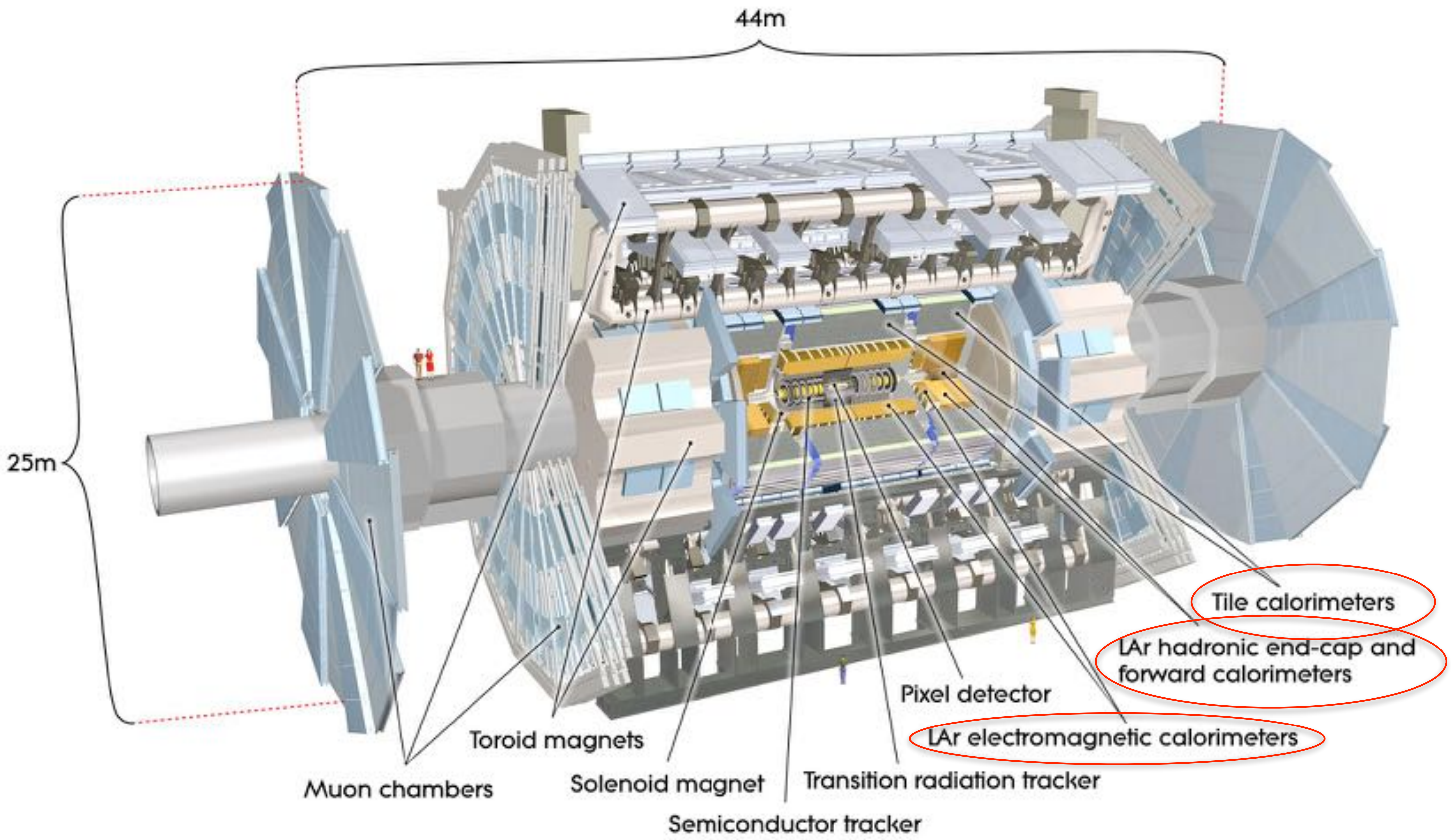
ATLAS Calorimeter Software

Walter Lampl, University of Arizona

Disclaimer: Performance numbers (CPU, Memory) given in this talk are indicative

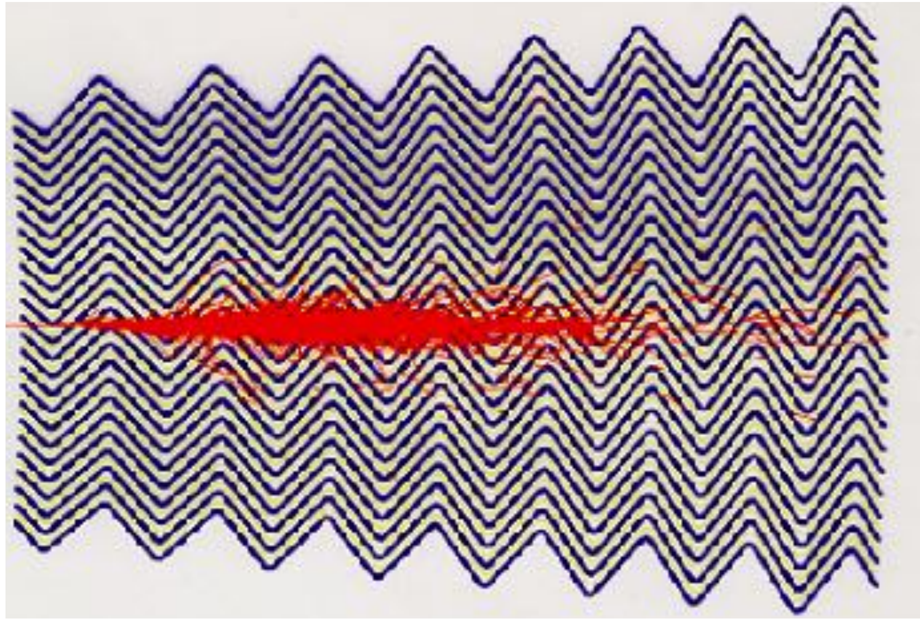
Depend on data being processed and hardware

Calorimeters in the ATLAS Detector

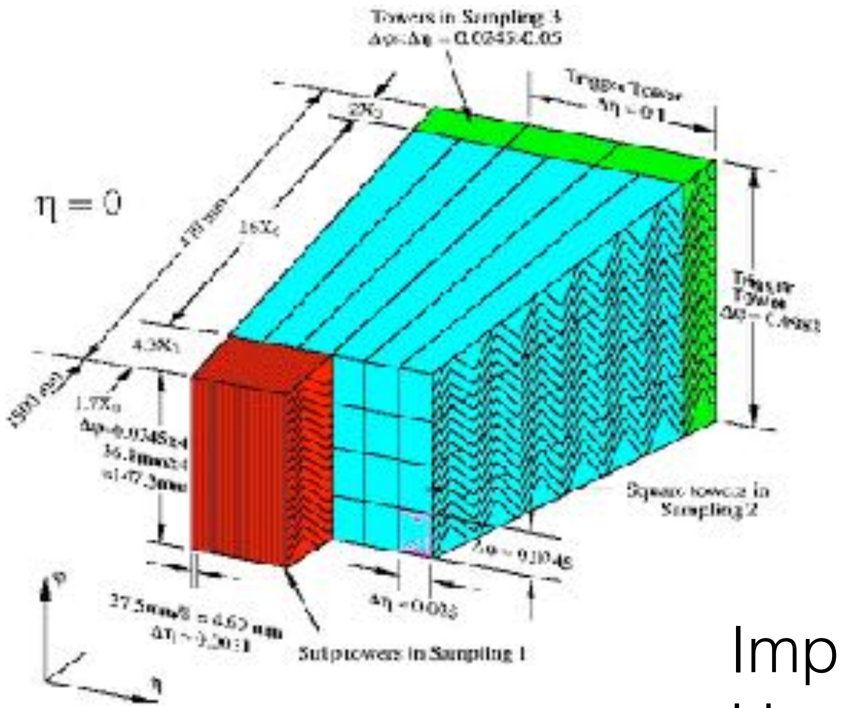


The basic principles of calorimetry

- Calorimeters are built out of dense material
  - In case of ATLAS these are Lead, Iron and Copper interleaved with *active material* like Liquid Argon or scintillating plastic “tiles” -> “Sampling Calorimeter”
- Incoming particles create a shower of secondary particles that is (ideally) completely absorbed inside the calorimeter
- While traversing the active material, the ionising particles in the shower leave a signal that is read out electronically
  - This signal is proportional to the energy of the incoming particle
  - The showering is a stochastic process!
- Two types of calorimeters:
  - **Electromagnetic calorimeter**: Measures the energy of photons and electrons
  - Hadronically interacting particles (like pions) typically create a shower that penetrates the EM calorimeter and leaves most of its energy in the **hadron calorimeter** that is located behind the EM calorimeter



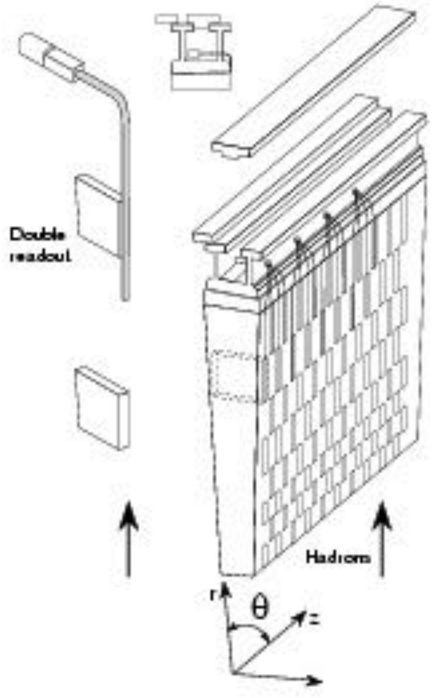
Calorimeter Technologies in ATLAS



EM-Barrel: Liquid Argon in accordion-geometry:
Perfectly hermetic coverage in phi

Important for the software:
Huge number of readout channels (fine detector granularity)

Hadronic Barrel:
Tile-Calorimeter



Sub-Calo	Nbr of Channels
EM Barrel (LAr)	109696
EM Endcap (LAr)	62208
Had Endcap (LAr)	5632
Had Barrel (Tile)	5184
Forward (LAr)	3524

Total: 186244

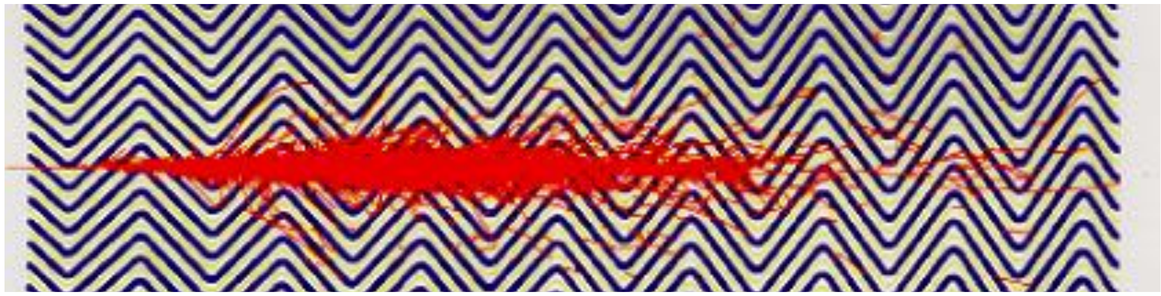
All calorimeter cells are read out on every event

Detector Simulation of the Calorimeter

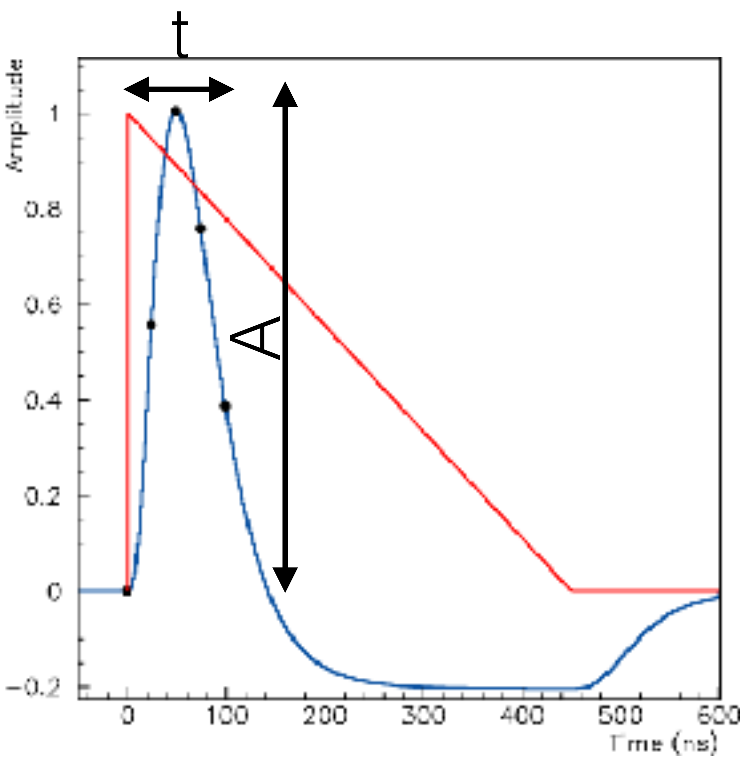
- Traditional detector simulation (using the Geant4 simulation software) tracks every particle in the shower through the accordion-geometry and simulates the energy deposit in the active material

- Because of the large number of particles in the shower, this is very CPU-intensive

- Fast-Calo simulation: An approximation of the detailed shower simulation that is much less CPU-intensive
  - See Heather’s talk this afternoon



Calorimeter Readout & Data objects



- Ionization signal (red) is shaped (blue) and digitized at four points.
- Peak is determined by the *Optimal Filtering* method
  - Real Data: Done by a DSP part of readout electronics
  - Simulated data: Done (after smearing with electronic noise) done as part of the Digitization
- For each cell, we store energy (prop. Amplitude)
  - For cells with energy above a threshold (usually 5 sigma-noise), we also store the time and a quality-factor (how well the pulse shape matches the expected one)

Digitization / Real Detector

Reconstruction

LAr(Tile)Hit

Simulated
- Energy
- Time
- Identifier

LAr(Tile)Digit

- ADC samples
- readout-Identifier

LAr(Tile)RawChanel

- Energy
- Time
- Quality
- readout Identifier

CaloCell

Calibrated
- Energy
- Time
- Quality
- Identifier
- ptr to DetDescr

CaloCluster

- Energy
- Time
- Location/Direction
- Variable list attributes

Some data size estimates

- In ByteStream (what comes out of the detector) LAr+Tile is about 850 kBytes/event (>50% of the total)
- The CaloCell container in the output file is highly-compressed 260kBytes/Event
    - In memory (back-of-the-envelop calculation) ~ 5.3 MBytes
        - Not counting the static geometry information
- LArRawChannel: (back-of-the-envelop calculation) ~4.4 MBytes
- Clusters have too many dynamic attributes to make any generally-valid size estimate

Real Data: LArRawChannel Building

- For almost all cells, the Energy is already computed by the readout electronics and written as such to the byte-stream
- For high-energy cells (>5 sigma-noise) we store also the raw ADC samples and re-reconstruct the cell-energy offline.
  - Slightly better precision, in particular time and quality
  - Allow corrections if the calibration used online is found to be less-than-optimal
- Tiny CPU usage (<10ms/event) but need some 80MByte RAM to store the electronic calibration constants.

Digitization: Simulate detector electronics for simulated data

For calorimeters, that means:

1. Convert Hits (simulated energy deposits) to Digits (‘fake’ ADC samples)
    - Sum hit-energies from signal and pile-up (concurrent collision) events
    - LAr: Simulated cross-talk btw neighbouring cells by moving some fraction of the energy to their neighbour
    - Convert Energy to ADC counts
    - Create ADC-samples by scaling the known pulse-shape with the energy
    - Smear the samples with the (known) electronic noise
2. Apply the Energy reconstruction like done for real data
    - Get the signal peak using optimal filtering, convert ADC to MeV values

- Result is an `LAr/TileRawChannelContainer` like the one read from RAW data
- LArPileUpTool (main digitisation tool) takes about 1 sec for mu=40

With ideal electronics w/o noise or cross-talk
we would end up with the same number

Cell-Building

- Re-organizing the data read from ByteStream (or digitized RDO) so that we have a complete and ordered container of CaloCells
    - Organised by “offline hash” (index)
- Mask cells known to suffer from pathological noise
- Apply corrections for:
    - High-Voltage variations or HV trips
    - Baseline-shift due to LHC bunch structure
    - Patching of known dead cells based on their neighbours
    - Hooks for ad-hoc corrections of energy or time (mostly not needed)
- In a recent real-data (Tier-0) reconstruction job of data taken this year, this steps takes about 200 ms/event (out of ~20 sec total event processing time)
    - No dependency on pile-up (how busy the event is), since we are always processing all cells anyway

Cells are (almost) independent of each other: Could run building of each cell in parallel
Exception: Neighbour patching

Calo-Cluster Algorithms

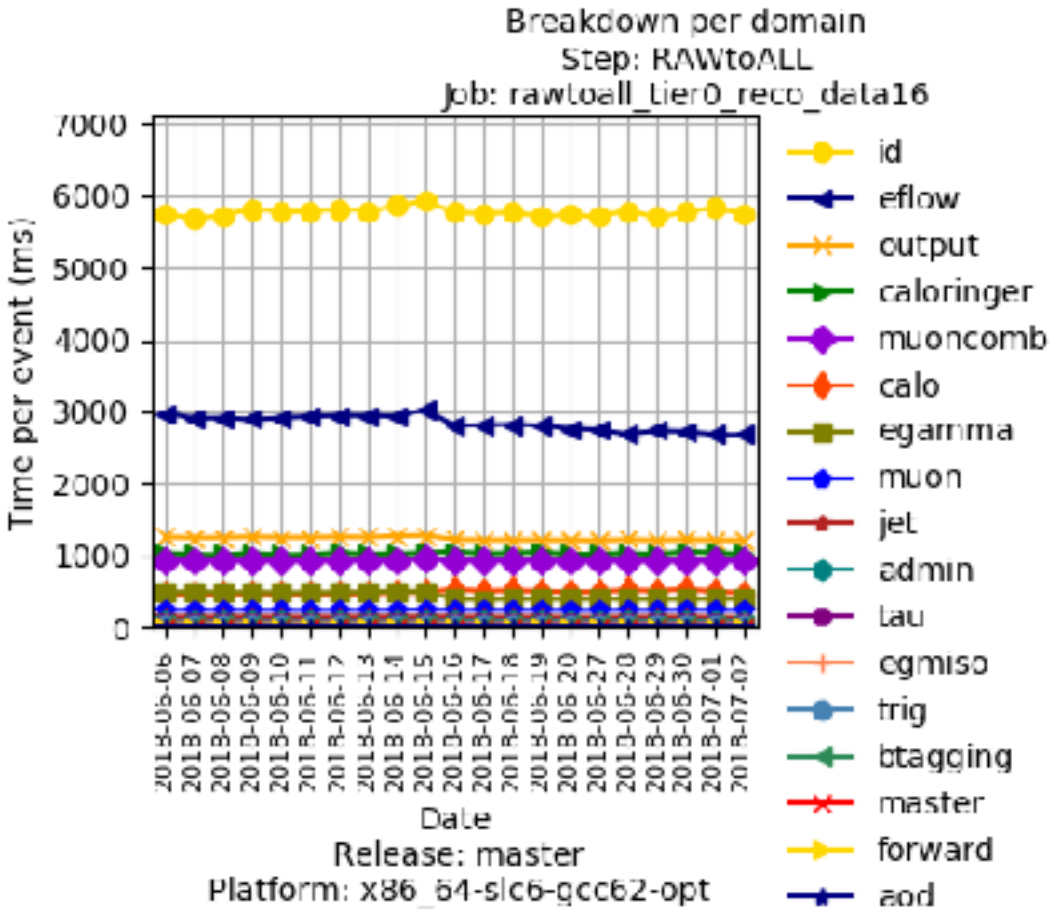
- One particle hitting the calorimeter creates a shower that spans several calorimeter cells
    - The fine granularity allows us (among other things) to distinguish showers from different particles
- Summing the energy of these cells to reconstruct the energy and direction of the incoming particle is known as **Clustering**
- Two basic strategies:
    - Topological clustering: Start from a high-energy seed cell and cluster neighbouring cells depending on their signal/noise ratio
        - Used hadronic clusters (jets)
    - Fixed-size clusters: Found by sliding-window algorithm, maximising the energy in the window
        - Used for electrons and photons (aka egamma objects)
- Clusters may overlap, leading to cells being shared between clusters
- The input to clustering is the full container of all ~190k calorimeter cells
- The cluster energy (and position) is then corrected/calibrated for a things like the energy lost upstream of the calorimeter or the impact point.
    - These corrections usually rely an calibration contents read from a database
- Topo-Cluster algorithm (+ corrections) takes about 130 ms/event
- egamma-clustering takes about 15ms/event

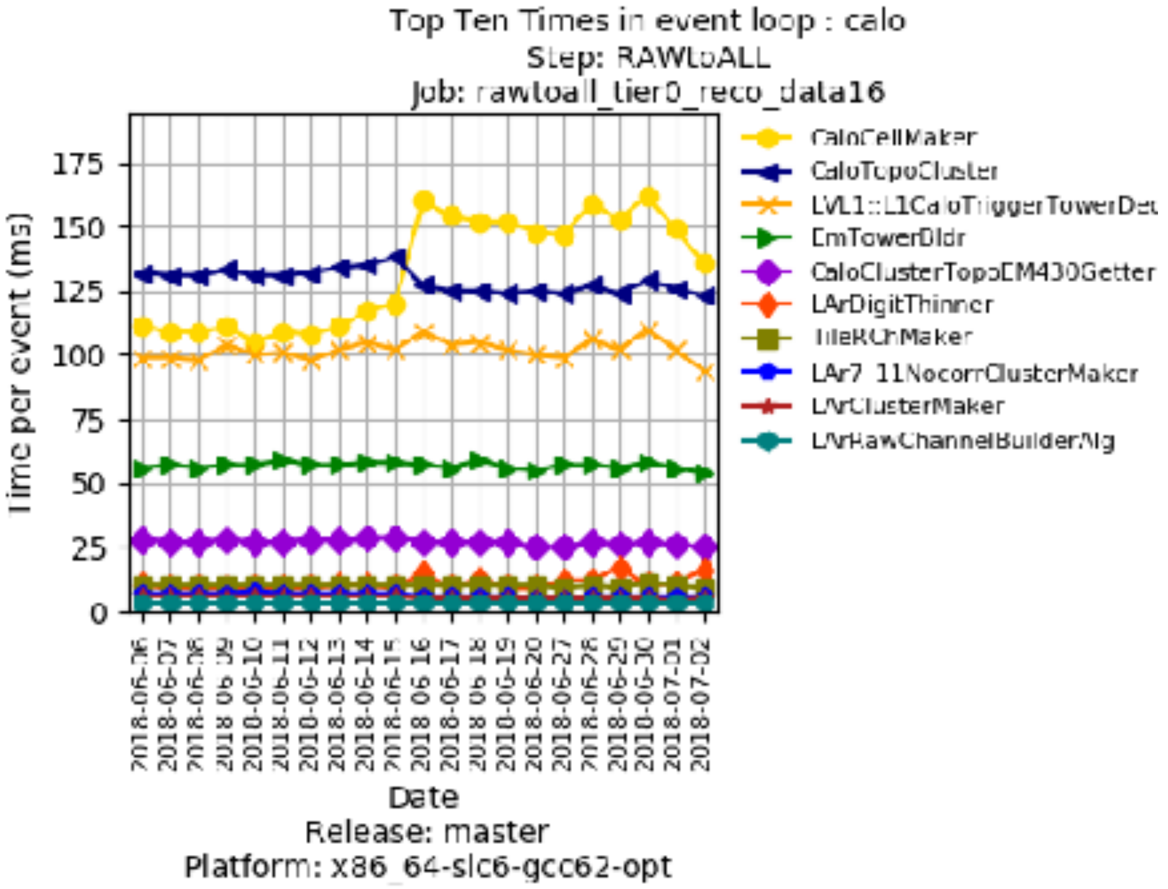
Conditions data required

- For simulated data, the conditions are fixed for the entire job
    - And in some cases we assume phi-symmetry to save memory
- For real data reco, we rely on more detailed and measured conditions:
    - For every cell, no symmetry assumptions
    - Evolution over time, sometimes changing within one reconstruction job
- Examples:
    - List of known problematic channels
    - Measured High-Voltage values, in particular when a HV-line trips (happens few times per run)
        - Noise (electronic and pile-up), needs to be re-scaled if HV changes
    - Luminosity
- Examples of stable conditions read from the database are Cluster calibration data, Cabling and Alignment

Calo Reco summary

- Large amount of event data to be processed
- Large amount of conditions data needed
- No much math, practically only multiplications with calibration constants
    - Deriving the constants is much more challenging from the math point of view, but done only 1/week (or 1/year) instead of kHz
- Plots below are from out “Performance Monitoring System”: Same job on every nightly on a otherwise quiet machine





Migration to athenaMT (multi-theading)

- Sort-of-works as long as the conditions data doesn’t change (like MC case)

- Migration of the code handling conditions data (in particular the high-voltage) to be done

Possible CPU improvements

- There is certainly room for improvement if we aggressively re-organize data structures
- Examples:
  - Skip LArRawChannels, make CaloCells straight from ByteStream
    - Done already in High-Level Trigger
  - Replace current Cell-Container (`DataVector<CaloCell>`) by flat arrays of E,t,Q (struct-of-arrays)
  - Arrange calibration constants in the same way
    - Should allow SIMD in cell-calibration
- Not done (yet), because:
  - Lots of work
  - Lose hooks for data-quality monitoring
  - Little benefit, since the CPU time is small compared to other things happening in the same job
- Backward compatibility requirements: New versions of our software is supposed to be able read old bytestream and reconstruct it using old conditions data

The End