Minimizing CPU Utilization Requirements to Monitor an ATLAS Data Transfer System

Georgios Leventis - CERN
Hello!

I AM GEORGIOS LEVENTIS

Technical Student at CERN

geoleven@outlook.com
ATLAS Experiment at CERN

LHC general purpose detector

- Constant stream of data
- Data reduced by 2 level of triggers
- Currently the readout system uses custom electronics
Current TDAQ
The ATLAS Trigger and Data Acquisition System during Run 2
FELIX is the new readout system for the ATLAS experiment

FELIX routes data coming from the ATLAS detector front-end electronics to the DAQ network

- Servers with custom FPGA-based PCIe cards
- Critical infrastructure:
  - Control of ATLAS detector
  - Data flow
- Every server - system is driven by the FelixCore software application
TDAQ after FELIX integration as foreseen for Run 3
FelixCore

FelixCore is the application that routes the data from the FELIX cards to commodity network.

- Needs to be able to handle up to ~1.5GiB/s or ~5GiB/s of constant data streams (mode depended)
- Needs to be able to monitor the FELIX machine it runs on as these machines are single point of failure components
The problem

Efficiently monitoring *FelixCore*
It is not an easy task

- A single FELIX computer receives up to 40MHz of data fragments
- Routing within FelixCore is a CPU-intensive task
- For efficient routing parallel threads are used
- Statistics from these threads have to be combined to be meaningful
What are we monitoring?

- **Data counter:**
  - Data packet rates
  - Throughput
  - Error rates

- **FELIX variables:**
  - Global buffer memory
  - Thread buffer memory
  - Queue sizes
  - Interrupts
  - Polls

- Etc
For the statistics to be useful, individual thread’s statistics must be:

- Gathered
- Processed / Combined

The data routing threads have to communicate with the statistics thread.
### General

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>154321356</td>
</tr>
<tr>
<td>blocks</td>
<td>300</td>
</tr>
<tr>
<td>chunks</td>
<td>290</td>
</tr>
<tr>
<td>shortchunks</td>
<td>7.1</td>
</tr>
<tr>
<td>block_rate</td>
<td>1.1 MBlocks / s</td>
</tr>
<tr>
<td>chunk_rate</td>
<td>31 MChunks / s</td>
</tr>
<tr>
<td>throughput</td>
<td>1500 Mbyte / s</td>
</tr>
<tr>
<td>max_queue_size</td>
<td>110 kBlocks</td>
</tr>
<tr>
<td>avg_queue_size</td>
<td>44 kBlocks</td>
</tr>
</tbody>
</table>

### Card

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks_free 0</td>
<td>1 MBlocks</td>
</tr>
<tr>
<td>blocks_free 1</td>
<td>1 MBlocks</td>
</tr>
<tr>
<td>blocks_free 2</td>
<td>1 MBlocks</td>
</tr>
<tr>
<td>blocks_free 3</td>
<td>1 MBlocks</td>
</tr>
</tbody>
</table>

### Errors

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>malformedSubchunks</td>
<td>0 Chunks</td>
</tr>
<tr>
<td>malformedChunks</td>
<td>0 Chunks</td>
</tr>
<tr>
<td>malformedBlocks</td>
<td>0 Blocks</td>
</tr>
<tr>
<td>size_error_chunks</td>
<td>0 Chunks</td>
</tr>
<tr>
<td>error_chunks</td>
<td>0 Chunks</td>
</tr>
<tr>
<td>truncated_chunks</td>
<td>0 Chunks</td>
</tr>
</tbody>
</table>
THE IMPACT

Extracting statistics has ~85% performance hit!
The initial implementation

- **Data routing threads:**
  - Push their individual monitoring data to concurrent queues
  - Pushing at a rate multiple to the data rate

- **Statistics thread:**
  - Retrieves monitoring data from the queues in set intervals
  - Combines the data
2

The solution

Atomics
Efficiently synchronizing data from the routing threads

The new statistics module:

- Hardware supported (x86/amd64)

**Atomic Operations**

- Atomic variables
Three implementations

1) Central Atomics

A set of atomic variables accessible from all the parallel routing threads.
1) Central Atomics

CPU Core - Data Routing Thread 1
CPU Core - Data Routing Thread 2
CPU Core - Data Routing Thread 3
CPU Core - Data Routing Thread 4
CPU Core - Data Routing Thread 5
CPU Core - Data Routing Thread 6
CPU Core - Data Routing Thread 7
CPU Core - Data Routing Thread 8

Main Memory

Statistics Set of Atomics
## Three implementations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1) Central Atomics</strong></td>
<td><strong>2) Separate Atomics - Push Config</strong></td>
</tr>
<tr>
<td>A set of atomic variables accessible from <strong>all</strong> the parallel routing threads.</td>
<td>A set of atomic variables for <strong>each</strong> thread. The accumulated values are held by the statistics thread. The routing threads push their own set to the statistics one.</td>
</tr>
</tbody>
</table>
2) Separate Atomics
- Push Config
2) Separate Atomics
- Push Config

Expected
2) Separate Atomics
- Push Config

Expected
2) Separate Atomics - Push Config

Expected
2) Separate Atomics - Push Config

Worst Case Scenario
Three implementations

1) Central Atomics

A set of atomic variables accessible from all the parallel routing threads.

2) Separate Atomics - Push Config

A set of atomic variables for each thread. The accumulated values are held by the statistics thread.
The routing threads push their own set to the statistics one.

3) Separate Atomics - Pull Config

A set of atomic variables for each thread. The accumulated values are held by the statistics thread.
The statistics thread pulls the partial sets from the routing threads.
3) Separate Atomics
- Pull Config
3) Separate Atomics - Pull Config

Always executed serially
3) Separate Atomics
- Pull Config

Always executed serially
3) Separate Atomics
- Pull Config

Always executed serially
3) Separate Atomics
- Pull Config

Always executed serially
The results of the three implementations

1) Central Atomics
   Negligible performance gains ( < 5% ).

2) Separate Atomics - Push Config
   ~400% of performance compared to the initial statistics module. 60% to 70% of target performance.

3) Separate Atomics - Pull Config
   ~500% of performance compared to the initial statistics module.
$\sim 5x$ the initial throughput performance.
Achievable throughput relative to no statistics

Comparison of the final implementation to the initial one
Conclusions

Parallel processing: Concurrency optimization matters
Cache invalidation

Measurements through Intel® VTune™ Amplifier suggested that the “1) Central Atomics” implementation was suffering from performance issues which were manifested as cache invalidation.
What we did and what we learned.

**Result**
The performance gains were enough for us to meet our internal target.

**Lesson**
Change on the concurrency while using atomics could yield totally different results.
Summary

~500%

FELIX data throughput performance gain when using the final statistics module

Concurrency Levels

A significant difference in results
Special thanks to all the people who wrote the original software and helped with my work:

- Jörn Schumacher
- Mark Dönszelmann