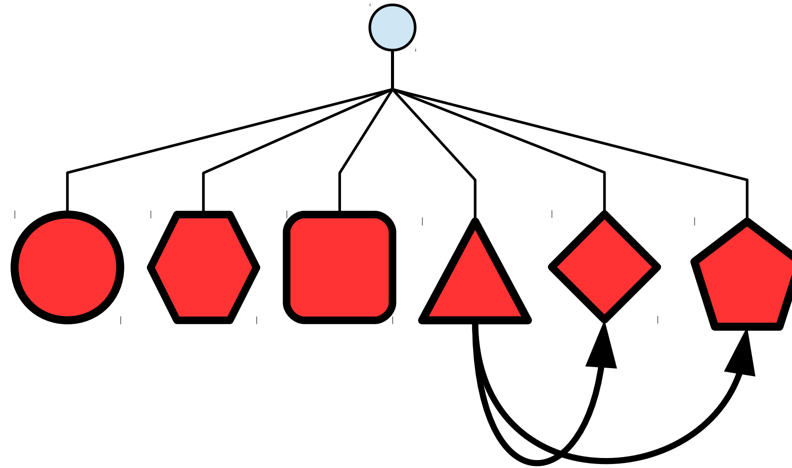
During the event loop, additional contexts are created for each sub-event region, and then associated with a given CF node

Algorithm states are stored independently for each sub-event, and data dependencies are resolved only within the regional context

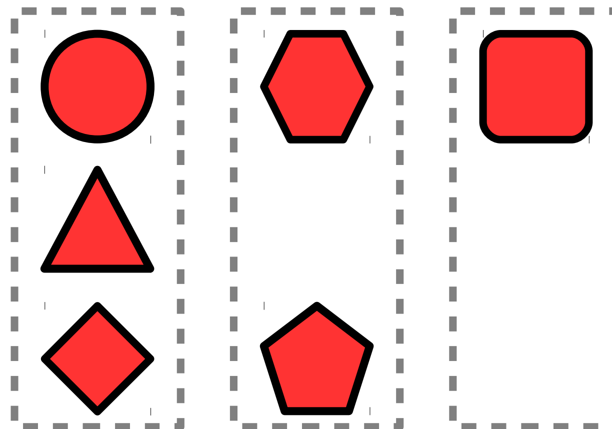
The parent node returns a decision once all sub-events are processed

Subtleties – algorithm priority

Consider the following PR graph:

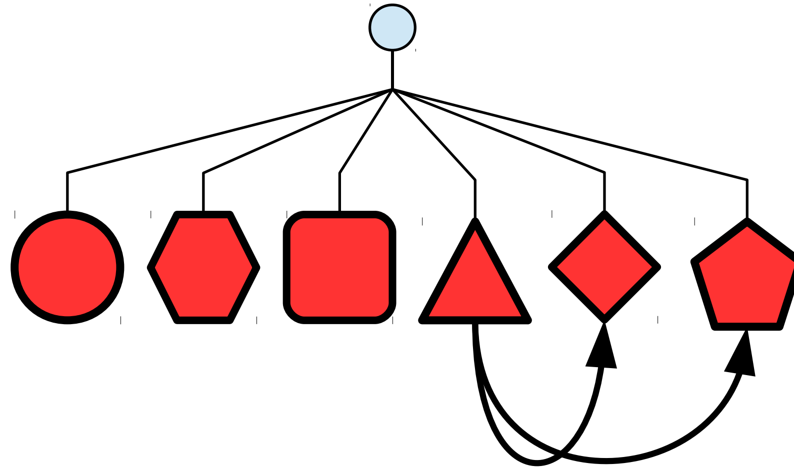


In a job with 3 available threads, naively executing algorithms in their order as children of the root CF node will lead to sub-optimal performance because of the data flow constraints:



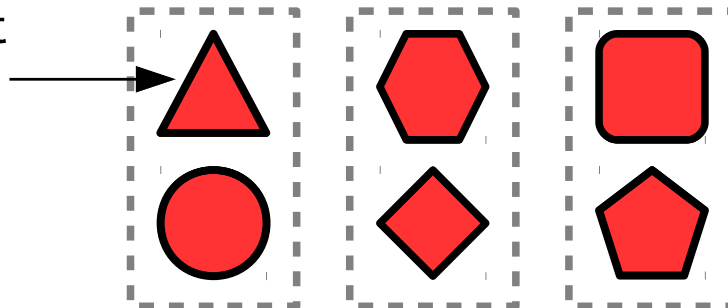
Subtleties – algorithm priority

Consider the following PR graph:



The PrecedenceSvc has a number of strategies for ranking algorithms (e.g. number of downstream data clients) to improve thread utilisation:

This algorithm has highest priorty because others rely on its output



In most real scenarios, we would expect to occupy idle threads with multiple events in flight. Nonetheless, this optimisation is a free win in applicable cases

Subtleties – stalling

Given all these rules, it's quite possible to configure a job where dependencies are not satisfied

Some errors can be caught outside the event loop, as the scheduler initialises:
- Un-matched input dependencies
- Cyclic data dependencies
- Two algorithms guaranteed to produce the same output object (conditional ok)

Since there are many things that can change within the event loop, it is still possble to reach a state where no algorithms are currently running, no new algorithms can be scheduled, and yet the CF root node has not returned a decision: this is a stall

Examples:
- Multiple algorithms might produce an output, conditional on independent filters. In a given event, none of the filters pass
- Data flow from Alg A to Alg B is interrupted when Alg B is declared part of a sub-event
- Alg A produces data required by Alg B, but CF demands that Alg B runs first (this probably could be detected in advance, but we don't currently do so)

Stalls are considered configuration errors, and should not occur in a real job. For the purposes of migration we have a (verbose) debug output from the scheduler

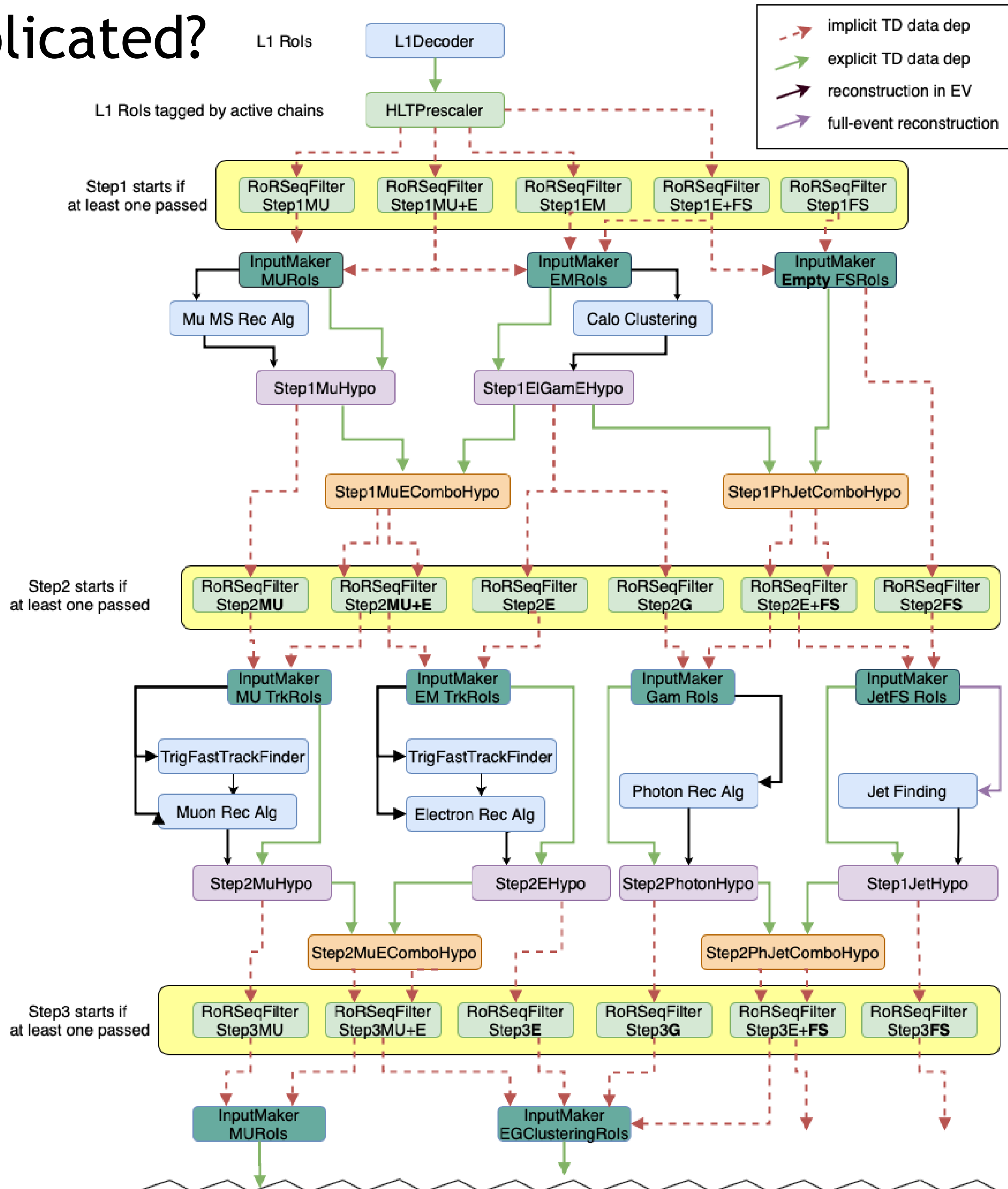
Isn't this all a bit complicated?

Do we really need all these rules? For offline reconstruction, not really

Most of the complexity comes from the HLT use-case, with conditional execution, regional reconstruction, and control flow both branching and merging

The diagram on the right (credit F. Pastore) is the "simplified" depiction of the HLT menu

One of the motivations of AthenaMT was to maximise code sharing between online and offline



Performance

I don’t have any recent, public performance figures for Athena workflows (sorry)

ATLAS internal link for HLT profiling:
https://indico.cern.ch/.../20200511_Rafal.pdf

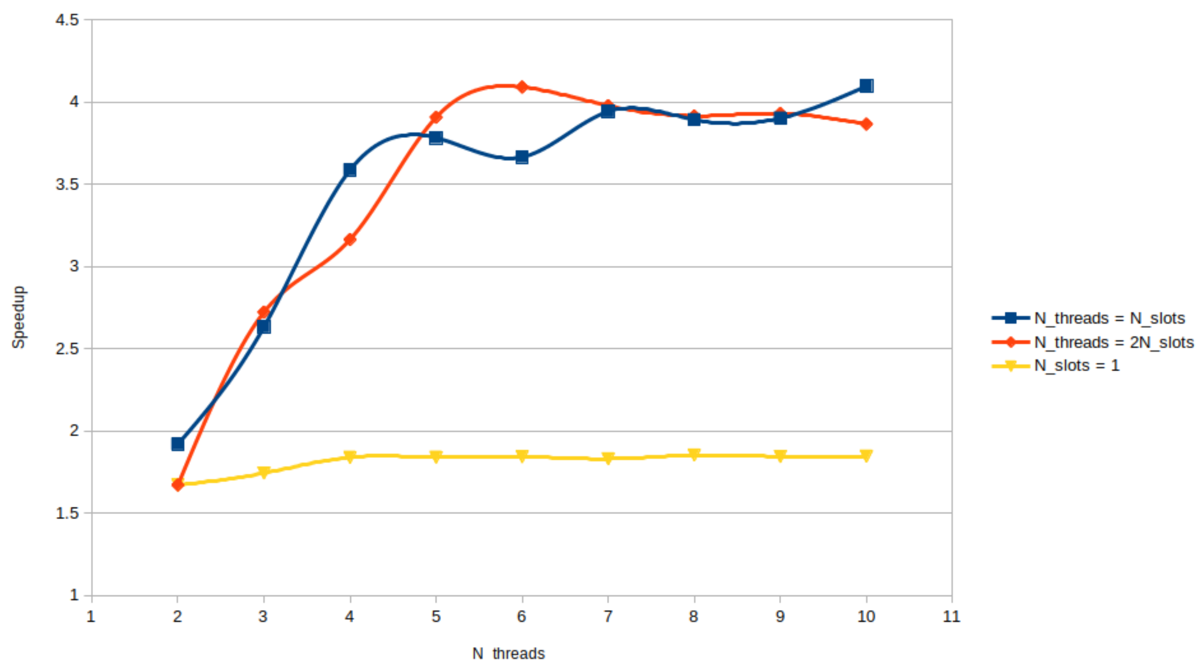
Qualitatively speaking, I can say that the memory scaling behaviour with number of concurrent events is far better than our existing multi-process forking

Throughput is currently limited by some bottlenecks that we expect to fix as we find them
 - in the tests linked above, we forgot to turn on cloning for non-re-entrant algs!

Gaudi itself has some synthetic benchmarks provided to mimic HEP workflows, so you can run some basic tests yourself

This rough set of results comes from my laptop, using the ATLAS MC reconstruction scenario

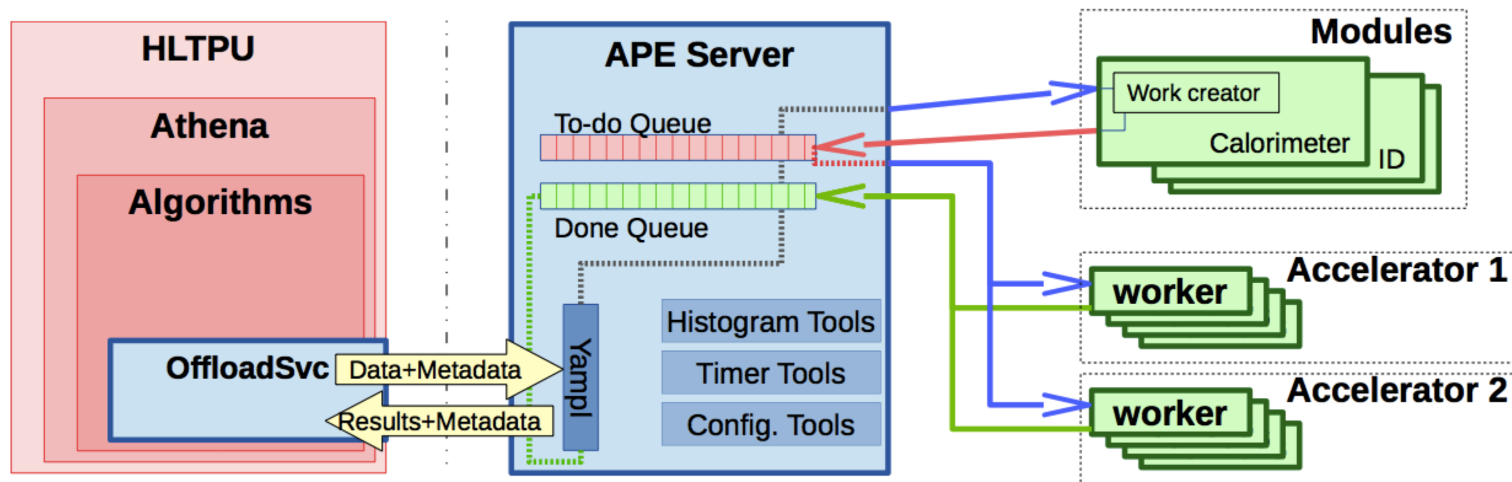
Suggests 2 threads per concurrent event as a sensible choice



Offloading

A lot of current framework development targets offloading algorithms to GPUs and other accelerators

In the past Athena/Gaudi had no "first class" support for accelerators, and this was accomplished for the ATLAS HLT GPU demonstrator project using a separate server process called APE



The server provided a uniform interface for multiple different accelerators, assuming that appropriate code was compiled for the host machine

- Supported >50 Athena CPU process clients with a single GPU, over network

- Client algorithms synchronous, simply waiting for results to be returned

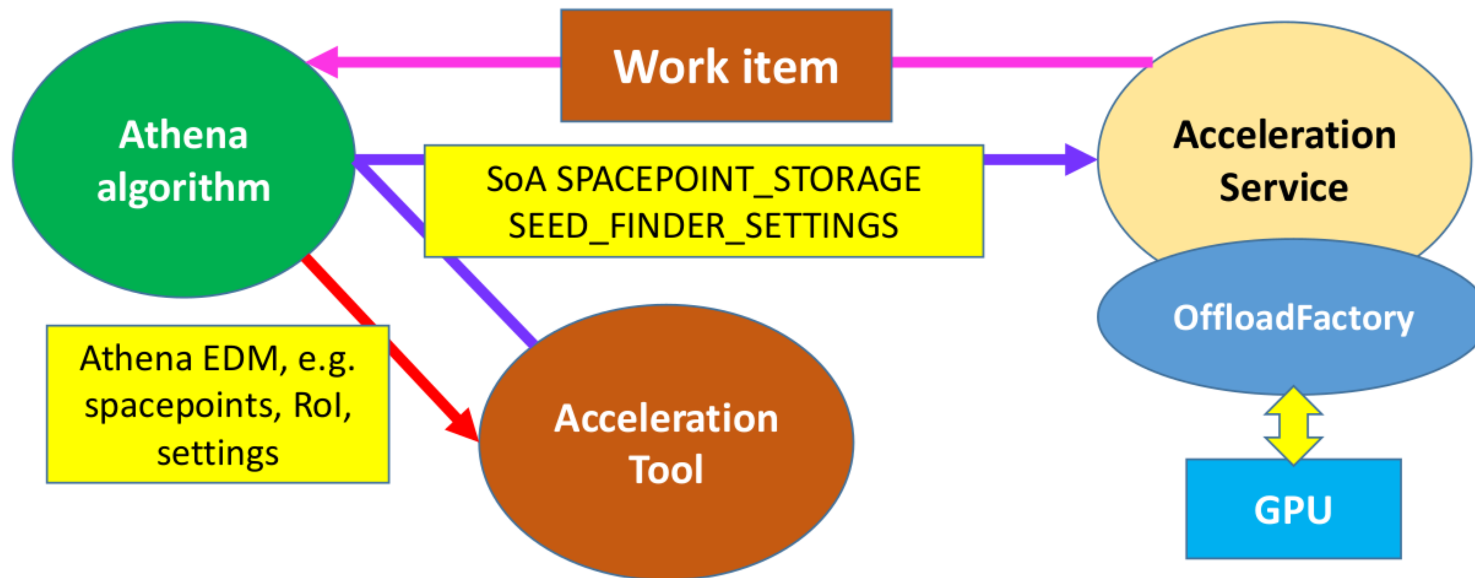
- Performance suggested server itself may have been a bottleneck

CUDA support in Athena

It is now possible to compile CUDA code within Athena, and run device kernels from within an algorithm (SYCL also supported!)

- Detect if CUDA is available, compile as C++ otherwise
- Detect if GPU is available, execute on CPU otherwise
- Creating compiler macros to standardise these options

New prototype has a hardware-specific implementation of an algorithm created by the OffloadFactory and returned to the Athena algorithm for execution, as shown in diagram below (credit D. Emeliyanov)



Current implementation of GPU track finding directly in AthenaMT MR !33485

Offloading in the scheduler

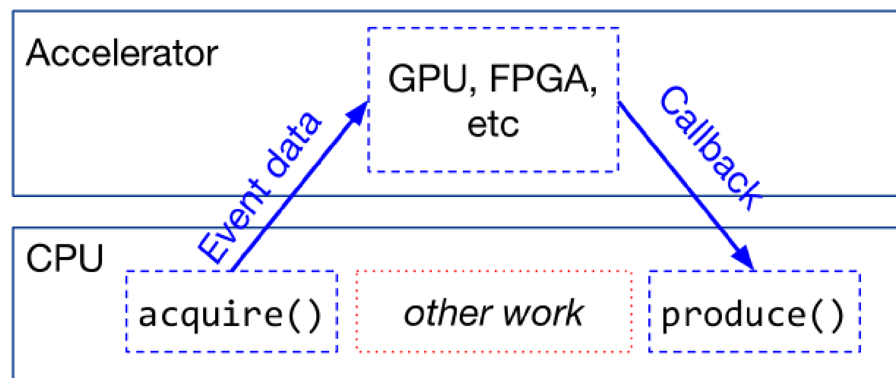
The Gaudi scheduler executes algorithms in TBB tasks by default, sharing a pool of threads

An alternative method of scheduling allows you to mark an algorithm as "blocking," meaning it will run in a detatched std::thread rather than a TBB task

- Result of algorithm execution is still pushed to the scheduler action queue

CMS uses an asynchronous scheduling approach for offloads, with the CPU thread freed and re-used while GPU execution occurs

Similar approach used in prototype variant for Gaudi scheduler



Given recent improvements to CUDA/drivers this may not be necessary any more, and "blocking" may be sufficient

- Synchronous offload leaves the host thread idle
- Little enough background polling activity that it can be automatically recycled
- Comparable performance to asynchronous offload

Task arena could be used to organise offloaded operations that require global device locking (e.g. CUDA malloc)

Summary

The Gaudi AvalancheScheduler provides multi-threaded execution in AthenaMT by identifying independent algorithms and executing them in TBB tasks

- Support for algorithms that are not re-entrant through cloning

Algorithm data dependencies are analysed at configuration time

- Event data is available for consumers if a producer has run successfully
- Conditions data availability is tested, and producer algs run on-demand

Control flow rules can be applied, may vary event-by-event workflow

- Sequential execution, with algorithm true/false decisions
- Independent processing of sub-event regions

Allows intra- and inter-event parallelism

- Memory usage scales with multiple events in flight, better than MP forking
- Improve throughput with multiple threads per event
- All threads shared, so lack/surplus of work in given events can be balanced

GPU offloading supported in the framework

- Recent development!
- May not require significant change to scheduler