*The future of **art***

Marc Paterno *for the **art** team*
22 November 2011

"Code must be scalable or its lifetime will be short."

"Scalar performance is not scaling."

—Michael McCool, Intel, at SC'11

The ***art*** *development team is*:
Walter E. Brown
Lynn Garren
Chris Green
Jim Kowalkowski
Marc Paterno

The purpose of this talk

We want to communicate:

- our starting point,
- our driving use cases, and
- our vision of where we are headed.

Who are our users?

art is used by upcoming Intensity Frontier (IF) experiments

- Experiments have *many* fewer collaborators than the collider experiments;
- They can't afford to invent their own infrastructure;
- They can't afford a full-time framework support person;
- They need both data acquisition (DAQ)—including software trigger—and offline processing infrastructure.

Current users include:

- NOνA;
- mu2e;
- muon g-2;
- LAr community: MicroBooNE, ARGONeut, LBNE/LAr.

Some other future experiments, and other Fermilab efforts, are also targets for our prototyping.

The state of **art**

- Forked from CMSSW code base $\sim$ two years ago.
- Removed several complicated features not needed by our clients.
- Replaced the build system with something we could sustain.
- Replaced software delivery system with something we could sustain.
- Added features needed by the IF experiments.
- Code size:
    - 55,000 lines of non-blank non-test source, excluding comments,
    - 23,000 lines of non-blank test source, excluding comments.
- We build with GCC 4.6.1 and Boost 1.47.0.
- We are beginning to push for `-std=c++0x`[1].

[1] Move semantics!

Features removed

The major features that were not needed by our community were:

- the `EventSetup` and all associated support
- the XML job report system
- the POOL file catalog support
- the collection class templates which supported containing base classes (sometimes called "polymorphic collections")
- the persistent reference templates in support of the container templates we removed
- the "one file, two file" code to support breaking data files into tiers

Features changed

- Simplified the `Ptr` and `PtrVector` class templates.
- The plug-in system, including loading of dynamic libraries; replaced with a system decoupled from the build process, and relying upon simple naming conventions. Modules, services, and Root dictionaries are now directly handled, and are less coupled.
- The services system; decoupled from library loading; we supported *required*, *optional*, and *user* services. Each experiment can introduce its own services without modifying the framework.
- The Python-based configuration system, replaced with a YAML[2]-like language, shared by projects not using **art**.

We have done widespread refactoring in many places.

[2] See `http://www.yaml.org`

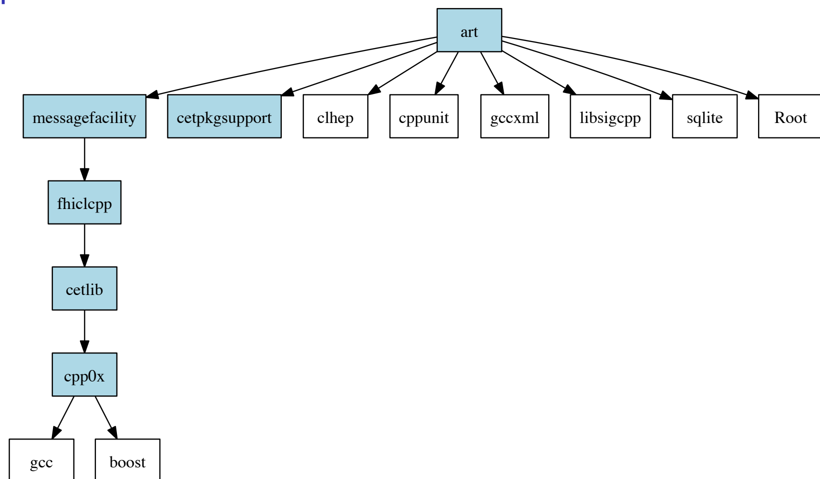
Some of the new features in **art**

- External product references: the `Assns` class template and supporting `FindOne` and `FindMany` “smart query” class templates, providing bidirectional inter-product references.
- The `MixFilter` class template, providing a generic facility for experiments to “mix” data from additional files into their primary data stream. Used by mu2e to mix thousands of background events into each signal event.
- The `ReaderSource` class template, providing a generic facility for experiments to write their own input sources.
- A facility for embedding an *SQLite3* database in the ROOT event data file (thanks to P. Russo), which the framework uses for storage of meta-data and which experiments can use for their own needs.

Changes and additions to the **art** ecosystem

- We replaced SCRAM with a build system based on *CMake*, which has been easy to maintain and has provided excellent support for parallel builds, including parallel running of tests.
- We deliver software products (both ones we write and ones upon which we depend) using an enhanced "relocatable" UPS. Installation of the entire system is performed by downloading of tarballs and unwinding them in a directory of the user's choice. No *root* permissions are needed.
- **art** is an external dependency for the experiments, provided as an "umbrella product" called the *art suite*.
- Internally, the art suite is layered as a set of products, each of which has its own dependencies.
- Our build system understands our layering of products and uses UPS to set up dependent products, rather than having a "base release" and a "test release", thus avoiding inconsistent builds.

Dependencies of **art**



- Transitive dependencies are suppressed in this diagram.
- Each box is a UPS *product*.
- Blue boxes indicate the products we directly support.

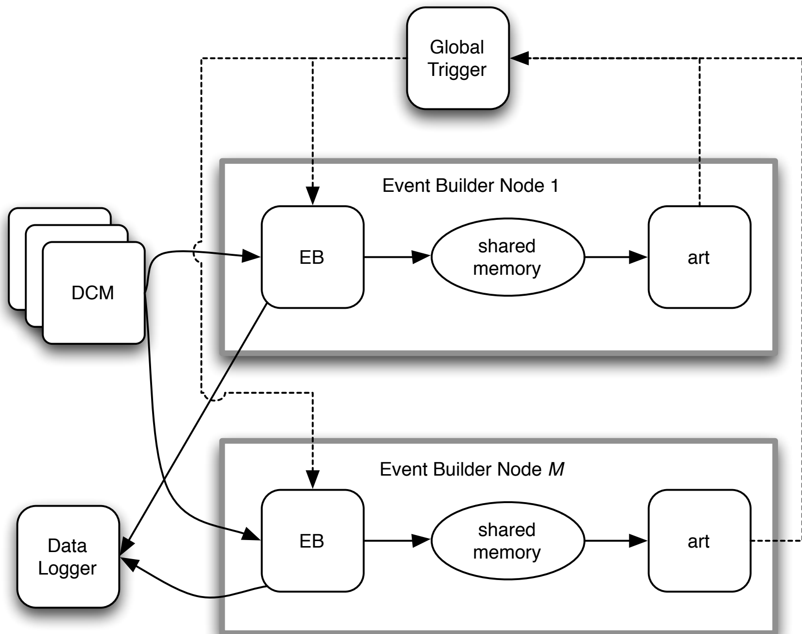
Uses cases guiding our efforts

Disclaimer: While these use cases are guiding our efforts, we are *not* saying that all these groups have decided to follow our designs.

- We are using our understanding of the DAQ needs of several IF experiments, as well as some other efforts, to guide our development efforts.
- These efforts share a need for development.

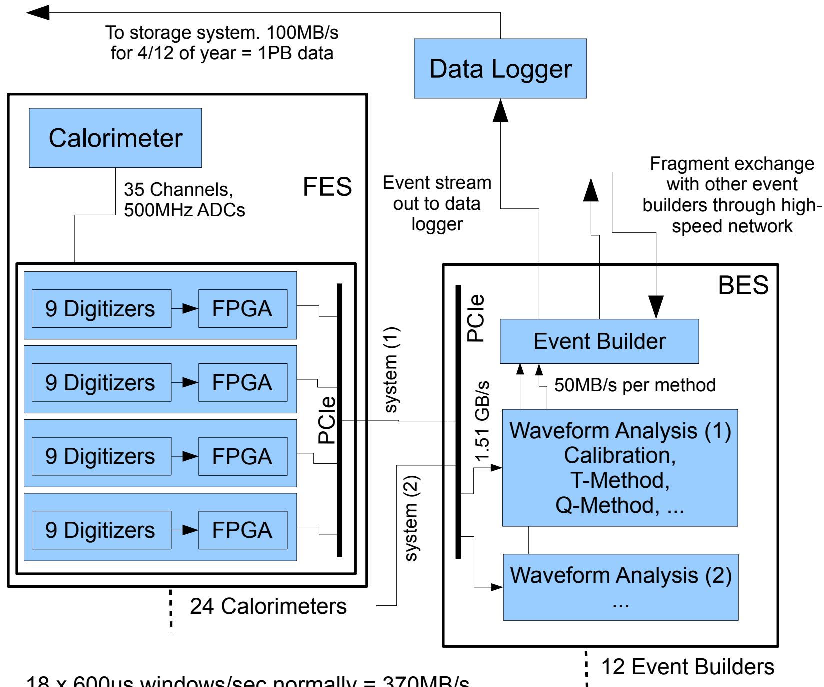
Use case #1: the NOνA data-driven trigger

- designed to continuously process data at full sampling rate
- collects and buffers time-continuous data from over 350k readout channels
- every time window is analyzed
- for the far detector, custom designed upstream hardware
- 5ms time slices into 180 multicore commodity computing nodes, at 2GB/s.
- positive trigger decisions fed back to global trigger to cause readout of the data.


trigger [start, duration]
Global Trigger
Event Builder Node 1
DCM
EB
shared memory
art
Event Builder Node M
Data Logger
EB
shared memory
art

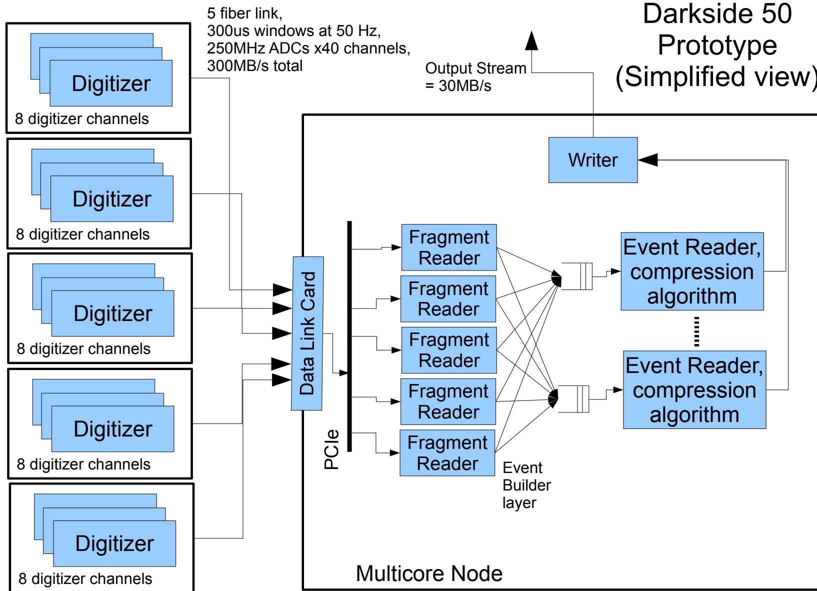
Use case #2: the muon g-2 DAQ

Our goal is to see how few BES nodes can perform the task.


To storage system. 100MB/s
for 4/12 of year = 1PB data
Data Logger
Calorimeter
35 Channels,
500MHz ADCs
FES
9 Digitizers
FPGA
9 Digitizers
FPGA
9 Digitizers
FPGA
9 Digitizers
FPGA
PCIe
24 Calorimeters
Event stream
out to data
logger
system (1)
system (2)
Fragment exchange
with other event
builders through high-
speed network
BES
PCIe
Event Builder
50MB/s per method
1.51 GB/s
Waveform Analysis (1)
Calibration,
T-Method,
Q-Method, ...
Waveform Analysis (2)
...
12 Event Builders
18 x 600us windows/sec normally = 370MB/s,
4x18 x 600us windows/sec maximum = 1.51GB/s

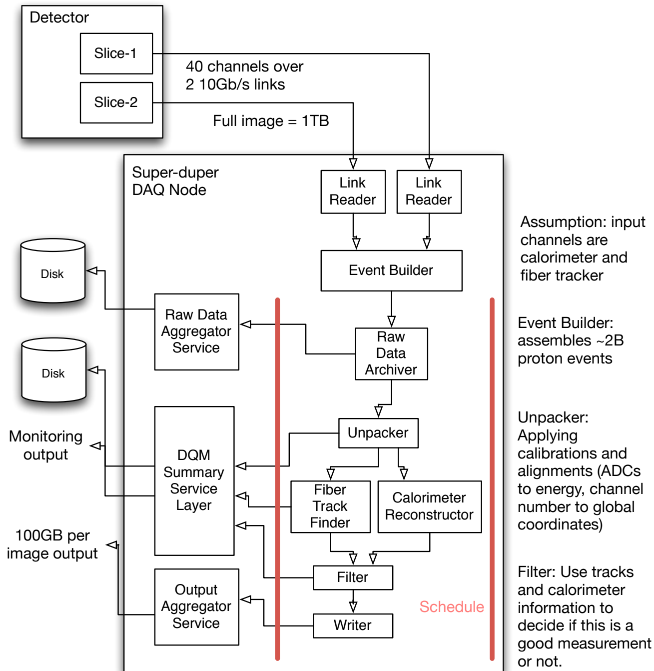
Use case #3: the DarkSide-50 DAQ

Our goal is to see how few nodes are needed for this task.


Darkside 50
Prototype
(Simplified view)
5 fiber link,
300us windows at 50 Hz,
250MHz ADCs x40 channels,
300MB/s total
Output Stream
= 30MB/s
Digitizer
8 digitizer channels
Digitizer
8 digitizer channels
Digitizer
8 digitizer channels
Digitizer
8 digitizer channels
Digitizer
8 digitizer channels
Data Link Card
PCIe
Writer
Fragment Reader
Fragment Reader
Fragment Reader
Fragment Reader
Fragment Reader
Event Reader, compression algorithm
Event Reader, compression algorithm
Event Builder layer
Multicore Node

Use case #4: the PCT testbed

The Proton Computed Tomography (PCT) project is a joint effort between NIU, LLU Medical Center, SCIPP and Fermilab to produce a high-resolution medical imaging system using a proton beam.
Our goal is to see how rapidly the data can be collected and converted to raw tracks, ready for processing at the image reconstruction cluster.


Detector
Slice-1
Slice-2
40 channels over
2 10Gb/s links
Full image = 1TB
Super-duper
DAQ Node
Link
Reader
Link
Reader
Event Builder
Disk
Raw Data
Aggregator
Service
Raw
Data
Archiver
Disk
Monitoring
output
Unpacker
DQM
Summary
Service
Layer
Fiber
Track
Finder
Calorimeter
Reconstructor
100GB per
image output
Filter
Output
Aggregator
Service
Writer
Schedule
Assumption: input
channels are
calorimeter and
fiber tracker
Event Builder:
assembles ~2B
proton events
Unpacker:
Applying
calibrations and
alignments (ADCs
to energy, channel
number to global
coordinates)
Filter: Use tracks
and calorimeter
information to
decide if this is a
good measurement
or not.

The **art-mt** framework architecture prototype

We want to obtain both *event-level* and *subevent-level* parallelism.

- Input source runs in its own thread.
- In our initial version, runs must be processed serially, as will subruns. This is not a problem for DAQ applications.
- `Events` are read by the input, and put onto a shared queue
- `Schedules` pop `Events` from the shared queue, and process them asynchronously.
- Our initial use cases do not require output modules, avoiding threading issues with Root IO. Of course, the IO problem will have to be handled eventually.

Services

We have two categories of services:

- Shared services, which have the same state for the whole program These include *e.g.*, `TFileService`, which must deal with Root's global state, and `SimpleMemoryCheck`, which is observing the global state of the process. Only one instance will exist for each of these services.
- `Schedule`-specific services, *e.g.*, `CurrentModuleService`, which understands which module is executing in the `Schedule` to which it belongs. Each `Schedule` will contain its own instances of each `Schedule`-specific service.

Signals and callbacks

- **art** issues *signals* to indicate the occurrence of *events* during the event processing life-cycle.
- Each signal is issued either by the application object and so is program-wide, or by a `Schedule` and then is thread-specific.
- There is no issue of asynchronous processing of signals, because they are defined parts of the event loop.

Modules

Event-level parallelism "happens" in the `Schedule`.

- Except for the input, modules configured by the user will be replicated in each `Schedule`.
- Modules must use only thread-safe libraries.
- Modules must be written in a thread-safe style.

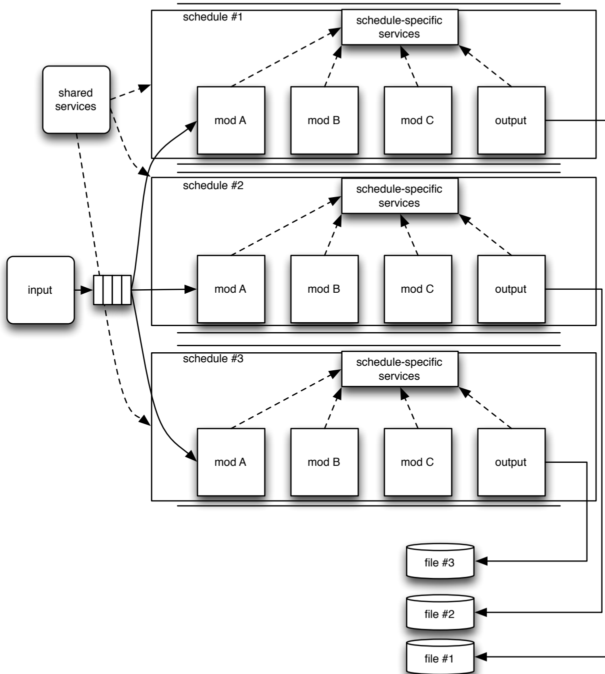
When we moved from Fortran to C++, we had to learn:

- code that leaks memory is wrong code, and
- code that is not exception safe is wrong code.

Now we are writing C++ code for concurrent systems, and we have to learn:

- code that is not thread safe is wrong code.

Our goal is to identify one or more programming models that make writing concurrent code easier.


schedule #1
schedule-specific services
shared services
mod A
mod B
mod C
output
schedule #2
schedule-specific services
input
mod A
mod B
mod C
output
schedule #3
schedule-specific services
mod A
mod B
mod C
output
file #3
file #2
file #1

Sketch of our work plan

We do not (yet) have a detailed plan. We plan to introduction functionality in steps.

1. Input source that reads NOνA raw data from shared memory and creates `Events` on a shared queue.
2. `Schedule` that draws `Events` from the queue.
3. `EDProducer` or `EDFilter` that posts DDS[3] messages announcing triggers.
4. Multiple `Schedules` reading from the queue.
5. GPU-enabled algorithms running in some modules.
6. Subevent-level parallelism in some modules.

[3] Data Distribution Service, used by the NOνA DAQ for messaging.

Promising technologies

We are only beginning to investigate these possibilities.

For GPU programming:

- CUDA
- Thrust (C++ template library for more "accessible" CUDA)
- OpenCL

OpenACC, just recently announced, does not seem plausible—unless GCC were to adopt it. We will watch its progress.

For CPU multi-threading:

- OpenMP
- Intel Thread Building Blocks

Cilk++, in GCC 4.7 development thread, seems too speculative until 4.7 is released. Intel Array Building Blocks would have been interesting, but will it regain support?

Still further in the future . . .

- We need to have concurrent-capable support for histograms and ntuples.
- We will be working towards a fully demand-driven system, using task parallelism, first at the framework level and later in the module.
- The technologies we are moving toward are those embraced by the HPC community.