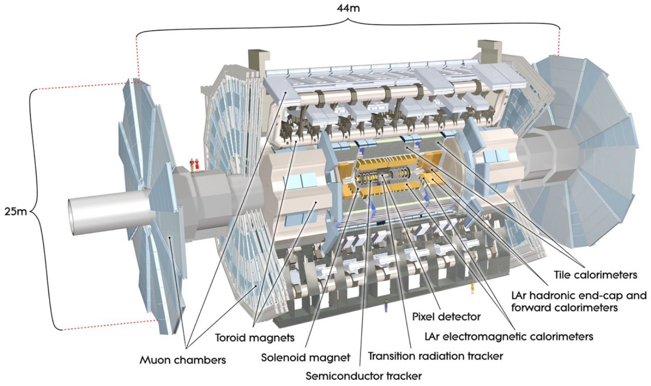
The new software based readout driver for the ATLAS experiment

Serguei Kolos,
University of California Irvine
On behalf of the ATLAS TDAQ Collaboration

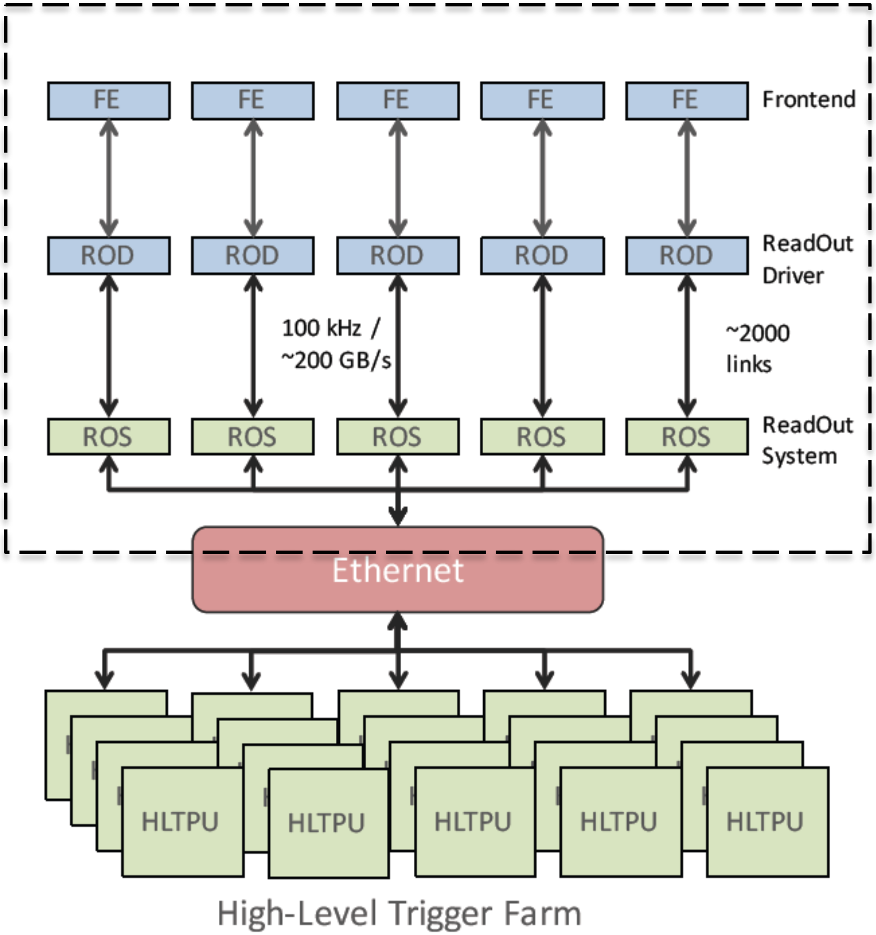
LHC Performance and ATLAS TDAQ Evolution

	Period	**Energy** [TeV]	**Peak Lumi** [10^{34} cm^{-2}s^{-1}]	**Peak Pileup**
Run 1	2009 - 2013	7 - 8	0.7	35
Run 2	2015 - 2018	13	2	60
Run 3	2022 - 2024	13 - 14	2	60
Run 4+	2027 -	14	5 - 7.5	140 - 200



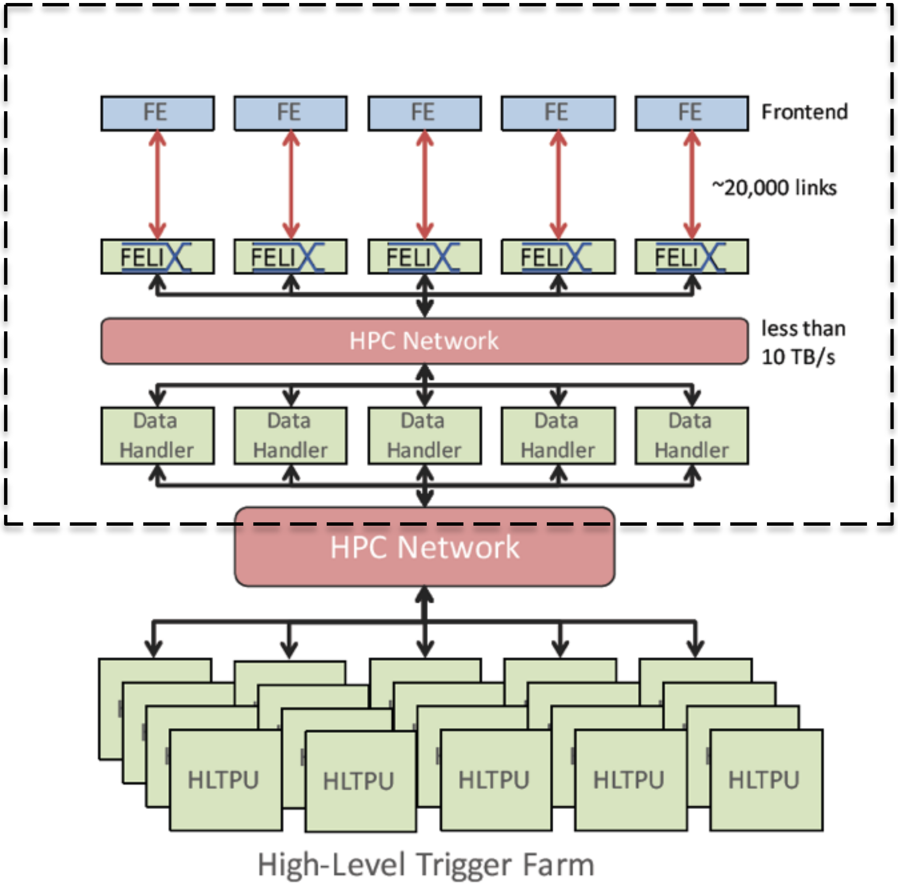
- ATLAS TDAQ system evolution has been mainly driven by the evolution of LHC performance
- The current system still copes with updated requirements:
  - Upgrading individual components was sufficient
- High Luminosity LHC upgrade will be done after Run 3
- It will require a major upgrade of the ATLAS TDAQ system:
  - Phase-2 upgrade will take place during Long Shutdown 3 between Run 3 and Run 4

ATLAS TDAQ Readout for Run 1 & 2



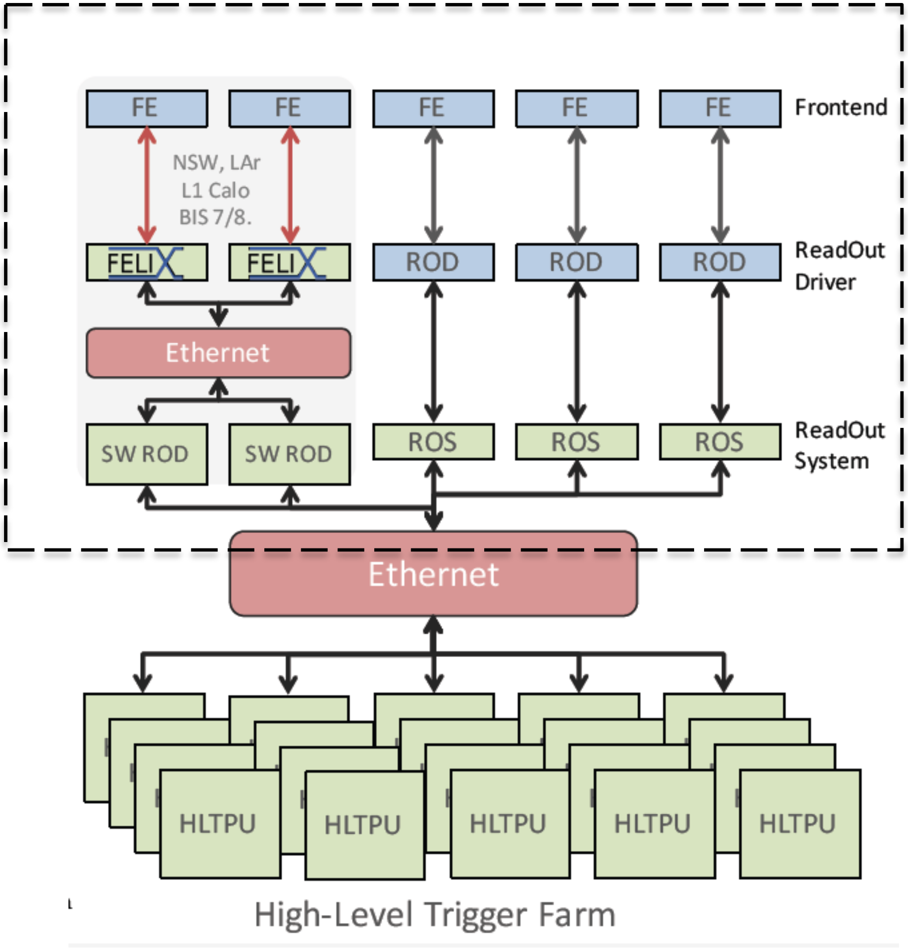
- Readout Drivers (RODs) provide interface between Front-End (FE) and DAQ:
  - A dozen different flavors of VME boards developed and maintained by detectors
  - Connected via point-to-point optical link to a custom ROBin PCI cards
- ROBin cards are hosted by Readout System (ROS) commodity computers:
  - Transfer data to the High-Level Trigger (HLT) farm via a commodity switched network
- Evolutionary changes for Run 2:
  - A new version of the ROBin card called ROBinNP used PCIe interface

ATLAS Readout for Run 4



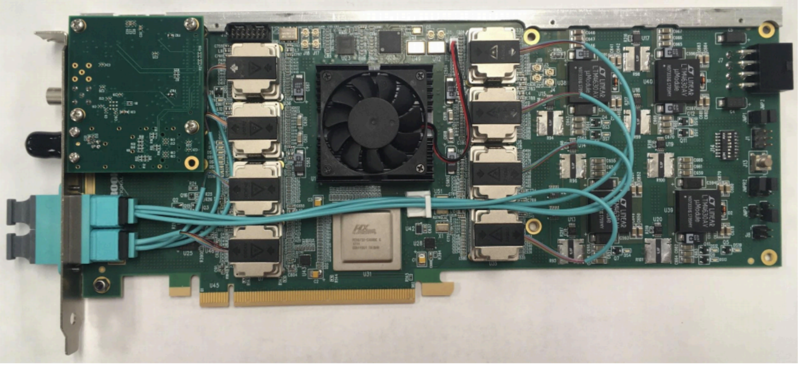
- HL-LHC upgrade will eventually provide:
  - Up to 7.5 times of nominal luminosity
  - Up to 200 interactions per bunch crossing
- Readout Upgrade Requirements:
  - 1 MHz L1(L0) rate (10x)
  - 5.2 TB/s data readout rate (20x)
- New readout architecture is based on the **FELIX** system:
  - Transfers data from detector Front-End electronics to the new **Data Handler** component of the DAQ system via a commodity switched network

The ATLAS Readout Evolution: Run 3



- ATLAS will use a mixture of the legacy and new readout systems
- First generation of FELIX system will be used for the new Muon and Calorimeter detector components and Calorimeter Trigger
- A new component, known as the **Software Readout Driver (SW ROD)** has been developed:
  - Will act as a **Data Handler**
  - Will support the legacy HLT interface

FELIX Card for Run 3



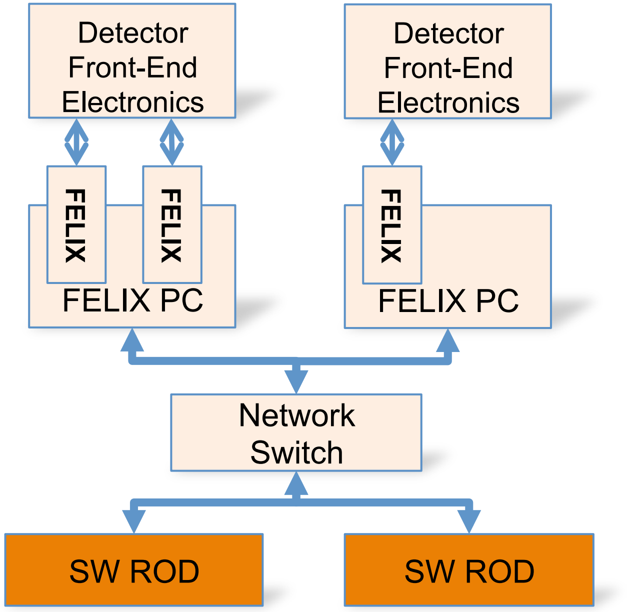
- A custom PCIe board with Gen 3 x 16 interface installed into a commodity computer:
  - 24 optical input links for data taking
  - 48 links variant exists for larger scale Trigger & Timing distribution
- Can be operated in two modes:
  - GBT Mode:
    - 4.8 Gb/s per link input rate
    - Each link can be split into multiple logical sub-links (E-Links)
    - Up to 192 virtual E-Links per card for Run 3
  - FULL Mode:
    - 12 links at full speed or 24 links with 50% occupancy
    - Up to 9.6 Gb/s per link input rate
    - No virtual link subdivision for Run 3

---

* A dedicated talk about FELIX was given earlier in this session by Roberto Ferrari

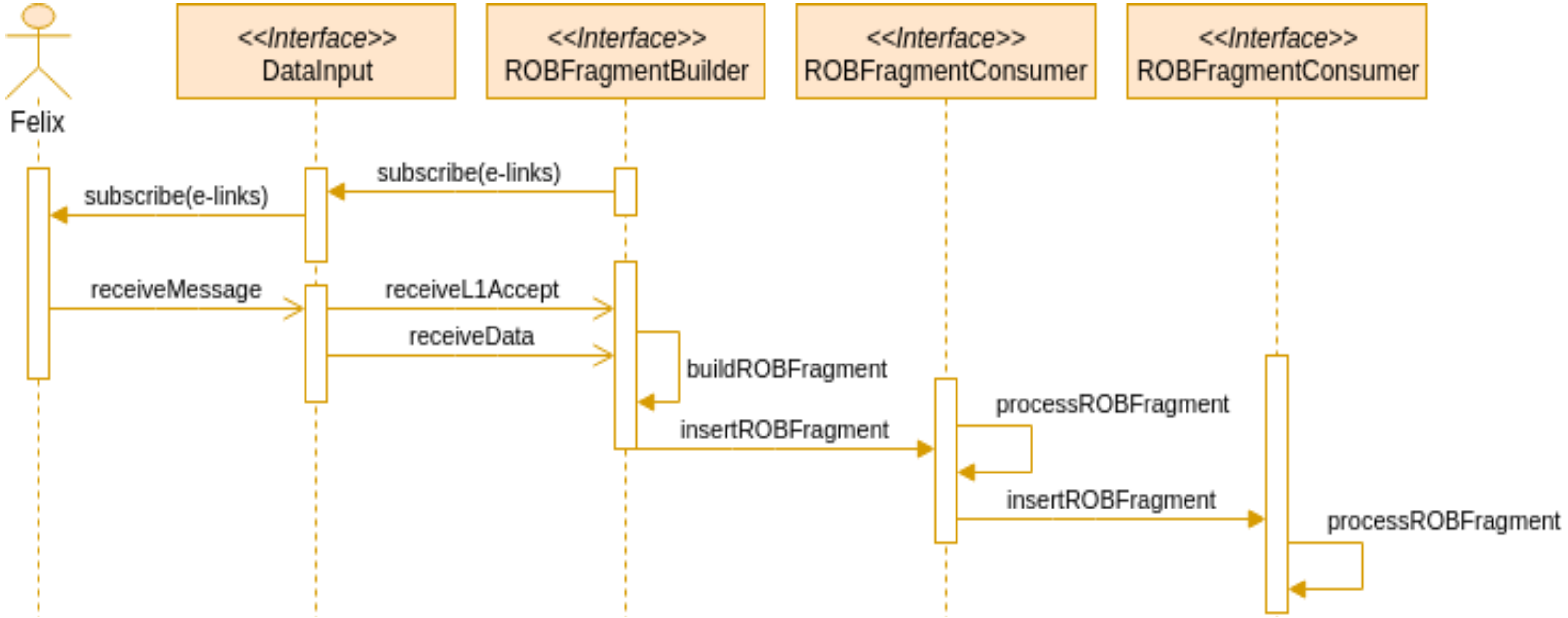
SW ROD Functional Requirements

- Receive data from FELIX system:
  - Support both GBT and FULL mode readout via FELIX
- Replace legacy ROD component:
  - Support custom data aggregation procedures as specified by detectors
  - Support detector specific input data formats
- Support multiple data handling procedures:
  - Writing to disk for commissioning, calibration, etc.
  - Transfer to HLT for normal data taking
  - Etc.



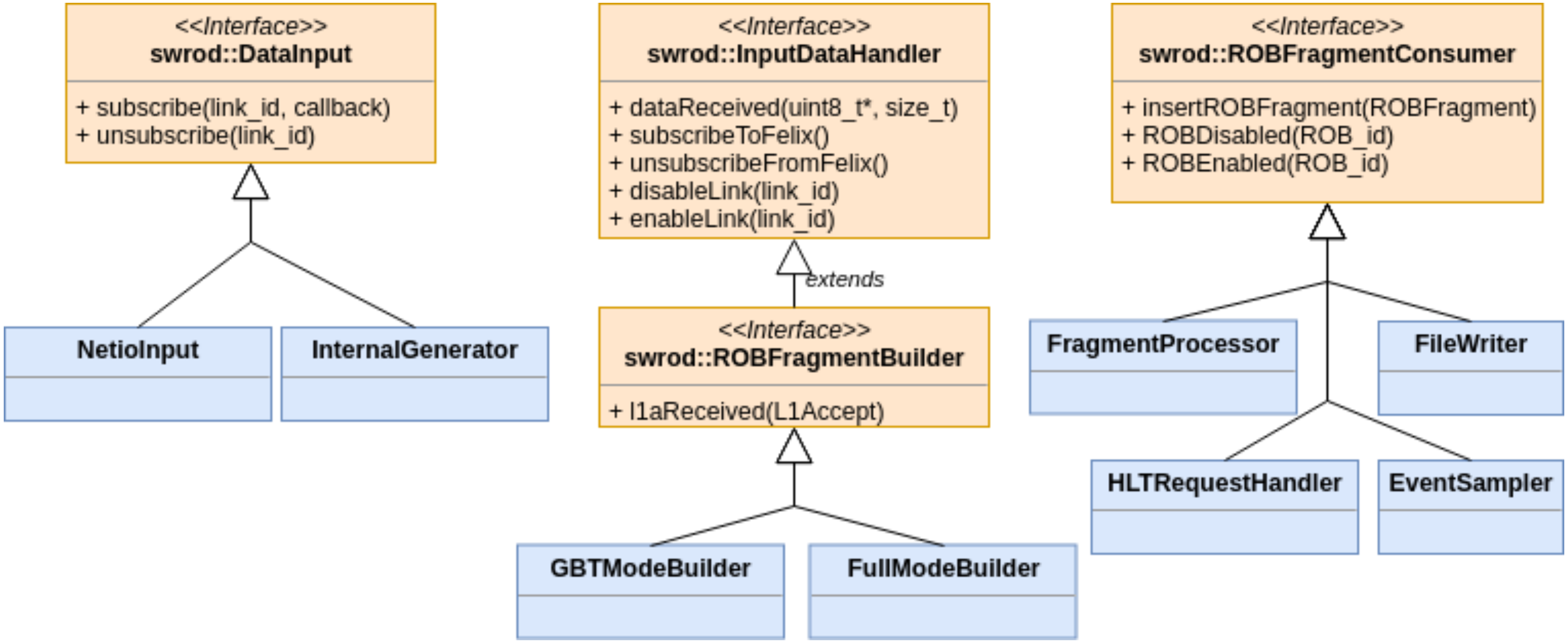
- To address these requirements the SW ROD is designed as a highly customizable framework:
  - Defines several abstract interfaces
  - Internal components interact with one another via these interfaces
  - Interface implementations are loaded dynamically at run-time

SW ROD High-Level Architecture



- ***DataInput*** – abstracts input data source
- ***ROBFragmentBuilder*** – abstracts event fragment aggregation procedures
- ***ROBFragmentConsumer*** – an interface for data processing to be applied to fully aggregated event fragments:
    - Multiple Consumers are organized into a list
    - Each Consumer passes event fragments to the next one in this list

SW ROD Components: Default Implementations



<<Interface>>
swrod::DataInput

+ subscribe(link_id, callback)
+ unsubscribe(link_id)

NetioInput

InternalGenerator

<<Interface>>
swrod::InputDataHandler

+ dataReceived(uint8_t*, size_t)
+ subscribeToFelix()
+ unsubscribeFromFelix()
+ disableLink(link_id)
+ enableLink(link_id)

extends

<<Interface>>
swrod::ROBFragmentBuilder

+ l1aReceived(L1Accept)

GBTModeBuilder

FullModeBuilder

<<Interface>>
swrod::ROBFragmentConsumer

+ insertROBFragment(ROBFragment)
+ ROBDisabled(ROB_id)
+ ROBEnabled(ROB_id)

FragmentProcessor

FileWriter

HLTRequestHandler

EventSampler

- These implementations are provided in the form of a shared library that is loaded by the SW ROD application at run-time
- A custom implementation of any SW ROD interface can be integrated in the same way

SW ROD Performance Requirements

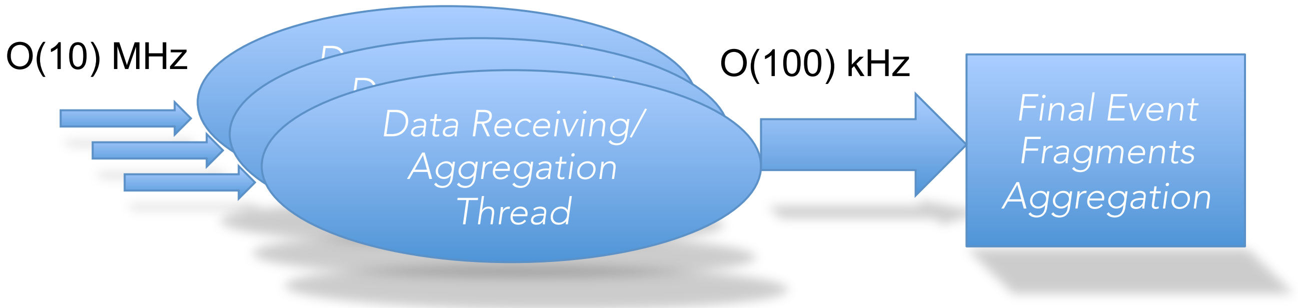
	Chunk Size (B)	Chunk Rate per Link (kHz)	Links per FELIX Card	Chunk Rate per card (MHz)	FELIX Cards per SW ROD	Total Chunk Rate (MHz)	Total Data Rate (GB/s)
GBT Mode	40	100	192	19.2	6	115	4.6
Full Mode	5000	100	12 (24)	1.2 (2.4)	1	1.2 (2.4)	6

- The table contains the worst case requirements
- Data rates are similar for both GBT and FULL modes
- Chunk rate in GBT mode is higher by a factor of 100:
  - Input chunks have to be aggregated into bigger fragments based on their L1 Trigger IDs
  - That represents the main challenge for GBT mode data handling

GBT Mode Performance Challenge

- In average a modern reasonably priced CPU has:
  - **# of cores * core frequency = ~20-30 * 10^9** of CPU cycles
  - Can perform multiple operations per cycle but this is hard to achieve for a complex application:
    - In practice code **operation/cycle >= 1.0** is considered well optimized
- With a total input rate of **115 * 10^6** Hz that would give:
  - **~ 200-300** CPU operations per input chunk
  - Using multiple CPU cores requires a multi-threaded application
  - Passing data between threads at O(100) MHz rate would be practically impossible:
    - Using queues or mutex/conditions will not fit into this budget
- The solution employed by the SW ROD is to assemble input chunks in the data receiving threads

GBT Event Building Algorithm



O(10) MHz

Data Receiving/ Aggregation Thread

O(100) kHz

Final Event Fragments Aggregation

- Input links are split between a configurable number of reading/assembling threads per Data Channel:
  - To scale with the number of input links that varies between detectors
- Each thread builds a fragment of a particular event:
  - Copies input data chunks to a pre-allocated contiguous memory area
  - Happening at O(10) MHz rate
  - No synchronization or data exchange between threads
- Finally the slices are assembled together:
  - Happening at the O(100) kHz rate
  - Implemented with **Intel tbb::concurrent_hash_map**

Amdahl's Law based parallelization formula

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}}$$

S(n) - *the theoretical speedup*

n - *number of CPU cores/ threads*

P - *parallel fraction of the algorithm*

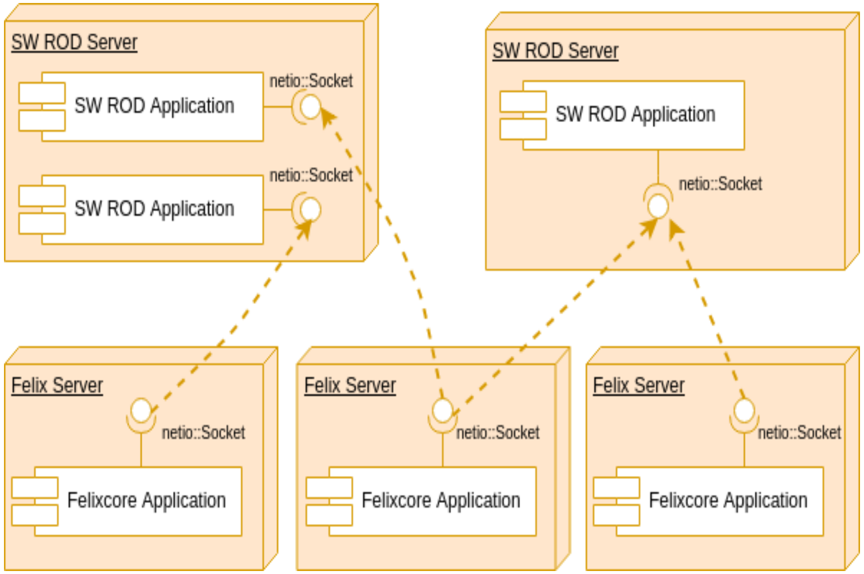
$$\mathbf{P} = 1 - C_{EA} * 10^5/10^7$$
$$= 1 - C_{EA} * 0.01$$

$\mathbf{C_{EA}}$ – relative cost of final event aggregation operation

$\mathbf{C_{EA}} < 10 \quad => \quad P > 0.9$

will offer good algorithm scalability

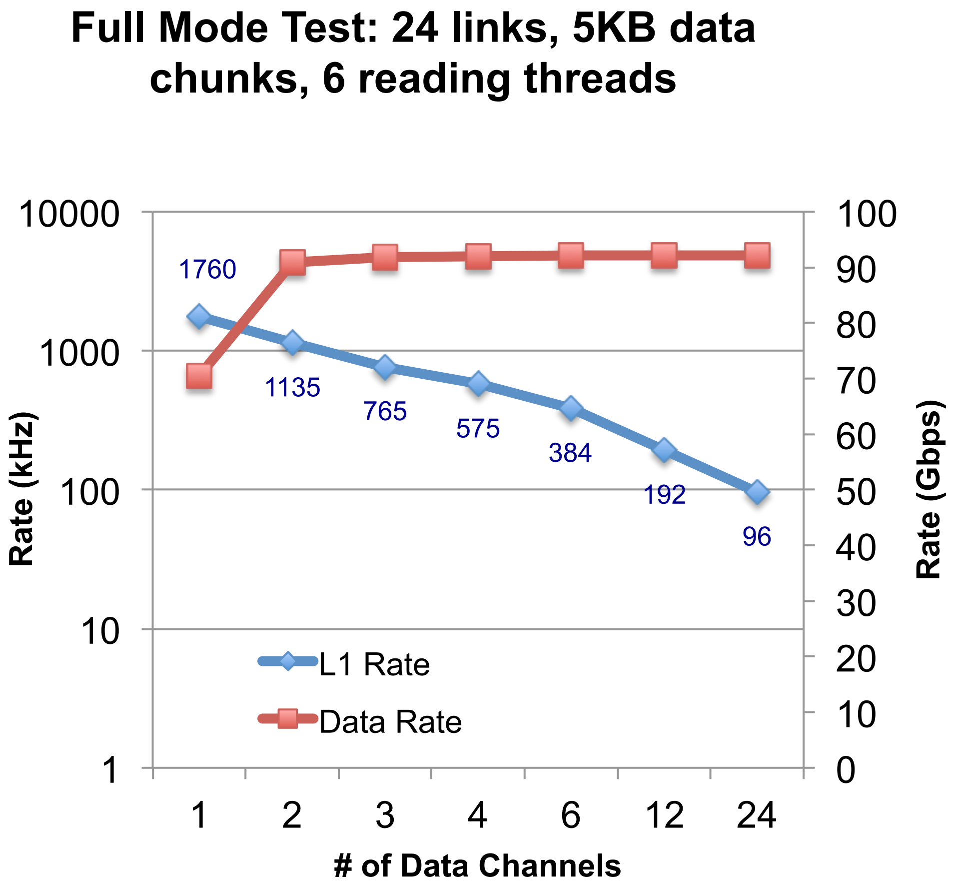
Hardware Configuration for Run 3



- FELIX and SW ROD installation for Run 3 finished recently
- SW ROD Computer:
  - Dual Intel Xeon Gold 5218 CPU @ 2.3 GHz => 16x2 physical cores
  - 96 GB DDR4 2667 MHz memory
  - Mellanox ConnectX-5 100 Gb to FELIX
  - Mellanox ConnectX-4 40 Gb to HLT
- FELIX Computer:
  - Intel Xeon E5-1660 v4 @ 3.2GHz
  - 32 GB DDR4 2667 MHz memory
  - 1 Mellanox network card:
    - ConnectX-5 100 Gb for FULL Mode computers
    - ConnectX-4 25 Gb for GBT mode

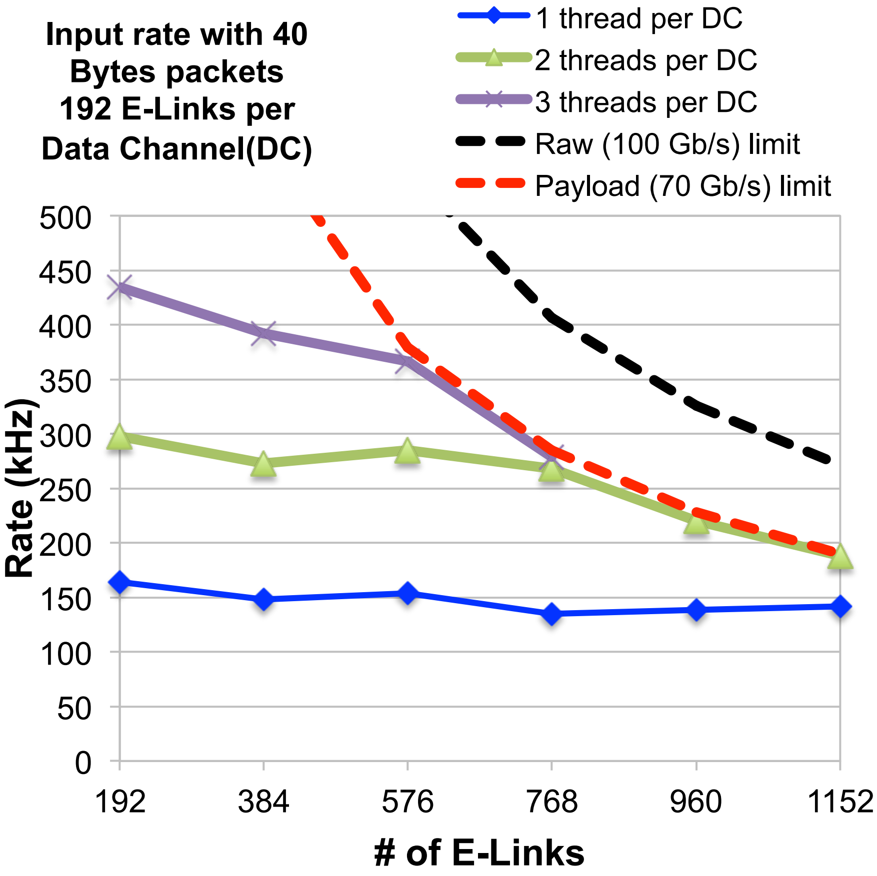
- Such a setup has been used for the performance measurements presented in the following slides:
  - **Netio** is a FELIX software network communication protocol built on top of Remote Direct Memory Access (RDMA)
  - RDMA does not use kernel interrupts and makes it possible to pass data from the network card directly to user process memory

Full Mode Performance Results



- Data Channel – is a single logical data input from detector Front-End:
  - Data packets for the same data channel can be distributed over multiple optical links of the FELIX card
- For all tests except the first one the rate is limited by the network bandwidth:
  - The communication protocol overhead for large data chunks is marginal

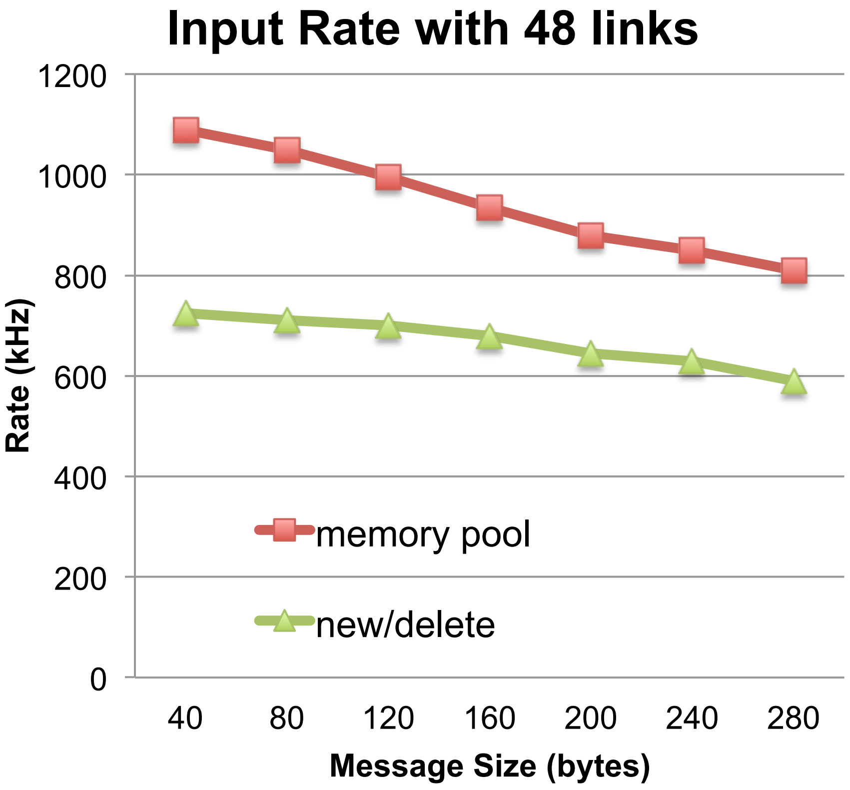
GBT Mode Algorithm Performance



- Can sustain ~150 kHz input rate for the input from 6 FELIX cards:
  - 6 * 192 = 1152 E-Links
- Can be further improved by optimizing the network protocol:
  - The overhead is ~ 40% for 40B data chunks
- Scales very well with the number of threads/cores:
  - $\mathbf{C_{EA}} \approx 7$, $\mathbf{P} \approx 0.93$

# of E-Links	**192**		**384**	
# of threads	Rate	Efficiency	Rate	Efficiency
1	164	1	148	1
2	298	1.82	273	1.84
3	435	2.65	392	2.65

SW ROD Scalability towards Run 4



- In GBT mode 1 MHz rate can be achieved for a small number of input links:
  - Rates are CPU-limited
  - Something that had almost no impact at 100 kHz becomes critical at 1 MHz
  - E.g. memory management adds significant overhead
- Memory Pool implementation was used in place of new/delete:
  - Uses **tbb::concurrent_queue** for handling pre-allocated memory chunks
  - This gives ~40% performance improvement
- Other possible optimizations are being studied

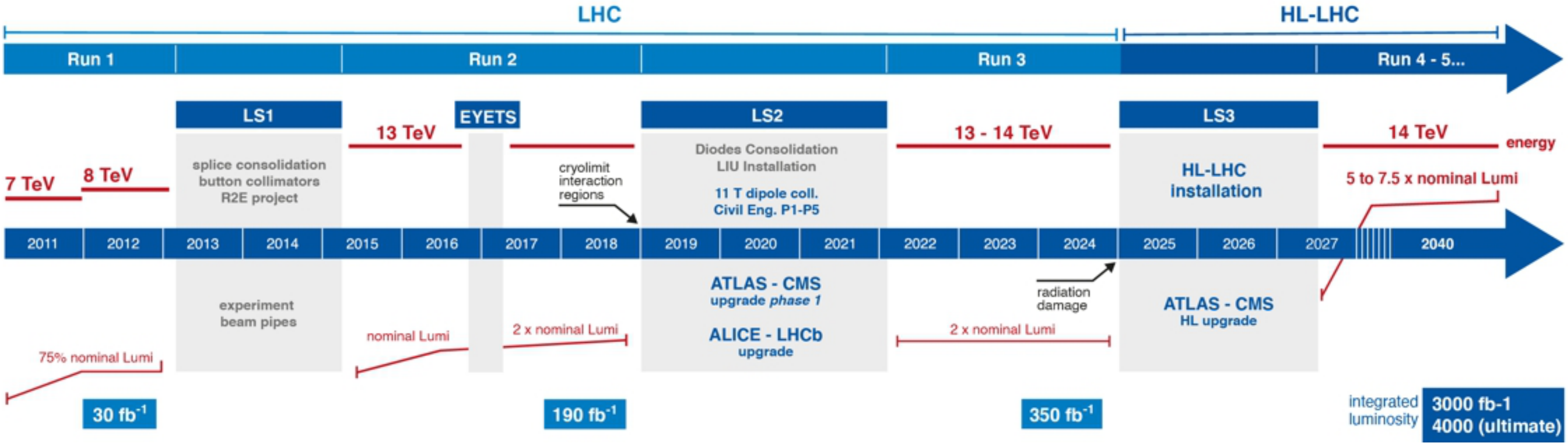
Summary

- A mixture of the legacy ROD-based and the new FELIX-based readouts will be used by ATLAS for the LHC Run 3
- SW ROD is a new component of the ATLAS DAQ system that will be used to receive data from the FELIX readout interface
- SW ROD provides a high performance framework that supports:
  - Custom input data format
  - Custom event building algorithms
  - Custom event processing
- New FELIX based Readout paths have been mostly installed at the ATLAS experimental area
- A fully functional SW ROD implementation is ready for Run 3:
  - Fully satisfies performance and functional requirements
- A study of how Run 4 performance requirements can be met is ongoing

Backup

LHC Evolution Timeline

https://project-hl-lhc-industry.web.cern.ch/content/project-schedule



Data Receiving Thread Optimization Example

- Each data receiving thread operates at O(10) MHz data chunk rate:
  - Even a trivial code modification can affect performance
- An example of the optimizations applied:
  - To get an appropriate fragment from the cyclic buffer the algorithm used input chunks counter in a usual way:

```
int buffer_pos = chunk_counter % buffer_size
```

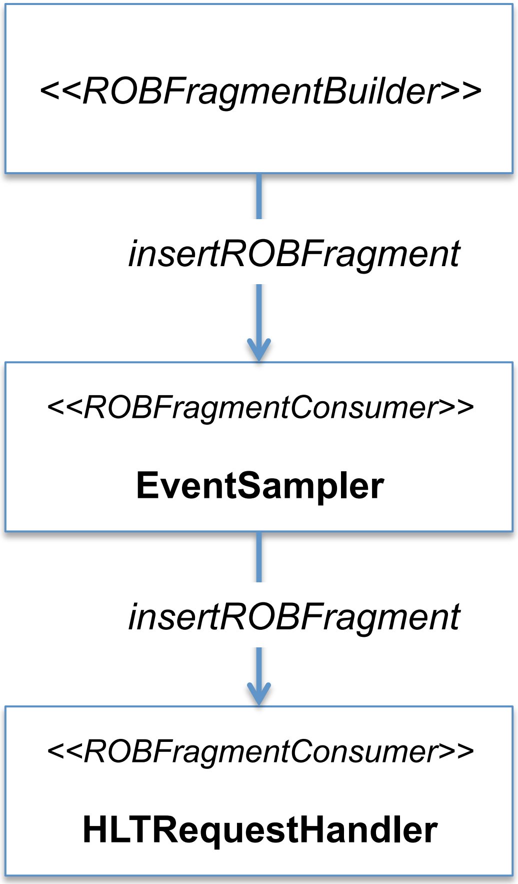
  - If a buffer size is set to 2^n then this can be replaced with:

```
int buffer_pos = chunk_counter & (buffer_size - 1)
```

  - Applying this change gave 10% of overall performance gain

ROBFragmentConsumer Interface Performance Optimization (1/2)

- Uses push-style asynchronous communication model:
  - When a new Fragment is ready Fragment Builder pushes it to the first consumer
- Multiple Consumers are organized into a list:
  - Each consumer in the list forwards fragments to the next one
- A scalable default implementation is provided:
  - *insertROBFragment()* function pushes event to the **Intel tbb::concurrent_queue:**
    - Very fast operation which minimizes impact on the fragment supplier
    - If the queue is full that exerts back-pressure, which is a required behavior
  - A configurable number of threads retrieve fragments from this queue and apply specific processing



ROBFragmentConsumer Interface Performance Optimization (2/2)

```
insertROBFragment(ROBFragment & f){
  m_queue.push(f);
  if (m_next) {
    m_next->insertROBFragment(f);
  }
}
```

```
// the next consumer
m_next(std::bind(&insertROBFragment,
      next, std::placeholders::_1);
// the last consumer
m_next([](){});

insertROBFragment(ROBFragment & f){
  m_queue.push(f);
  m_next(f);
}
```

- The first implementation suffered 20% performance loss for 2 consumers in the list:
  - CPU branch prediction was confused as **`if (m_next)`** statement chooses different code branches with 50% probability
- Using **std::function** object fixes performance:
  - More instructions to be executed
  - But no branch prediction problem