Prepared for submission to JINST

Minimizing CPU Utilization Requirements to Monitor an ATLAS Data Transfer System

Georgios Leventis,[a] **Jörn Schumacher,**[a] **Mark Dönszelmann**[b]
[a] *CERN,*
Espl. des Particules 1, 1211 Meyrin, Switzerland
[b] *Radboud University,*
Houtlaan 4, 6525 XZ Nijmegen, Netherlands

E-mail: geoleven@outlook.com

Abstract: The ATLAS experiment at LHC will use a PC-based read-out component called FELIX to connect its front-end electronics to the Data Acquisition System. FELIX translates custom front-end protocols to Ethernet and vice versa. Currently, FELIX makes use of parallel multi-threading to achieve the data rate requirements. In order to establish the FELIX operation conditions, monitoring of its parameters is necessary. This includes, but is not limited to, data counters and rates as well as compute resource utilisation. However, for these statistics to be of practical use, the parallel threads are required to intercommunicate. The FELIX monitoring implementation prior to this research utilized thread-safe queues to which data was pushed from the parallel threads. A central thread would extract and combine the queue contents. Enabling statistics would deteriorate the throughput to less than a fifth of the baseline performance. To minimize this performance hit to the greatest extent, we take advantage of the CPU's microarchitecture features and reduce concurrency. The focus is on hardware-supported atomic operations. When a thread performs an atomic operation, the other threads see it as happening instantaneously. They are used to complement and/or replace parallel computing lock mechanisms. The aforementioned queue system gets replaced with sets of C/C++ atomic variables and corresponding atomic functions, hereinafter referred to as atomics. Three implementations are tested. Implementation I has one set of atomic variables being updated by all the parallel threads. Implementation II has a set of atomic variables for every thread. These sets are periodically accumulated by a central thread. Implementation III is the same as implementation II, but appropriate measures are taken to eliminate any concurrency implications. The compiler used during the measurements is GCC, which supports the hardware (microarchitecture) optimizations for atomics. Implementations I and II resulted in negligible differences compared to the original one. Some benchmarks even show deterioration of the performance. Implementation III (concurrency & cache optimized) yields results with a performance improvement of up to six-fold increase compared to the original implementation. Achieved throughput is significantly closer to what is desirable. Similar structured software applications could benefit from the results of this research, especially Implementation III. The results presented demonstrate that atomics can be useful for efficient computations in a multi-threaded environment. However, from the results, it is clear that concurrency, cache invalidation and proper usage of the system's microarchitecture needs to be taken into account in this programming model. The paper details the challenges of properly using atomics and how they are overcome in the implementation of the FELIX monitoring system.

Keywords: Large detector-systems performance, Software architectures, Detector control systems, Data compression

ATL-DAQ-PROC-2019-021
05 October 2019

Contents

1 Introduction

At the Large Hadron Collider (LHC) at CERN there are four major experiments, the biggest of which is ATLAS [1]. ATLAS is the largest detector by volume ever constructed for a particle collider. It is located in a cavern 100 m below ground. The detector generates about 1.6 MB per event at a rate of 40 million events per second, and the trigger system reduces these data to a manageable amount.

In ATLAS the TDAQ project has primary responsibility for Level 1 Triggers, Data Acquisition System (DAQ) and High Level Trigger (HLT) infrastructure [2].

The Front-End Link eXchange (FELIX) [3] system is a proposed solution for the interface between the data acquisition, detector control and TTC (Timing, Trigger and Control) systems and new or updated trigger and detector front-end electronics as shown in Figure 1. Its job is to route data from the front-end electronics of the ATLAS detector to ATLAS TDAQ commodity network. FELIX is a server-based system. Each server is a commodity PC that hosts two custom FPGA PCIe cards to be able to communicate with the front-end electronics of ATLAS. These cards are named FLX.

The custom firmware of the FLX cards supports two modes of operation: GBT mode and FULL mode [3]. GBT mode uses virtual links to the detector which are called E-Links. This paper focuses on GBT mode. This mode is the most demanding from the monitoring point of view since it implies much higher input frame rate than the FULL mode.

FelixCore is the software application that routes the data from the FLX PCIe cards to the commodity network. The problem is that the original implementation of the statistics module at the FelixCore application introduces a significant performance reduction of about ~85% in GBT mode.

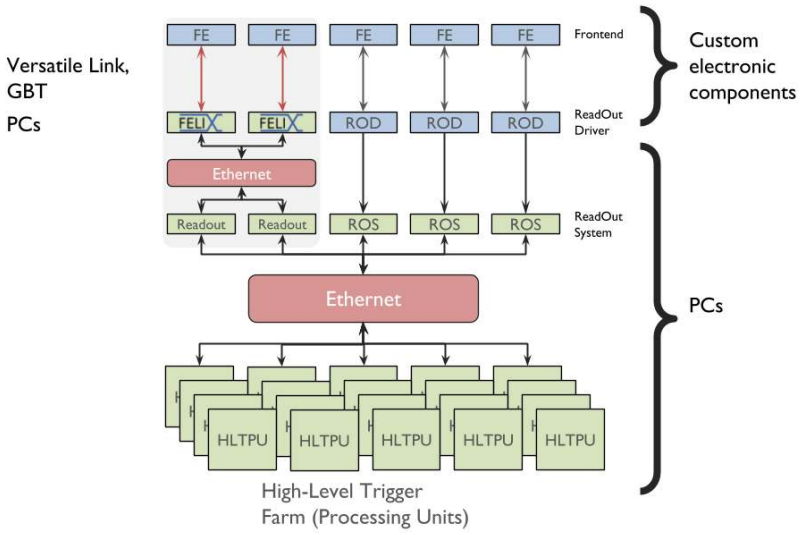

Figure 1. Schematic of the placement of the FELIX system. The gray area represents a subset of detectors for which FELIX will be used for[3].

This paper describes the original implementation with its problems along with the three different implementations which have been exercised to mitigate these problems.

2 Background

2.1 Setup

The presented results were obtained on a FELIX candidate system (hereafter referenced as the server) whose specifications can be seen in Table 1.

The server also houses two FLX cards. Each card provides two logical PCIe endpoints (logical FLX cards) on the server. As such, in total there are four logical FLX logical cards listed by the operating system of the server.

Motherboard	Supermicro X10SRW-F 1.02B
CPU	Intel®Xeon®CPU E5-1660 v4 @ 3.20 GHz
RAM	4x8GB DDR4 Registered 2400 MHz Samsung M393A1K43BB0-CRC
SSD	Intel 240 Gb SSDSC2KB240G8
Ethernet	Mellanox Technologies MT27800 Family (25 Gbps)
O.S.	CentOS 7 (CERN Version)

Table 1. Specifications of the FELIX server

2.2 Measurement methods

The key performance metric for a FELIX system is data throughput. The throughput describes the amount of data per unit time which the FELIX system can route from the physical links connecting the detector to the commodity network.

Each FELIX server will have data throughput as high as 1536 MB/s. It can be calculated as follows:

- 2 physical FLX cards,
- 2 logical FLX cards (PCIe endpoints) per physical FLX card,
- 12 links per logical FLX card,
- up to 8 E-Links per link,
- up to 40 bytes of data message size per E-Link,
- up to 100 kHz of data messages per E-Link.

Therefore the total amount is: $2 \times 2 \times 12 \times 8 \times 40\,\text{bytes} \times 100\,\text{kHz} = 1536\,\text{MB/s}$

The throughput is measured in two ways. One is by measuring the data rate at which a client is receiving data from FelixCore, if said client is subscribed to all available E-Links on one FelixCore instance. The second is to get the measurement of the throughput from the statistics of FelixCore itself.

Various performance metrics like microarchitecture usage, cache invalidations, CPU utilization and time consumed by functions are obtained through the Intel® VTune™ Amplifier software suite (hereinafter referred to as "VTune"). This method is mainly used to identify the bottlenecks of our codebase.

2.3 FelixCore software application

FelixCore makes use of multiple parallel threads which are used to acquire (read) the data coming from the FLX cards and then push them to the network stack. Each data routing thread process only its own set of links or E-Links. There is no need for these threads to intercommunicate.

To obtain the baseline performance, the performance of the FelixCore application was measured with the statistics module disabled. By inspecting the code we come to the conclusion that the presence of the disabled statistics module has no impact on performance.

The results of the baseline measurement show that the application is able to handle a data throughput of up to ~1.48 GB/s. This throughput is below the maximum amount of data expected from the detector. Optimizations, not related to statistics, are expected to be applied. As they are beyond the scope of this paper, we focus on achieving as close to the baseline rate as possible.

2.4 Original implementation

In the original, non-optimized version, there is a dedicated statistics thread which maintains a concurrent queue for every statistics metric that is gathered. The routing threads enqueue values to these queues while processing data as seen in Figure 2. The concurrent queues used are moodycamel's ConcurrentQueue [4].

The statistics thread has a loop which operates at regular configurable intervals. Part of its operation is to check the queues for values, dequeue and sum them up as needed. Thread safety and concurrency are managed by the concurrent queues as implied by their name.

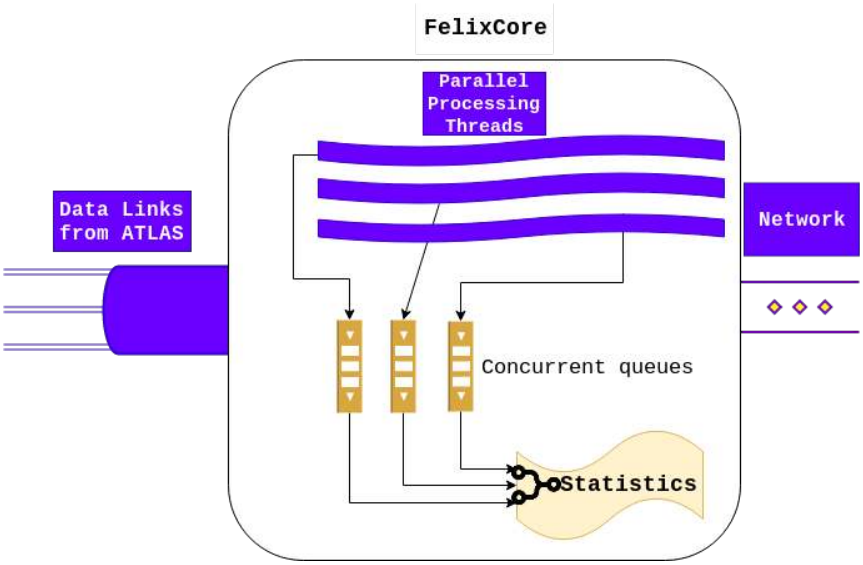

Figure 2. Original implementation

This original implementation works well for a small amount of data passing through the FelixCore application. However the synchronization, which is added to the data handling threads by the statistics module causes a major hit to the FelixCore data handling performance. The peak throughput of the FelixCore application decreases to about ~0.23 GB/s which corresponds to ~85% performance loss.

3 Optimization Methods

3.1 Atomic Variables & Atomic Operations

The approach of reducing the performance hit of the statistics module is based on the use of atomic variables and operations in place of the concurrent queues. Recent Intel® CPUs take advantage of underlying hardware features [7] to achieve synchronization tasks more efficiently. Atomic variables and their corresponding operations are usually described as a way to avoid mutual exclusion. They have a limited set of functionalities, but they are faster than locks and also lack deadlock and convoying problems. [8]

3.2 Implementation I

In implementation I, the concurrent queues are completely removed. The statistics thread creates a set of atomic variables as seen in Figure 3. The atomic counters include (but are not limited to): data packet / chunk counters, error counters, ports used, message sizes and blocks per threads.

The routing threads have a reference to the structure of the atomic variables. Each time a routing thread needs to update a certain statistics value, it does so by directly modifying the corresponding atomic variable with an atomic operation.

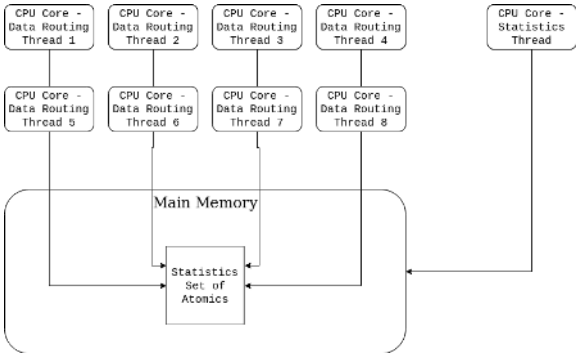

Figure 3. Implementation I, the threads running and the statistics' set of atomic variables in memory

Measuring performance of this implementation yields results similar to the original implementation with no significant increase to the performance. This, alongside major concurrency issues observed through VTune, was the key factor which led to the creation of Implementation II.

3.3 Implementation II

In implementation II, each data routing thread uses a separate set of atomic variables. The statistics thread also uses a different set. The data routing threads update the values of their own sets as they route data. This is done so they do not have to operate on a shared set and as such removing the concurrency issues of implementation I. Each routing thread creates an update thread which at programmatically configurable intervals, tries to add the statistics data of its creator thread to the statistics data held by the statistics thread, using atomic operations.

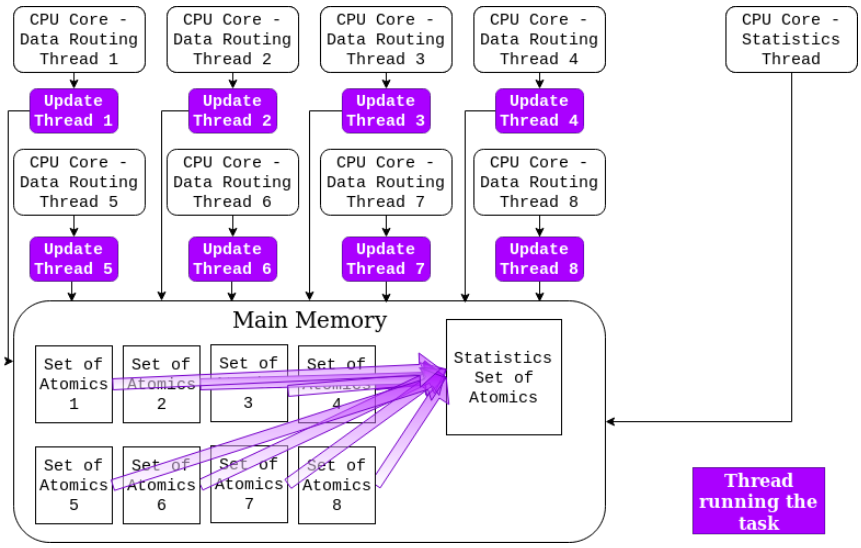

Figure 4. Implementation II, showing the possibility of all update threads trying to push their statistics simultaneously to the statistics thread.

Measuring the peak throughput for this implementation gives the result of ~0.87 GB/s for a 0.1 s statistics update interval. This is an improvement compared to the original implementation

but it is still only ~60% of the target rate. Analysis from VTune shows that there still are some concurrency issues which were often manifested as cache invalidation problems. The data routing threads, due to their synchronous start, also have a great probability of trying to push their values simultaneously to the statistics thread, which makes them stall while waiting for the others to finish, as seen in Figure 4.

3.4 Implementation III

The design of implementation III, resembles that of implementation II. The structure of the atomic variables is exactly the same. The way the statistics data are gathered from the data routing threads is altered. Instead of the data routing threads attempting to push their data to the statistics thread, which can lead to possible concurrency issues, the statistics thread itself retrieves them serially, with no possibility of this happening in a concurrent way.

The measured throughput of this implementation is ~1.18 GB/s. This is a five-fold improvement over the original implementation and as such a general improvement over all previous implementations.

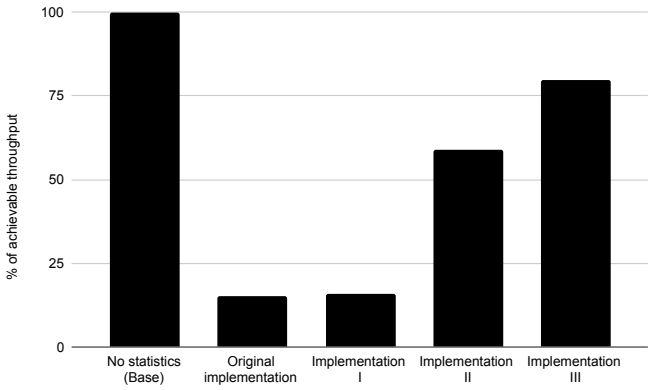

Figure 5. Comparison of the throughput achievable by FelixCore with no statistics, the original implementation and the three implementations described within this paper.

4 Conclusion

To sum up, our research concluded with a desirable result. The performance hit of implementation III is only ~20% compared to not having statistics. It is also ~5 times the performance of the original implementation as seen in Figure 5. Further future work on the rest of the application is required to achieve the target rate of 1.5 GB/s which is needed by the ATLAS experiment at LHC while having the statistics module enabled. The current results get us closer to this goal.

References

[1] ATLAS Collaboration, The ATLAS Experiment at the CERN Large Hadron Collider, JINST, 3, 2008, S08003

[2] ATLAS TDAQ Collabration, The ATLAS Data Acquisition and High Level Trigger system, 2016 JINST 11 P06008

[3] ATLAS Collaboration, Technical Design Report for the Phase-I Upgrade of the ATLAS TDAQ System, CERN-LHCC-2013-018. ATLAS-TDR-023, 2013, https://cds.cern.ch/record/1602235,

[4] https://github.com/cameron314/concurrentqueue

[5] https://software.intel.com/en-us/vtune-amplifier-help-microarchitecture-usage

[6] https://software.intel.com/en-us/vtune-amplifier-help-cpu-utilization

[7] Lightweight contention management for efficient compare-and-swap operations, Dice, David and Hendler, Danny and Mirsky, Ilya, European Conference on Parallel Processing, 2013, Springer, https://arxiv.org/abs/1305.5800

[8] https://software.intel.com/en-us/node/506090