January 31, 2002

# HLT Validation of Athena

**M. Bosman, K. Karr**
IFAE, Barcelona

**C. Bee**
CPPM, Marseille

**S. Gonzalez**[1], **W. Wiedenmann**
University of Wisconsin

**Summary**

In the present view, the ATLAS High Level Trigger will base its event selection software on the offline reconstruction framework, Athena. It is therefore imperative that the offline software – and its relevant components – are able to handle the large CPU and bandwidth loads required in a real-time environment. This note presents a first set of measurements aimed at validating Athena as the ATLAS online event selection framework. Although Athena is at an early development stage, detailed profiling can already yield clues as to which components can be optimized. In this note such areas are identified and a proposal is made on a road map to full performance.

ATL-DAQ-2002-005
12/02/2002

## 1 Introduction

The ATLAS High Level Trigger [1] will use reconstruction algorithms developed by the offline community in its Event Filter selection stage [6]. This requirement minimizes duplication of work and ensures consistency between the offline and the online event selections.

The re-use of offline algorithms in the online environment implies that the online framework, where the algorithms reside, must provide the same interfaces and services as the offline framework. Furthermore, the demanding online environment at the LHC places severe constraints on the performance of the event selection software.

This note presents a first study of the suitability of Athena [3] as the event selection framework for the ATLAS High Level Trigger (HLT), addressing the use of Athena at the Event Filter (EF) only. Re-using Athena code in the Level-2 trigger is a possibility that will be studied once Athena is validated as an EF framework.

In addition to the validation activities, an important aim of this work is to establish reference points and metrics that can be used to monitor the performance of Athena as it evolves. This is especially important given that Athena is in full development.

It is important to remember that the framework choice influences the programming model (i.e., architecture) used for the HLT software. For example, the separation of data and algorithms in Athena requires an "intelligent" Event Data Model (EDM), which may have performance consequences because algorithms are forced to communicate via a Transient

[1]Contact: Saul.Gonzalez@cern.ch

Event Store (TES). The HLT software Design Document [4] already foresees a TES-based framework.

The evaluation of alternatives to Athena is beyond the scope of this note.

# 2 Athena as an Online Event Selection Framework

## 2.1 Scope of Validation

Both functional and performance aspects need to be considered when evaluating Athena as an HLT candidate framework. Since the ATLAS offline software is in an early development stage, it is impossible to conclude today whether Athena is a reasonable basis for the HLT software or not. However, it is essential to evaluate Athena now with an HLT point-of-view, so that weaknesses and problems can be identified early.

The functional aspects of Athena must be considered in the context of the HLT software requirements [6, 7]. Although it is too early for such a functional evaluation, the validation of Athena must include an appraisal of its capabilities as a trigger framework.

In addition to fulfilling the functional requirements – and since the online software is a mission-critical application – the Athena framework must be designed and use core software components that are fast, flexible, and reliable.

## 2.2 Caveats on Athena Validation

The HLT requirements will eventually translate into a detailed architectural design of the trigger software. Until such a design is available, only preliminary evaluations of Athena can be made. This note is based on a very simple "computing model" of the Event Filter. This model, shown in Figure 1, incorporates the basic elements of event selection at the Event Filter:

- Reconstruction algorithms: imported from the offline and including, e.g., steering, calorimetry, tracking.
- Dataflow and data model: a critical component since it controls how much and in what way data is organized and accessed.
- Raw data conversion: conversion of event data from raw Event Builder (EB) format to a format that algorithms understand (using the EDM).
- Meta-data handling: includes "low-frequency" components like configuration, calibration, monitoring, and run control.

However, presently all of these components are either under development or in a prototype stage. In this note we concentrate on the more critical "high-load" components, including algorithms, data flow, and raw data conversion.

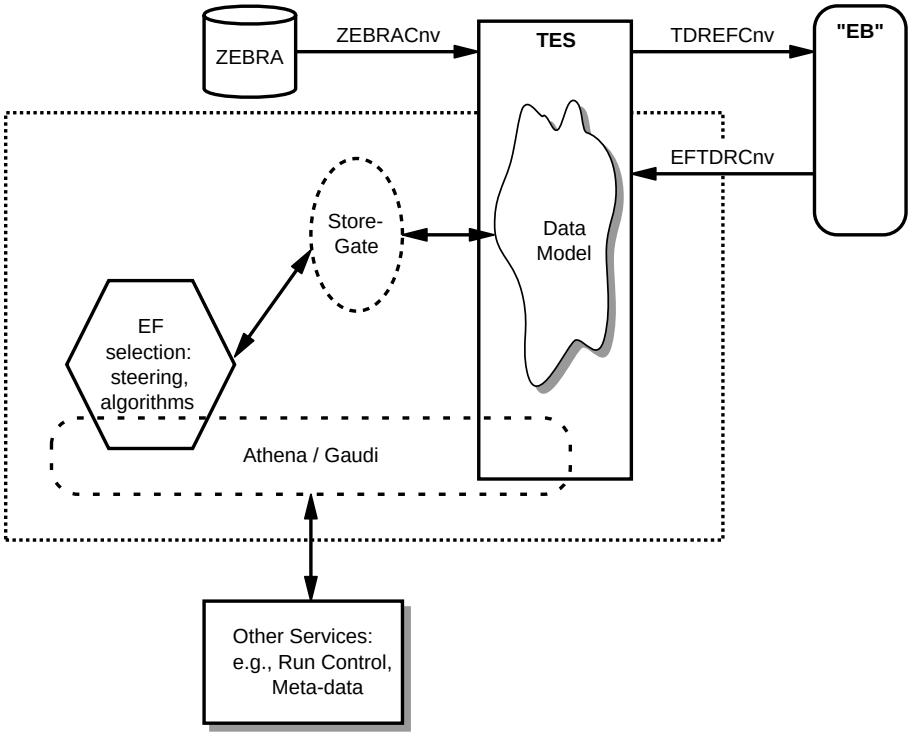



Figure 1: A validation perspective of Event Filter components. The domain of the EF considered here falls within the dotted lines.

## 3 Performance Measurements

As mentioned in Section 1, one of the aims of measuring the performance of the software is to provide a "first data-point" in order to gauge whether development of Athena components that are relevant for the HLT is headed in the right direction. Many measurements of ATRECON-based HLT software exist (see [1] and references therein). However, performance measurements of Athena and Athena-related components must be brought – at least – to the same level as presented in the HLT/DAQ/DCS Technical Proposal [1].

Performance measurements of the HLT framework should include, for example, measurements of:

- Execution time
- Memory utilization
- I/O requirements

Ideally, the above should be measured using a representative data sample on a full or a prototype HLT system. The measurements presented in this note are mainly on execution time, although memory utilization issues will be addressed.

A decision on the suitability of Athena as an HLT framework will follow only after the HLT software is mature and its performance is well understood. The consequences of not adopting the offline framework and associated reconstruction algorithms for the Event

| Package | Athena version | Gaudi version |
|---|---|---|
| EFTDRCnv | 2.0.2 | 0.7.4 |
| RD Event | 2.0.2 | 0.7.4 |
| XKalman+Calo. | 1.3.5 | 0.7.2 |

Table 1: ATLAS release versions used in TAU instrumentation.

Filter need to be carefully examined. Adopting a different framework at the HLT would demand much more manpower than presently available, it would increase the complexity of the ATLAS software, and it would slow the transfer of new physics selections to the online system.

### 3.1 Benchmarking tools

In order to measure execution time, the HLT prototype code has been instrumented with the TAU profiling tool [12]. More details on TAU and technical aspects of the instrumentation can be found in Appendix A.

All available components of a prototype EF system (shown in Figure 1) have been instrumented, including:

- The Event Filter byte stream converter [5] (only the Silicon and Pixel detectors were available at the time of this study);
- The event data model ("RD Event");
- All Gaudi base libraries and selected Athena components;
- One full set of algorithms, including XKalman++ and the calorimeter reconstruction.

Since the ATLAS software is developing at a fast rate and since the instrumentation work is very time-consuming, the instrumentation has only been carried out on officially tagged and released ATLAS code. This approach should help ensure reproducibility in benchmarking. Table 1 shows the ATLAS software versions that were instrumented for this study.

### 3.2 Platforms and data sets

While development work was carried out in two different machines, the results shown here were obtained in a 733 MHz Pentium III machine with 512 MB of RAM, running Linux 6.1 (Standard CERN installation). The machine is a standard desktop residing at CERN in building 32. The equivalent SpecInt95's for this machine is approximately 30.

The data sets used for benchmarking Athena were $Z \rightarrow b\overline{b}$ events and jets (electron stream). The latter sample was used in profiling the algorithms. Samples with and without pileup were used.

### 3.3 Profiling Results

In the following, the measurements performed on Athena, and related components, are summarized. Since we are presently most concerned with Athena's steady-state performance in the High Level Trigger framework, all methods that are solely part of the initialization or finalization phase of program execution were excluded from the analyses described in this section.

The absolute execution times presented in this section, even today, do not reflect the best possible performance of the software. The source code has not been optimized for performance and the executables have not been built with optimal compilation flags. However, the scaling behavior and the measurements of the relative performance of the components are not affected by this lack of optimization.

#### 3.3.1 Algorithms (electron-ID)

The electron ID test example of the LAR group [11] was used as an example for a typical algorithm application. Based on Atlas release 1.3.5/1.3.6, it was integrated with StoreGate[2] and included the complete reconstruction chain from the raw data handling to the final identification of electrons. As mentioned above, the code was not run in an optimized form, In addition, due to a memory leak not more than 30 pileup events could be processed. This memory leak should also affect the execution times, so that any total times should be taken with caution.

The algorithm was executed in the following configurations:

- On single electron data;
- On jet data;
- On jet data with pileup;
- On jet data with pileup and a $p_\perp = 15$ GeV cut for the calorimeter and a $p_\perp = 5$ GeV cut for XKalman++;
- On jet data with pileup with a seeded reconstruction from the calorimeter[3].

In the first four cases the reconstruction was attempted for the complete detector; in the last case only in a certain region defined by the seed.

Detailed profiles for each case can be found in Appendix B. It can be observed that typically the execution times are dominated by methods for building the calorimeter information. The contribution to the total execution time from reconstruction algorithms like Xkalman++ is very small in the case of high $p_\perp$ electrons but it dominates for pileup events, where reconstruction in the complete detector (B physics case) was attempted. With increased $p_\perp$ cuts and with seeded reconstruction for events with pileup, the execution times are again dominated by methods for building the calorimeter information.

[2] This is the version referred in Section 3.3.4 as "old StoreGate".

[3] At the time of the test the code to do this was not yet officially released. A private version was used [10].

The total execution times for the algorithm, run in the aforementioned configurations, were: 2.35 sec./event (electrons), 7.35 sec./event (jets), 80 sec./event (jets with pileup), 31 sec./event (jets with pileup, $p_\perp$ cuts), and 24 sec./event (jets with pileup, seeded reconstruction).

### 3.3.2 EFTDREventCnv

The *EFTDREventCnv* package [5] is an Athena service designed to convert packed raw event persistent data, in the format expected from the Event Builder (EB), into a form suitable for access from the TES of Athena. To simulate the input from the EB several samples of packed raw events were produced for both high and low luminosity using another package called *TE2REConverter* (See Figure 1.)

The *EFTDREventCnv* package was instrumented with TAU along with a copy of the Gaudi software. TAU profiling output was produced by running the instrumented package via a simple test algorithm over two different packed raw event data samples: one with 100 low luminosity $Z \rightarrow b\overline{b}$ events and one with 20 high luminosity jet events. For each run the resulting profile output file was used to extract the following information for all steady-state methods of the *EFTDREventCnv* package called during the given run:

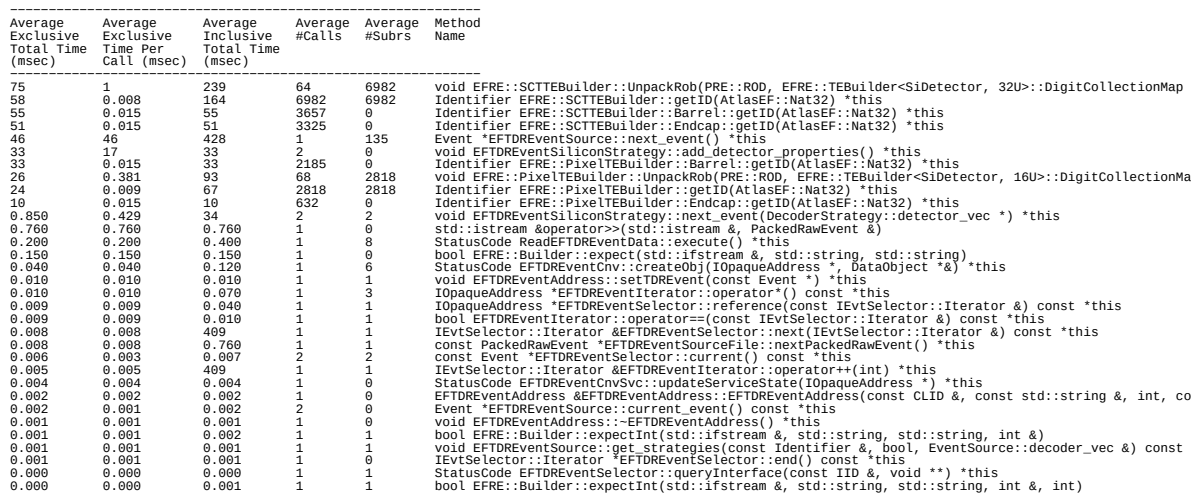
- Average total exclusive time consumed per event (including multiple calls to the method).
- Average exclusive time consumed per call to the method.
- Average total inclusive time consumed per event (includes total exclusive times of all subroutines called by the method).
- Average number of calls to the method.
- Average number of subroutine calls made by the method.

Similar information was obtained for all steady-state methods of the Gaudi framework that were called during the run. Figure 2 shows a sample profiling output from TAU.

Analysis of the profile data is discussed in the following sections.

**3.3.2.1 Low Luminosity Results** The average total real time consumed per event was measured to be approximately 470 ms. The largest contribution to the total exclusive time (231 ms) came from some encapsulated methods of the *EFTDREventCnv* package called *GetID* which are used to extract the offline identifier of a Digit from its online representation. Although the average exclusive times per call for these methods were quite small ($\simeq 23\mu$s) they had to be called for every Digit. The profile output shows that the average number of Digits for the Pixel and SCT detectors in the low luminosity data sample are approximately 2,800 and 7,000, respectively.

The second largest total exclusive time (101 ms) was consumed by methods of the *EFTDREventCnv* package called *UnpackRob*, which performs the following tasks: Loop over each of the event's RODs; For each ROD, loop over all of its Digits; Unpack each Digit



| Average Exclusive Total Time (msec) | Average Exclusive Time Per Call (msec) | Average Inclusive Total Time (msec) | Average #Calls | Average #Subrs | Method Name |
|---|---|---|---|---|---|
| 75 | 1 | 239 | 64 | 6982 | void EFRE::SCTTEBuilder::UnpackRob(PRE::ROD, EFRE::TEBuilder<SiDetector, 32U>::DigitCollectionMap |
| 58 | 0.008 | 164 | 6982 | 6982 | Identifier EFRE::SCTTEBuilder::getID(AtlasEF::Nat32) *this |
| 55 | 0.015 | 55 | 3657 | 0 | Identifier EFRE::SCTTEBuilder::Barrel::getID(AtlasEF::Nat32) *this |
| 51 | 0.015 | 51 | 3325 | 0 | Identifier EFRE::SCTTEBuilder::Endcap::getID(AtlasEF::Nat32) *this |
| 46 | 46 | 428 | 1 | 135 | Event *EFTDREventSource::next_event() *this |
| 33 | 17 | 33 | 2 | 0 | void EFTDREventSiliconStrategy::add_detector_properties() *this |
| 33 | 0.015 | 33 | 2185 | 0 | Identifier EFRE::PixelTEBuilder::Barrel::getID(AtlasEF::Nat32) *this |
| 26 | 0.381 | 93 | 68 | 2818 | void EFRE::PixelTEBuilder::UnpackRob(PRE::ROD, EFRE::TEBuilder<SiDetector, 16U>::DigitCollectionMa |
| 24 | 0.009 | 67 | 2818 | 2818 | Identifier EFRE::PixelTEBuilder::getID(AtlasEF::Nat32) *this |
| 10 | 0.015 | 10 | 632 | 0 | Identifier EFRE::PixelTEBuilder::Endcap::getID(AtlasEF::Nat32) *this |
| 0.850 | 0.429 | 34 | 2 | 2 | void EFTDREventSiliconStrategy::next_event(DecoderStrategy::detector_vec *) *this |
| 0.760 | 0.760 | 0.760 | 1 | 0 | std::istream &operator>>(std::istream &, PackedRawEvent &) |
| 0.200 | 0.200 | 0.400 | 1 | 8 | StatusCode ReadEFTDREventData::execute() *this |
| 0.150 | 0.150 | 0.150 | 1 | 0 | bool EFRE::Builder::expect(std::ifstream &, std::string, std::string) |
| 0.040 | 0.040 | 0.120 | 1 | 6 | StatusCode EFTDREventCnv::createObj(IOpaqueAddress *, DataObject *&) *this |
| 0.010 | 0.010 | 0.010 | 1 | 1 | void EFTDREventAddress::setTDREvent(const Event *) *this |
| 0.010 | 0.010 | 0.070 | 1 | 3 | IOpaqueAddress *EFTDREventIterator::operator*() const *this |
| 0.009 | 0.009 | 0.040 | 1 | 1 | IOpaqueAddress *EFTDREventSelector::reference(const IEvtSelector::Iterator &) const *this |
| 0.009 | 0.009 | 0.010 | 1 | 1 | bool EFTDREventIterator::operator==(const IEvtSelector::Iterator &) const *this |
| 0.008 | 0.008 | 409 | 1 | 1 | IEvtSelector::Iterator &EFTDREventSelector::next(IEvtSelector::Iterator &) const *this |
| 0.008 | 0.008 | 0.760 | 1 | 1 | const PackedRawEvent *EFTDREventSourceFile::nextPackedRawEvent() *this |
| 0.006 | 0.003 | 0.007 | 2 | 2 | const Event *EFTDREventSelector::current() const *this |
| 0.005 | 0.005 | 409 | 1 | 1 | IEvtSelector::Iterator &EFTDREventIterator::operator++(int) *this |
| 0.004 | 0.004 | 0.004 | 1 | 0 | StatusCode EFTDREventCnvSvc::updateServiceState(IOpaqueAddress *) *this |
| 0.002 | 0.002 | 0.002 | 1 | 0 | EFTDREventAddress &EFTDREventAddress::EFTDREventAddress(const CLID &, const std::string &, int, co |
| 0.002 | 0.001 | 0.002 | 2 | 0 | Event *EFTDREventSource::current_event() const *this |
| 0.001 | 0.001 | 0.001 | 1 | 0 | void EFTDREventAddress::~EFTDREventAddress() *this |
| 0.001 | 0.001 | 0.002 | 1 | 1 | bool EFRE::Builder::expectInt(std::ifstream &, std::string, std::string, int &) |
| 0.001 | 0.001 | 0.001 | 1 | 1 | void EFTDREventSource::get_strategies(const Identifier &, bool, EventSource::decoder_vec &) const |
| 0.001 | 0.001 | 0.001 | 1 | 0 | IEvtSelector::Iterator *EFTDREventSelector::end() const *this |
| 0.000 | 0.000 | 0.000 | 1 | 1 | StatusCode EFTDREventSelector::queryInterface(const IID &, void **) *this |
| 0.000 | 0.000 | 0.001 | 1 | 1 | bool EFRE::Builder::expectInt(std::ifstream &, std::string, std::string, int &, int) |

Figure 2: Profiling results for *EFTDREventCnv* for low luminosity $Z \to b\overline{b}$ events.

and 2 data words and get its offline identifier (via a call to *GetID*); For each offline identifier corresponding to a new wafer, a SiDetector object is created; For each Digit, a new SiDigit object is created and added to the SiDetector object. Then a pointer to the latter is inserted into a map using the offline identifyer as its key. Similarly to the *GetID* methods, the average exclusive time per call for the *UnpackRob* methods was relatively small (0.38 ms for Pixel and 1.00 ms for SCT) but they had to be called for an average of 68 RODs for the Pixel detector and 64 RODs for the SCT detector. In addition, for each ROD, the above mentioned operations had to be performed.

The third largest contribution to the total exclusive time (46 ms) came from a single call to the method *EFTDREventSource*::*next_event*. This method is dominated by uninstrumented ATLAS software. Further analysis using time stamps showed that most of the time is consumed by the Event::accept method, which fills the detector hierarchy structure with the information of a given event. This software is part of the old event data model ("RD Event"), which will eventually be replaced.

The time taken to read in the packed raw event from the input stream was measured to be 0.76 ms. The total average time consumed per event by all of the Gaudi methods called was approximately 12 ms, which is less than 3% of the total average event time.

**3.3.2.2 High Luminosity Results** The average total real time consumed per event was measured to be approximately 4900 ms. This is an order of magnitude greater than the average total time consumed per event at low luminosity.

The largest contribution to the total exclusive time (2200 ms) again resulted from the *GetID* methods. This is about 10 times greater than the time consumed in the case of low luminosity, which is consistent with the fact that there are about 10 times more Digits for the pileup date sample: $\simeq 26,000$ for Pixel and $\simeq 64,000$ for SCT. The average exclusive time per call remained the same at $23\mu$s.

This time the second largest contribution to the total exclusive time (1600 ms) came from the single call to the method *next_event*. This is about 35 times greater than the time consumed for the low luminosity case and therefore it does not scale with the increased

| Step | Input data | Read byte-stream | Unpack ROBs | Fill maps | RD Event |
|---|---|---|---|---|---|
| no pileup | 9.8K digits | 0.76 ms | 231 ms | 101 ms | 46 ms |
| pileup | 90K digits | 12 ms | 2200 ms | 830 ms | 1600 ms |

Table 2: Execution times for the main EFTDRConverter tasks. For each task, the times are given without pileup and with design luminosity pileup. "RD Event" denotes the time it takes to fill event objects using the EDM.

number of Digits. Again, this method is dominated by software from the "RD Event" event data model, which is discussed in Section 3.3.3.

The third largest total exclusive time (830 ms) was consumed by the *UnpackRob* methods. The average number of RODs accessed by these methods did not increase very much from the low luminosity case: 80 RODs for Pixel and 70 RODs for SCT. However, as mentioned above, the number of Digits increased by an order of magnitude for the high luminosity case which accounts for the factor of 10 increase in the total average exclusive time. The average exclusive times per call of the *UnpackRob* methods for pileup (3 ms for Pixel and 9 ms for SCT) are also consistent with the increase in the number of Digits.

The time taken to read in the packed raw event from the input stream was measured to be about 12 ms which scales well with the increased number of Digits. The average time consumed per event by the Gaudi framework was approximately 116 ms, which is consistent with the increase in Digits and is again less than 3% of the total average event time.

The results from the analysis of the *EFTDREventCnv* benchmarks can be summarized as follows:

- The average number of Digits in the high luminosity sample is about 10 times greater than that in the low luminosity sample.
- The average time per event is dominated by Digit processing and scales linearly with increasing number of Digits.
- The old Raw Event Data Model makes a significant contribution to the average event time and increases non-linearly with increased pileup.
- The average time taken to read an event from the byte stream is relatively small and scales linearly with increasing number of Digits.
- The time consumed by the Gaudi framework scales linearly with pileup and makes a negligible contribution to the total event time.

Table 2 contains the results discussed in the last two sections.

**3.3.2.3 Results with Time Stamps** In order to obtain an independent measurement of the real time consumed per event and test the consistency of the TAU results, time stamps were placed in the test algorithm of the *EFTDREventCnv* package.

The algorithm was executed over both the 100 low luminosity $Z \rightarrow b\overline{b}$ events and the 20 high luminosity jet events. The results were used to calculate the average time consumed per event for both samples.

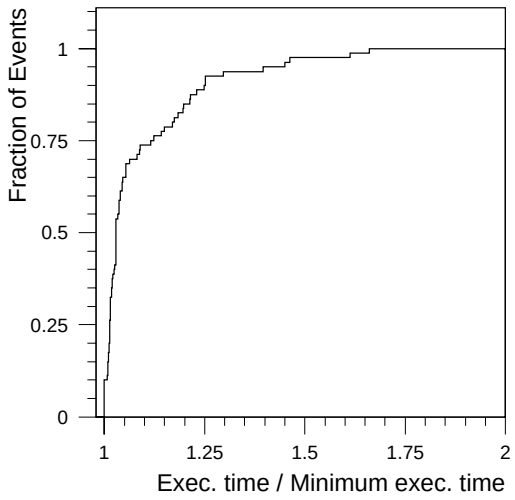



Figure 3: Fraction of events as a function of measured time using time stamps for low luminosity $Z \rightarrow b\overline{b}$ events. To generate this plot, 8 events were processed with *EFTDREventCnv* 10 times each. For each event, the 10 execution times were then normalized to the minimum execution time. The figure shown is the integral of the latency distribution.

In addition, the TAU profiling tool was turned off and the above procedure was repeated to get an estimate of the overhead due to TAU itself.

Relatively large variations in event-to-event execution times were observed. In addition to the variations due to event occupancy, an additional execution time variation was observed by running the code on the same event multiple times. Figure 3 shows such a spread when executing the converter on a few low luminosity events. For example, for a single event, 5% of the time the measurement will yield a latency 50% higher than "normal". This same event latency tail can be attributed to system- and OS-related contributions, e.g., context switching. One way to minimize these contributions is to repeat the measurement a few times and quote the minimum observed latency as the final result.

The following conclusions were drawn from the time-stamped code:

- The average time per event for low luminosity is 430 ms, which is consistent with the TAU profiling result of 470 ms (see Section 3.3.2.1);
- The average time per event for high luminosity is 4800 ms, which is consistent with the TAU profiling result of 4900 ms (see Section 3.3.2.2);
- The overhead due to TAU was estimated at 15% for both the high and low luminosity data samples. This is consistent with estimates obtained from studying the profile output mentioned in section 3.3.2 with the aid of the TAU overhead specifications mentioned in Appendix A;
- OS-related overheads can contribute up to 50% of additional latency per event, skewing the performance measurements.

### 3.3.3 RD Event

The electron ID application was used to estimate the overhead due to the present Atlas Event Data Model. In addition to the standard libraries, also the code for the present EDM was instrumented with TAU . However, due to the design of the EDM software, certain sections of the code with fast execution times were called very frequently during a typical run. They, in turn, caused considerable distortions of the time profiles in the case of a complete instrumentation of the EDM due to the additional TAU overhead (see Appendix A). TAU was therefore only used to identify the main entry and exit points to the EDM, which were then instrumented to get an estimate of the integral time spent in the EDM during a run.

In the case of pile-up events, about 30% of the total execution time was spent in the EDM code. However, it should be mentioned that the present EDM also handles certain geometry information, so that the access time for this is included in the above estimate.

### 3.3.4 StoreGate

Two versions of the Atlas Transient Event Store (TES) management software StoreGate (StG) have been evaluated and compared to the standard Gaudi TES implementation. A first implementation of StoreGate on top of the existing Gaudi TES implementation (subsequently called "old StoreGate"), as it was available in Athena release 1.3.2, was first evaluated. An improved version (subsequently called "new StoreGate"), which was available as a private implementation[4] at the time of this test, was also evaluated. The software was not compiled in an optimized form; however, all tests were done with the same settings on the same machine so that relative comparisons are still valid. The software was instrumented with TAU to get an idea about the distribution of execution times among the different modules. When possible, timing measurements were done with the uninstrumented software and simple timestamps in dedicated places to avoid distortions of the execution times due to TAU overheads.

The test algorithm first stores $N$ objects in the plain Gaudi TES and reads them back afterwards. The time for each step is measured with the Athena ChronoSvc. The same procedure is then repeated in the same run with StoreGate as the TES management software. Running the two tests in the same job exposes the last run test to all objects stored by the first test in the TES, i.e if the pure TES test is run first and the StoreGate test afterwards, the StoreGate test will be exposed not only to the objects created by itself but also to the objects created by the TES test before.

In the case of the "old StoreGate" implementation the following observations can be made:

- The execution time did not scale linearly with the number of handled objects (see Figure 4).
- The measured times for StoreGate were completely different if the TES test was run first or the StoreGate test was run first (Column 3 and 4 in Figure 4 show different heights).

[4]The functionality of this version is now available in the official Atlas software since the recent Atlas release 2.4.1

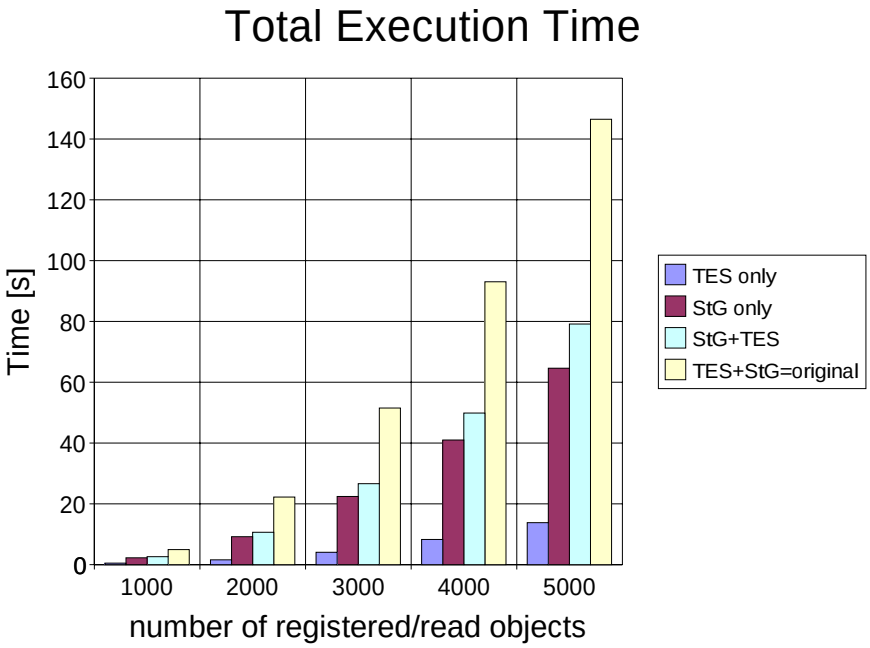



Figure 4: Total access times for registering and retrieving $N$ objects of the same type with TES or the "old StoreGate" implementation. The 4 columns show the access times for running the TES test only (TES only, column 1), the StG test only (StG only, column 2), first the StG test and then the TES test (StG+TES, column 3) in the same job and finally the reversed combination (TES+StG, column 4). Since in this implementation of StG also the objects which have been registered by the TES test are exposed to StG the columns 3 and 4 don't show the same access times.

- If TES and StoreGate tests were run individually, so that no interaction could take place between them, the time to store and retrieve objects with StoreGate was about five times the time needed by the original Gaudi TES implementation. This was due to 5 times more calls to the "Gaudi RegistryEntry" module in the "old StG" implementation when searching through objects for retrieving.

The same tests were repeated with the "new StoreGate" implementation in the same environment. The main observations were now:

- TES and StoreGate showed the same total execution times (see Figure 5).
- There was no interaction anymore between the StoreGate and the TES tests if they were run one after the other in the same job (Column 3 and 4 in Figure 5 have almost the same height).
- Registering objects was very fast for the "new StoreGate" : 1 ms for StG compared to 16 ms for the plain TES with 1000 objects of the same type (see Figure 6).
- Reading back objects with a key was about two times slower with StG than with the plain TES (see Figure 6).

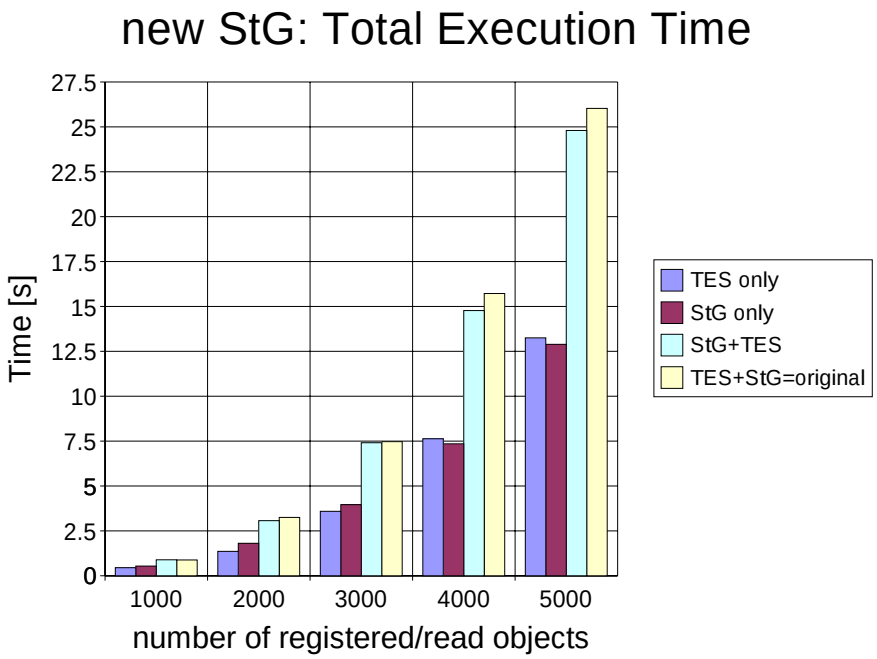



Figure 5: Total access times for registering and retrieving $N$ objects of the same type with TES or the 'new StoreGate" implementation. The 4 columns show the access times for running the TES test only (TES only, column 1), the StG test only (StG only, column 2), first the StG test and then the TES test (StG+TES, column 3) in the same job and finally the reversed combination (TES+StG, column 4).

- The total time for registering and reading back objects was the same for TES and StG. With the TES it took about the same time to register and retrieve an object, whereas in the case of StG registering an object was much faster than reading back an object (see Figure 6).

The above tests were all done with $N$ objects of the same time. For the more realistic case of $N$ objects of different types the "new StG" implementation showed a performance advantage over the plain TES implementation.

- For reading back e.g. 1000 objects of 5 different types (= 5000 objects in total) a performance advantage of the "new StG" respective to the TES was observed (see Figure 7):

  $$\text{newSTG read}(N \text{ objects of } M \text{ types}) \simeq 0.3 \cdot \text{TES read}(N \text{ objects of } M \text{ types})$$

- The approximate scaling of access times for reading back $N$ objects of $M$ different types ($= N \cdot M$ objects in total) were (see also Figure 7) in the case of the TES

  $$\text{TES read}(N \cdot M \text{ objects of 1 type}) \simeq \text{TES read}(N \text{ objects of } M \text{ types})$$

  and

  $$\text{newStG read}(N \cdot M \text{ objects of 1 type}) \simeq M \cdot \text{newStG read}(N \text{ objects of } M \text{ types})$$

  for the "new StG".

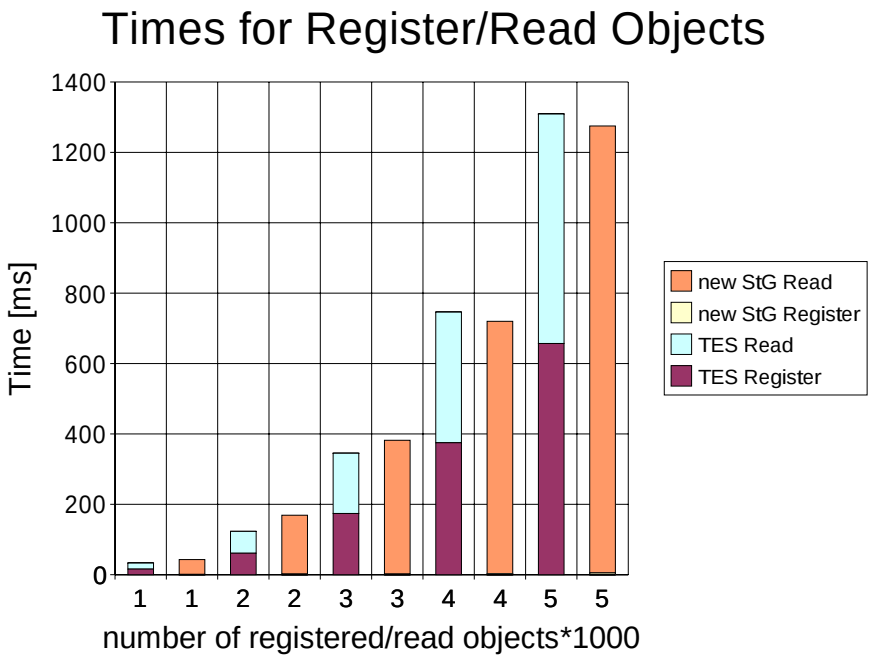



Figure 6: Access times for registering and retrieving $N$ objects of the same type with the TES or the "new StoreGate" implementation. For the access to the TES almost equal times for registering and reading objects are observed, whereas in the case of the "new StG" the time is dominated by reading back the objects. The time to register objects with the "new StG" is almost invisible in the plot.

# 4 Implications for the HLT

## 4.1 Conclusions of validation exercise

The bottom line when evaluating Athena for online use is whether it adds undue resource requirements to the trigger system. A first measurement of these overheads, in the context of Figure 1, was presented in Section 3.

It is worth noting that the notion of what is a "reasonable overhead" can be difficult to define. Depending on how the various system components factorize, it can be very difficult to cleanly separate algorithmic work from data movement work. This is especially important in the case of the Data Model, where the data organization and access methods have intelligent – and thus algorithmic – components. For example, if a data model is very "smart", it can make event selection work more efficient while making the data model appear slow. This performance trade-off needs to be optimized during the ATLAS EDM and the HLT selection software design process.

One way to place bounds on what a reasonable overhead should be is to compare with other frameworks or experiments. For the electron identification studies presented in the HLT Technical Proposal [1], the data access and framework overheads of ATRECON amounted to 40% to 50% of the algorithm execution time. Similar overheads in BaBar amount to 10% to 15% [8].

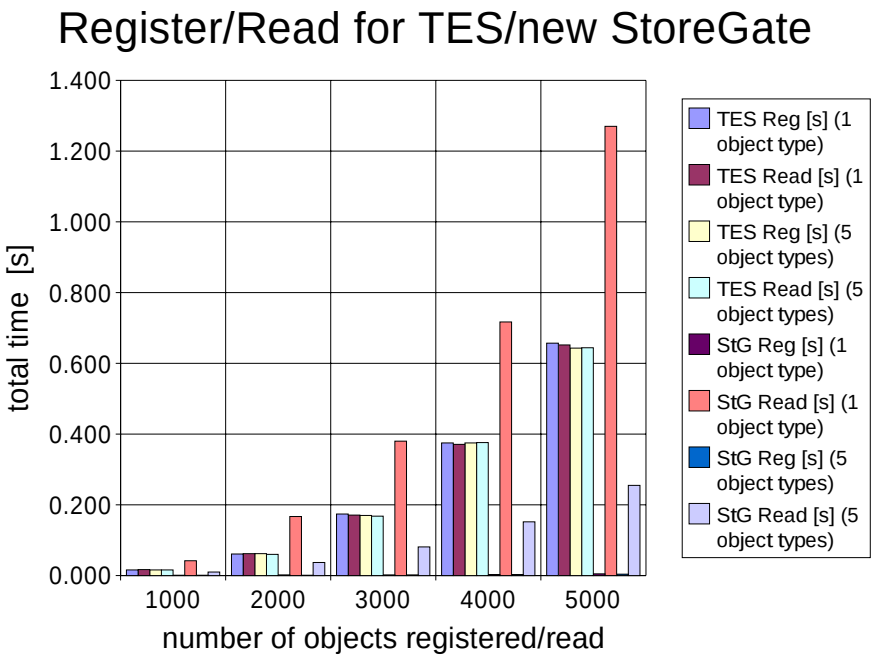



Figure 7: Comparison of access times for registering and retrieving n objects in total of the same type and of 5 different types with the TES or the "new StoreGate" implementation.

Section 3.3.3 shows that the EDM benchmarked in this study can take up to 30% of the total execution time for an event. This is a very large fraction of the CPU budget for selecting events and is an area that should be monitored closely.

A question that must be answered, however, is how smart does the EDM need to be for the HLT (this is being addressed by the PESA Requirements Document [6]). The danger is that the "offline EDM" may incorporate functionality that – while essential for the offline task – is not necessary for the HLT. This excess functionality may, in turn, degrade the performance of the HLT software. This is especially crucial for the HLT because the HLT software suffers the inefficiencies of the EDM in two fronts: in the byte-stream conversion to EDM format; and in algorithm access to the data.

Another area that must be monitored is data unpacking. As shown in Section 3.3.2, converting raw digits to an offline format can take considerable time; typically unpacking times are more than algorithm execution times. Pure framework overheads, that is, overheads due to Gaudi/Athena methods, consume just a few percent of the overall budget, as shown in Section 3.3.2.

Clearly, as of today, the ATLAS offline software is not good enough to be used as an HLT event selection framework. However, this is not unexpected since ATLAS software is in an early development phase. The HLT validation work must ensure that Athena development follows a course compatible with the HLT performance requirements.

Ultimately, the decision on whether an overhead is acceptable will rely more on costing constraints than on code efficiency arguments. However, the same "performance vs. cost" issues that the HLT is facing today will be faced eventually by the offline. In this sense, the HLT software can be a "testbed" for the offline community.

## 4.2 Recommendations

More work needs to be done before making an informed decision on the suitability of the offline software for the HLT. Although we are years away from full deployment, a few steps taken today can ease tremendously the task of using the offline software (or a subset thereof) in the High Level Trigger.

First, the functional evaluation of the offline software should be an integral part of the validation exercise. Compliance with requirement documents (e.g., [6], [7]) should also be monitored. There are a few requirements of PESA SW that are particularly important for the validation work:

- **modularity:** We should start to test framework modularity now. For example, one should be able to build and run Athena without certain services. (Experience says things do not become modular, they start modular.)
- **independence:** We must be able to build Athena (and its components) completely stand-alone, with all libraries, auxiliary files, and services on a local processor. The possibility to build a stand-alone version should be a standard option in the regular builds. Otherwise, we need a special EF repository.
- **Athena light:** We should have a minimal version of Athena (related to modularity above). We should be able to build the Athena "kernel" and nothing more (just an event loop for example), in order to be able to replace services at will. Once the core functionality (minimal set of ATLAS components) is factorized, performance evaluations will be much easier.

The strategy of evaluating the system performance of development code needs to be accepted as a valid approach in validation. The HLT cannot afford to wait until a fully-optimized and finalized version of the offline software is available. This strategy has the distinct advantage of ensuring that the offline software performance improves by providing concrete feedback to the offline community. A recent instance of this feedback is documented in Section 3.3.4.

Once the ART migration is complete in ATLAS, the offline code should be instrumented with a profiling tool. All components relevant to the HLT must be fully instrumented (e.g., reconstruction, StoreGate, converters). It is not necessary to use TAU – certain compilers have sufficient profiling options. This instrumentation should be part of a standard release, and a user ought to be able to turn it on or off at will. System metrics, both CPU and memory utilization, should be integral parts of the offline code. It should also be possible to correlate these metrics to events or to sections of an event.

Ideally, the profiling work documented in this note should be carried out in the ATLAS offline community. It should also fall in the domain of code quality control. Code developers should also have access to the profiling results – they are the people best suited for optimizing their code. Creating a standard "test suite" that is executed after each release may help automate this procedure. Achieving quality software (in terms of system performance) will be ensured by integrating this activity in the offline software early enough. This, in turn, will guarantee an efficient spending of the CPU budget, both for the HLT and for the offline.

The High Level Trigger will probably only need a subset of the full offline functionality in the online environment. It should be possible to have a light version of Athena, where unnecessary functionality can be stripped away or disabled with no performance penalties. For example, neither graphic displays nor MC truth bookkeeping will be needed in the Event Filter farms. This requires a high level of modularity in the framework.

On the HLT side, we need to create a "Trigger Coding Standards" document or guidelines that go beyond proper naming conventions and documentation requirements. These guidelines would address known programming practices that – although possibly elegant – may make trigger code slow and inefficient[5].

The final evaluation of Athena as an online framework will have to rely on a detailed computing model – and an implementation – of the HLT selection processing. The HLT community must provide such a model, including:

- Handling, frequency, and direction of meta-data flow
- Realistic environment description (e.g., what is disk-resident, memory-resident, and network-resident)
- Ultimately, performance constraints
- List of Athena/Gaudi components relevant for HLT ("Athena Light").

A few specific recommendations can be gleaned from this report:

- The EDM performance is critical to the success of the HLT. It must therefore be closely monitored. The final EDM design should be developed with the HLT functional and performance requirements in mind, as the HLT is the weakest "performance link".
- Adding the e-ID execution times of Section 3.3.1 to the raw data conversion times of Section 3.3.2 yields a first rough estimate of EF execution time for electrons: 5 s (30 s) at low (high) luminosity. Correction factors of 1.8 (3) for low (high) luminosity were applied to the algorithm execution times to account for the expected gains from a seeded reconstruction [9]. At low luminosity, this is equivalent to 0.8 s on a 180 SI95 machine. This extrapolation to 2006 performance is a factor of five [14] higher than the budgeted CPU cycles for this trigger. A roadmap to reach this performance should be provided by the offline community[6].
- Performance goals should be set for the ATLAS offline software. These goals should be monitored at each major release.

# 5 Summary and Outlook

No hard conclusions can presently be drawn – given the state of the software – on Athena's suitability as a High Level Trigger event selection framework. However, by starting a

[5] This is of course a very difficult task since it depends on programming language, OS, platform, etc.

[6] This is a very rough first estimate that must be carefully monitored – it is given here as an "order of magnitude" estimate of the improvements needed to reach full HLT performance.

program of measurements now, the HLT community can provide feedback to the offline community on possible areas for optimization.

The short term aim of this work is to establish software metrics to be monitored. The longer term aim is to ensure that these metrics help point the way to efficient software that satisfies the stringent performance requirements of the HLT.

The HLT validation studies so far have helped in optimizing the performance of the TES access interface (StoreGate). They have also shown that the Event Data Model is a performance-critical component of the event selection and that it must be monitored carefully. The raw data converter, unique to the Event Filter, is another component whose performance will be critical to the success of the Event Filter.

This note contains a snapshot of the performance of the offline code at an early stage of development. This one "data point" will be used as the first reference point from which to track performance improvements over time. The next "data points" will be: the new StoreGate, the full set of converters, and the new EDM.

Once a suitability decision is reached on Athena, two important points must be considered. If Athena is validated as an HLT framework, then an understanding must be reached with the offline community on the need and implications of maintaining Athena's suitability. If instead Athena is found to be unsuitable for the HLT task, then an alternative solution must be developed. Furthermore, in such a case, the implications on manpower and future maintenance must be considered.

# A Profiling with TAU

TAU [12] is a free profiling tool, which can collect information in a distributed computing system, it can deal with multi-threaded programs and it can handle programs using shared libraries. TAU was also used for performance evaluations for the Atlas second level trigger reference software [13].

Code instrumentation can be done in two ways

- Source code instrumentation : Each source code file of the program parts which should be instrumented has to be analysed by a sequence of special preprocessing steps. They automatically modify the original source by inserting the necessary calls to the TAU profiling API [15]. The newly generated instrumented source files can then be compiled and linked in the same way as the original program.
- Dynamic instrumentation : Here the executable object code of the program is modified by TAU . The main advantage is that code can be profiled without recompilation and access to the program sources is not required. This instrumentation version, however, was at the time of writing still marked as "experimental" by the TAU developers and was not used for the studies presented in this note.

A basic installation of TAU for source code instrumentation requires besides the distribution kit for TAU itself also the distribution kit for the "Program Database Toolkit" (PDT) [16]. It contains the necessary preprocessors for analysing the source code and producing the "program databases", which are then used for automatic source instrumen-

tation in TAU . For the studies presented in this paper version 2.9.13 of TAU and version 1.3 of PDT were used.

### A.1 Source Code Instrumentation for Gaudi/Athena

The calls to the TAU profiling API can either be inserted by hand in the source code or automatically with a sequence of preprocessors available from the TAU and PDT tool kits. For the presented studies the second method was used.

The automatic instrumentation procedure inserts macro statements with the calls to the TAU profiling API in the source code. These macro statements expand to zero when no profiling options are used in the compile step. This makes it easy to produce instrumented and not instrumented versions of a program from the same source code.

For automatic instrumentation the normal compile and link step has to be augmented by the following sequence of preprocessing steps for a typical source file (*fn.cxx*):

- `cxxparse fn.cxx <options>`

  *output files: fn.il, fn.pdb*

  In `cxxparse` the original source file *fn.cxx* is first parsed with the EDG compiler front end `edgcpfe` [17] and the results are stored in an "intermediate-language tree" file *fn.il*. This file is then analysed by `taucpdisp` and informations about the program structure are written to a "program database" [16] file *fn.pdb* for further use with `tau_instrumentor`.

- `tau_instrumentor fn.pdb fn.cxx`

  *output file: fn.inst.cxx*

  `tau_instrumentor` creates from *fn.pdb* and the original source code file *fn.cxx* an instrumented source code file *fn.inst.cxx*, which contains macros with calls to the TAU profiling API.

- `g++ <options> <TAU_options> fn.inst.cxx`

  *output file: fn.inst.o*

  `link step with fn.inst.o (+ TAU libraries)`

  *output: executables, libraries*

  Builds with the instrumented source code files the instrumented versions of the object files, the executable programs and the (shared) libraries. If the `TAU_options` are omitted, not instrumented program versions are produced.

For the Atlas software this sequence can be automatically initiated by overwriting the normal compile and link process in the software release tools used by Atlas. At the time

of writing these are, CMT for the management of the Gaudi base libraries and SRT for all the other Atlas code[7].

- For CMT a package "InstrumentTAU" was created, which automatically instruments a source package when included in the respective requirements file of the package. Typically this is done with a CMT `use` directive

  ```
  use InstrumentTAU v*
  ```

  The package "InstrumentTAU" and more instructions together with an instrumented example of the Gaudi base libraries can be found at [18].

- For SRT *Makefile* fragments have been created. They have to be manually inserted in the *Makefiles* of the packages which should be instrumented. Instructions and examples can be found at [18].

Since the EDG compiler front end is used to parse the original C++ source code to produce the "intermediate-language tree" files and the "program database" files, the source code must support this front end as an additional compiler platform. This may produce parsing errors on the different platforms if e.g. not the same language set is implemented on both platforms. Most of the parsing problems with *edgcpfe* with respect to *g++* arise therefore from different resolution strategies for templates, from different implementations of the STL library and from language enhancements only supported by *g++* and not by *edgcpfe*.

### A.2 TAU overheads

The calls to the profiling API which are inserted in the source code produce an additional time overhead which may significantly distort timing profiles for frequently called program parts. As shown in Figure 8, the time overhead introduced by a typical call to the profiling API was measured on an Intel Pentium III (733 MHz clock frequency) as $\simeq 12\,[\mu s]$ for function registration and first call of the API within a function and as $\simeq 1-2\,[\mu s]$ for all subsequent calls. The time for registration depends on the length of the function name and on the function type. The numbers are in good agreement with similar numbers [19] quoted by the TAU developer team for measurements on SGI machines, namely $8-40\,[\mu s]$ for function registration and $\simeq 0.8-1.6\,[\mu s]$ for all subsequent calls.

## B Profiles for Electron ID

In the following profiles always the top 20 time consumers are shown for the respective ordering of the profiles.

[7] At the time of writing an evaluation is ongoing with the aim to make CMT as source code manager available for the complete Atlas codebase.

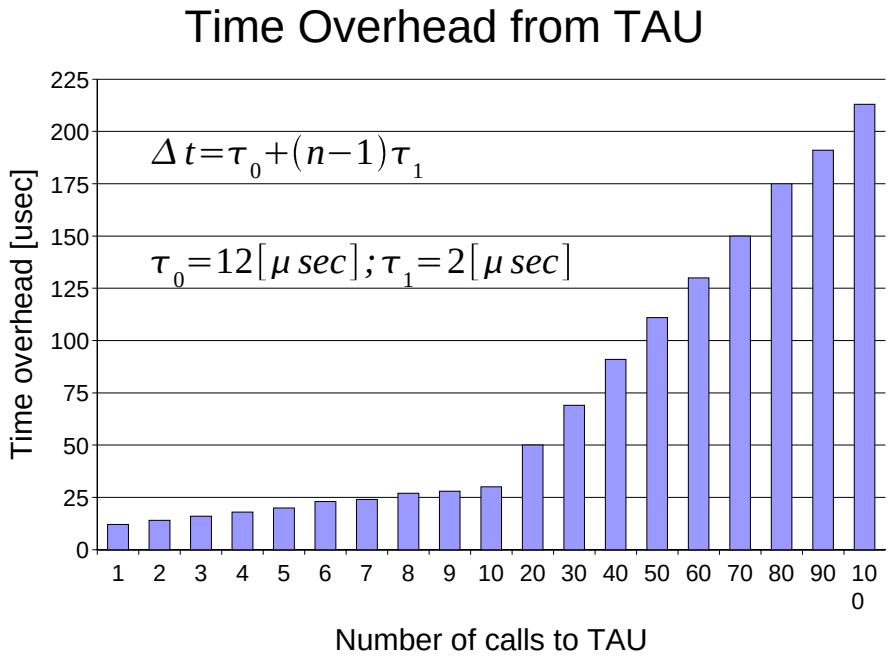



Figure 8: TAU execution time as a function of number of calls..

## B.1 Single Electron Data

### B.1.1 Profile ordered according inclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 100.0 | 0.168 | 3:55.346 | 1 | 14 | 235346948 | 0 | int main(int, char **) |
| 98.9 | 0.015 | 3:52.809 | 1 | 1 | 232809738 | 0 | StatusCode ApplicationMgr::nextEvent(int) *this |
| 98.9 | 1,401 | 3:52.809 | 1 | 625 | 232809723 | 0 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 97.4 | 2 | 3:49.256 | 100 | 500 | 2292561 | 4.155 | StatusCode EventLoopMgr::executeEvent(void *) *this |
| 97.4 | 13 | 3:49.248 | 100 | 3000 | 2292481 | 11.46 | StatusCode MinimalEventLoopMgr::executeEvent(void *) *this |
| 97.4 | 23 | 3:49.233 | 1500 | 7500 | 152822 | 4.093 | StatusCode Algorithm::sysExecute() *this |
| 86.3 | 40,589 | 3:23.206 | 400 | 2.77573E+07 | 508016 | 1.17E+05 | bool SlidingWindowFinder::nextcluster() *this |
| 83.6 | 7 | 3:16.684 | 100 | 1700 | 1966845 | 10.34 | StatusCode LArClusterMaker::execute() *this |
| 82.0 | 87 | 3:12.892 | 100 | 35947 | 1928921 | 69.26 | StatusCode LArSlidingWindow::execute() *this |
| 69.0 | 1:00.205 | 2:42.486 | 2.77084E+07 | 2.78355E+07 | 6 | 24.63 | double CaloTowerContainer::et(int, int) const *this |
| 43.8 | 1:42.984 | 1:42.984 | 2.78387E+07 | 0 | 4 | 31.59 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 5.4 | 96 | 12,740 | 100 | 39830 | 127409 | 91.36 | StatusCode CaloClusterBuilderSW::execute() *this |
| 4.3 | 440 | 10,150 | 27992 | 55984 | 363 | 6.084 | MsgStream &MsgStream::doOutput() *this |
| 4.1 | 6,460 | 9,689 | 28011 | 140055 | 346 | 692.8 | void MessageSvc::reportMessage(const Message &) *this |
| 2.7 | 25 | 6,419 | 200 | 3600 | 32099 | 15.77 | StatusCode CaloTowerMaker::execute() *this |
| 2.6 | 1,666 | 6,162 | 200 | 723584 | 30815 | 3314 | StatusCode LArTowerBuilder::execute() *this |
| 1.8 | 63 | 4,311 | 100 | 10900 | 43111 | 41 | StatusCode XKalMan::execute() *this |
| 1.6 | 263 | 3,760 | 1100 | 72600 | 3419 | 319.4 | StatusCode LArClusterCorrection::execute() *this |
| 1.4 | 71 | 3,279 | 100 | 6700 | 32798 | 4035 | StatusCode egammaBuilder::execute() *this |
| 1.3 | 935 | 3,043 | 28013 | 28013 | 109 | 463 | std::ostream &operator<<(std::ostream &, const Message &) |

### B.1.2 Profile ordered according exclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 43.8 | 1:42.984 | 1:42.984 | 2.78387E+07 | 0 | 4 | 31.59 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 69.0 | 1:00.205 | 2:42.486 | 2.77084E+07 | 2.78355E+07 | 6 | 24.63 | double CaloTowerContainer::et(int, int) const *this |
| 86.3 | 40,589 | 3:23.206 | 400 | 2.77573E+07 | 508016 | 1.17E+05 | bool SlidingWindowFinder::nextcluster() *this |
| 4.1 | 6,460 | 9,689 | 28011 | 140055 | 346 | 692.8 | void MessageSvc::reportMessage(const Message &) *this |
| 0.8 | 1,794 | 1,822 | 1 | 33 | 1822980 | 0 | StatusCode ApplicationMgr::initialize() *this |
| 2.6 | 1,666 | 6,162 | 200 | 723584 | 30815 | 3314 | StatusCode LArTowerBuilder::execute() *this |
| 98.9 | 1,401 | 3:52.809 | 1 | 625 | 232809723 | 0 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 0.6 | 1,297 | 1,297 | 58268 | 0 | 22 | 130.9 | bool LArTowerBuilder::cell_segmentation(Identifier &, float &, float &, Identifier &) *this |
| 0.7 | 1,076 | 1,563 | 1000 | 489217 | 1563 | 1121 | void LArqgcalo::fill(double, double, double, double, int) *this |
| 1.3 | 935 | 3,043 | 28013 | 28013 | 109 | 463 | std::ostream &operator<<(std::ostream &, const Message &) |
| 0.3 | 820 | 820 | 227681 | 0 | 4 | 50.02 | Identifier CaloCell::ID() const *this |
| 0.5 | 786 | 1,231 | 200 | 6000 | 6160 | 3823 | StatusCode LArG3CellBuilder::execute() *this |
| 0.6 | 774 | 1,339 | 224104 | 168078 | 6 | 22.45 | void Message::decodeFormat(const std::string &) const *this |
| 0.9 | 768 | 2,107 | 28013 | 224104 | 75 | 111.5 | void Message::makeFormattedMsg(const std::string &) const *this |
| 0.2 | 573 | 573 | 85361 | 0 | 7 | 49.36 | bool comp_RPAZ::operator()(const SiDetDescrElement *, const SiDetDescrElement *) *this |
| 0.4 | 542 | 858 | 106096 | 212192 | 8 | 43.22 | void CaloTower::addcell(CaloCell *, float) *this |
| 0.6 | 536 | 1,414 | 52877 | 317262 | 27 | 3.715 | void CaloCluster::addcell(CaloCluster::cell_type *) *this |
| 4.3 | 440 | 10,150 | 27992 | 55984 | 363 | 6.084 | MsgStream &MsgStream::doOutput() *this |
| 0.2 | 402 | 568 | 172376 | 172376 | 3 | 58.84 | double proxim(double, double) |
| 0.6 | 400 | 1,520 | 1 | 164347 | 1520448 | 0 | void XK_Tracker::Geometry() *this |

## B.2 Jet Data

### B.2.1 Profile ordered according inclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 100.0 | 0.21 | 12:30.384 | 1 | 14 | 750384018 | 0 | int main(int, char **) |
| 97.2 | 23 | 12:09.419 | 1 | 1 | 729419237 | 0 | StatusCode ApplicationMgr::nextEvent(int) *this |
| 97.2 | 53,715 | 12:09.395 | 1 | 625 | 729395953 | 0 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 88.3 | 2 | 11:02.554 | 100 | 500 | 6625541 | 4.477 | StatusCode EventLoopMgr::executeEvent(void *) *this |
| 88.3 | 13 | 11:02.545 | 100 | 3000 | 6625458 | 10.82 | StatusCode MinimalEventLoopMgr::executeEvent(void *) *this |
| 88.3 | 24 | 11:02.530 | 1500 | 7500 | 441687 | 4.328 | StatusCode Algorithm::sysExecute() *this |
| 37.7 | 44,017 | 4:42.957 | 895 | 2.79942E+07 | 316153 | 6.261E+04 | bool SlidingWindowFinder::nextcluster() *this |
| 36.7 | 7 | 4:35.526 | 100 | 1700 | 2755264 | 9.851 | StatusCode LArClusterMaker::execute() *this |
| 35.7 | 274 | 4:28.106 | 100 | 145587 | 2681063 | 1515 | StatusCode LArSlidingWindow::execute() *this |
| 31.7 | 1:06.347 | 3:58.208 | 2.77735E+07 | 3.17144E+07 | 9 | 49.67 | double CaloTowerContainer::et(int, int) const *this |
| 25.0 | 3:07.572 | 3:07.572 | 3.02101E+07 | 0 | 6 | 64.23 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 16.9 | 28 | 2:07.143 | 200 | 3600 | 635719 | 15.42 | StatusCode CaloTowerMaker::execute() *this |
| 16.9 | 38,195 | 2:06.764 | 200 | 1.7271E+07 | 633820 | 7.222E+04 | StatusCode LArTowerBuilder::execute() *this |
| 14.6 | 44 | 1:49.569 | 100 | 7708 | 1095692 | 258.3 | StatusCode egammaBuilder::execute() *this |
| 14.2 | 293 | 1:46.459 | 436 | 64092 | 244173 | 300.3 | StatusCode EMShowerBuilder::execute(egamma *) *this |
| 13.1 | 10,197 | 1:38.280 | 2180 | 7.33631E+06 | 45083 | 1.228E+04 | CaloCellList::Tselect |
| 11.7 | 35,433 | 1:28.082 | 7.33631E+06 | 2.2162E+07 | 12 | 50.04 | bool CaloCellList::CellSel::operator()(CaloCell *) *this |
| 9.3 | 71 | 1:09.919 | 100 | 10900 | 699195 | 382.4 | StatusCode XKalMan::execute() *this |
| 6.5 | 3 | 48,823 | 872 | 872 | 55991 | 0.9652 | void CaloCellList::select(double, double, double) *this |
| 6.2 | 4 | 46,265 | 872 | 872 | 53057 | 0.6054 | void CaloCellList::select(double, double, double, double) *this |

### B.2.2 Profile ordered according exclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 25.0 | 3:07.572 | 3:07.572 | 3.02101E+07 | 0 | 6 | 64.23 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 31.7 | 1:06.347 | 3:58.208 | 2.77735E+07 | 3.17144E+07 | 9 | 49.67 | double CaloTowerContainer::et(int, int) const *this |
| 97.2 | 53,715 | 12:09.395 | 1 | 625 | 729395953 | 0 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 37.7 | 44,017 | 4:42.957 | 895 | 2.79942E+07 | 316153 | 6.261E+04 | bool SlidingWindowFinder::nextcluster() *this |
| 16.9 | 38,195 | 2:06.764 | 200 | 1.7271E+07 | 633820 | 7.222E+04 | StatusCode LArTowerBuilder::execute() *this |
| 11.7 | 35,433 | 1:28.082 | 7.33631E+06 | 2.2162E+07 | 12 | 50.04 | bool CaloCellList::CellSel::operator()(CaloCell *) *this |
| 3.8 | 28,584 | 28,584 | 1.41901E+06 | 0 | 20 | 136.1 | bool LArTowerBuilder::cell_segmentation(Identifier &, float &, float &, Identifier &) *this |
| 2.4 | 18,094 | 18,094 | 4.83699E+06 | 0 | 4 | 262 | Identifier CaloCell::ID() const *this |
| 3.4 | 17,729 | 25,864 | 8.24554E+06 | 8.24554E+06 | 3 | 39.05 | double proxim(double, double) |
| 2.2 | 16,460 | 16,821 | 200 | 6000 | 84109 | 7.74E+04 | StatusCode LArG3CellBuilder::execute() *this |
| 2.1 | 15,376 | 15,581 | 36 | 422 | 432819 | 2.001E+06 | unsigned long doLoad(const std::string &, System::ImageHandle *) |
| 2.4 | 11,549 | 18,242 | 2.35391E+06 | 4.70783E+06 | 8 | 45.98 | void CaloTower::addcell(CaloCell *, float) *this |
| 13.1 | 10,197 | 1:38.280 | 2180 | 7.33631E+06 | 45083 | 1.228E+04 | CaloCellList::Tselect |
| 1.5 | 9,837 | 11,150 | 3613 | 1.50572E+06 | 3086 | 8363 | unsigned long DataObject::release() *this |
| 1.3 | 9,704 | 9,958 | 100 | 7200 | 99585 | 2.358E+04 | StatusCode XKaEventCollection::execute() *this |
| 1.3 | 9,384 | 9,384 | 1.03659E+07 | 0 | 1 | 31.73 | double CaloCell::eta() const *this |
| 1.3 | 8,783 | 9,650 | 100 | 198387 | 96508 | 2.404E+04 | void XK_Tracker::ConvertP(XK_Condition &, const XKdDetectorVisitor &) *this |
| 2.7 | 8,702 | 20,104 | 3.59159E+06 | 3.59159E+06 | 6 | 45.79 | bool CaloCellList::ConeR::test(double, double) *this |
| 1.1 | 8,286 | 8,286 | 9.61588E+06 | 0 | 1 | 22.27 | double CaloCell::phi() const *this |
| 1.1 | 8,134 | 8,134 | 8.24554E+06 | 0 | 1 | 33.97 | double round(double) C |

## B.3 Jet Data with Pileup

### B.3.1 Profile ordered according inclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 100.0 | 306 | 41:40.025 | 1 | 14 | 2500025955 | 0 | int main(int, char **) |
| 99.4 | 0.019 | 41:25.523 | 1 | 1 | 2485523355 | 0 | StatusCode ApplicationMgr::nextEvent(int) *this |
| 99.4 | 2:57.054 | 41:25.523 | 1 | 217 | 2485523336 | 0 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 91.4 | 0.873 | 38:04.117 | 31 | 155 | 73681199 | 7.094 | StatusCode EventLoopMgr::executeEvent(void *) *this |
| 91.4 | 4 | 38:04.114 | 31 | 930 | 73681113 | 30.78 | StatusCode MinimalEventLoopMgr::executeEvent(void *) *this |
| 91.4 | 7 | 38:04.109 | 465 | 2325 | 4912064 | 8.446 | StatusCode Algorithm::sysExecute() *this |
| 63.6 | 62 | 26:30.501 | 31 | 3379 | 51306512 | 3674 | StatusCode XKalMan::execute() *this |
| 56.6 | 15 | 23:34.970 | 31 | 3565 | 45644200 | 150.2 | StatusCode XKaSILRec::execute() *this |
| 56.6 | 1:42.318 | 23:34.864 | 31 | 2.35235E+06 | 45640783 | 1.787E+06 | void XK_Algorithm::SILRecN(XK_Tracker &, XK_Condition &, XK_BTrack &) *this |
| 43.8 | 5,489 | 18:15.408 | 560439 | 2.28159E+06 | 1955 | 84.72 | bool XK_Algorithm::SILTrack(XK_Tracker &, XK_Condition &, XK_Trajectory &, XK_BTrack &) *this |
| 25.3 | 2:44.274 | 10:32.773 | 587666 | 2.86629E+07 | 1077 | 524.2 | void XK_Trajectory::Extention(char, const XK_Condition &, XK_Tracker &) *this |
| 19.8 | 3:14.050 | 8:14.271 | 3.10686E+07 | 6.21348E+07 | 16 | 73.39 | bool XK_Helix::PropagatorPPo(const XK_Noise &, const XK_Surface &, XK_Helix &) *this |
| 12.3 | 22,918 | 5:06.780 | 588878 | 8.88142E+06 | 521 | 188.7 | void XK_Trajectory::Smoother(const XK_Condition &) *this |
| 11.4 | 43,947 | 4:46.083 | 9.96924E+06 | 2.36519E+07 | 29 | 49.93 | bool XK_Cell::FPropagator(const char &, int, float, XK_Cell &) *this |
| 9.6 | 36,087 | 4:00.090 | 7.50872E+06 | 1.92719E+07 | 32 | 76.27 | bool XK_Cell::SPropagator(const char &, float, XK_Cell &) *this |
| 9.6 | 9 | 3:59.974 | 62 | 1116 | 3870552 | 28.17 | StatusCode CaloTowerMaker::execute() *this |
| 9.6 | 1:08.978 | 3:59.890 | 62 | 3.17018E+07 | 3869200 | 4.157E+05 | StatusCode LArTowerBuilder::execute() *this |
| 9.1 | 15 | 3:47.603 | 31 | 2500 | 7342052 | 150.5 | StatusCode egammaBuilder::execute() *this |
| 9.0 | 117 | 3:45.101 | 172 | 25284 | 1308730 | 68.14 | StatusCode EMShowerBuilder::execute(egamma *) *this |
| 8.9 | 3:42.188 | 3:42.188 | 3.10686E+07 | 0 | 7 | 83.17 | bool XK_Algorithm::RkutaCP(float *, float *) *this |

### B.3.2 Profile ordered according exclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 8.9 | 3:42.188 | 3:42.188 | 3.10686E+07 | 0 | 7 | 83.17 | bool XK_Algorithm::RkutaCP(float *, float *) *this |
| 19.8 | 3:14.050 | 8:14.271 | 3.10686E+07 | 6.21348E+07 | 16 | 73.39 | bool XK_Helix::PropagatorPPo(const XK_Noise &, const XK_Surface &, XK_Helix &) *this |
| 99.4 | 2:57.054 | 41:25.523 | 1 | 217 | 2485523336 | 0 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 25.3 | 2:44.274 | 10:32.773 | 587666 | 2.86629E+07 | 1077 | 524.2 | void XK_Trajectory::Extention(char, const XK_Condition &, XK_Tracker &) *this |
| 7.6 | 2:08.266 | 3:11.115 | 1.59734E+07 | 5.24334E+07 | 12 | 82.1 | const XK_Counter *XK_Tracker::Guide(const char &, const XK_Counter *const &, const XK_Helix &, const float &) const *this |
| 4.6 | 1:55.452 | 1:55.452 | 1.32412E+07 | 0 | 9 | 79.49 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 5.3 | 1:44.322 | 2:12.888 | 1.49589E+07 | 2.99178E+07 | 9 | 82.57 | XK_ClusterP *XK_Helix::Search(float &, const XK_Counter &) *this |
| 56.6 | 1:42.318 | 23:34.864 | 31 | 2.35235E+06 | 45640783 | 1.787E+06 | void XK_Algorithm::SILRecN(XK_Tracker &, XK_Condition &, XK_BTrack &) *this |
| 8.0 | 1:20.543 | 3:19.723 | 1.6912E+07 | 5.10292E+07 | 12 | 61.34 | bool CaloCellList::CellSel::operator()(CaloCell *) *this |
| 3.1 | 1:18.180 | 1:18.180 | 3.11201E+07 | 0 | 3 | 57.07 | void XK_Helix::NewCov(const XK_Noise &, float *, XK_Helix &) *this |
| 9.6 | 1:08.978 | 3:59.890 | 62 | 3.17018E+07 | 3869200 | 4.157E+05 | StatusCode LArTowerBuilder::execute() *this |
| 2.5 | 1:02.848 | 1:02.848 | 5.24334E+07 | 0 | 1 | 30.46 | const XK_Counter *XK_Layer::Guide(const int &, const float &, const float &, const float &, const float &) const *this |
| 2.4 | 1:00.583 | 1:00.583 | 1590 | 0 | 38103 | 2.061E+04 | void XK_Algorithm::Segments(XK_Condition &, std::list<XK_SpacePo *, std::allocator<XK_SpacePo * > > &, std::list<XK_SpacePo *, std::allocator<XK_Spacel |
| 2.1 | 53,737 | 53,737 | 2.67185E+06 | 0 | 20 | 144.3 | bool LArTowerBuilder::cell_segmentation(Identifier &, float &, float &, Identifier &) *this |
| 11.4 | 43,947 | 4:46.083 | 9.96924E+06 | 2.36519E+07 | 29 | 49.93 | bool XK_Cell::FPropagator(const char &, int, float, XK_Cell &) *this |
| 2.1 | 36,672 | 53,347 | 1.75098E+07 | 1.75098E+07 | 3 | 38.49 | double proxim(double, double) |
| 9.6 | 36,087 | 4:00.090 | 7.50872E+06 | 1.92719E+07 | 32 | 76.27 | bool XK_Cell::SPropagator(const char &, float, XK_Cell &) *this |
| 6.3 | 34,781 | 2:36.554 | 1.79032E+06 | 5.97632E+06 | 87 | 197.5 | bool XK_Trajectory::Initiate(char, const XK_Condition &, XK_SpacePo *&, XK_SpacePo *&, XK_SpacePo *&) *this |
| 1.4 | 34,219 | 34,338 | 62 | 1860 | 553851 | 5.168E+05 | StatusCode LArG3CellBuilder::execute() *this |
| 4.0 | 33,430 | 1:41.070 | 31 | 2.51611E+07 | 3260329 | 3.118E+05 | void XK_Tracker::Initiate(XK_Condition &, const XKdDetectorVisitor &) *this |

## B.4 Jet Data with Pileup ($p_\perp = 15\,[GeV]$ cut for the calorimeter and $p_\perp = 5\,[GeV]$ cut for XKalman++)

### B.4.1 Profile ordered according inclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 100.0 | 249 | 16:05.509 | 1 | 14 | 965509574 | 0 | int main(int, char **) |
| 98.6 | 0.023 | 15:51.883 | 1 | 1 | 951883010 | 0 | StatusCode ApplicationMgr::nextEvent(int) *this |
| 98.6 | 3:00.220 | 15:51.882 | 1 | 217 | 951882987 | 2 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 77.6 | 0.858 | 12:28.978 | 31 | 155 | 24160610 | 7.613 | StatusCode EventLoopMgr::executeEvent(void *) *this |
| 77.6 | 4 | 12:28.976 | 31 | 930 | 24160523 | 72.54 | StatusCode MinimalEventLoopMgr::executeEvent(void *) *this |
| 77.6 | 7 | 12:28.971 | 465 | 2325 | 1610691 | 4.564 | StatusCode Algorithm::sysExecute() *this |
| 24.9 | 8 | 4:00.540 | 62 | 1116 | 3879678 | 25.83 | StatusCode CaloTowerMaker::execute() *this |
| 24.9 | 1:08.716 | 4:00.453 | 62 | 3.17018E+07 | 3878287 | 4.253E+05 | StatusCode LArTowerBuilder::execute() *this |
| 24.1 | 21 | 3:52.992 | 31 | 3379 | 7515903 | 59.79 | StatusCode XKalMan::execute() *this |
| 12.8 | 14 | 2:03.167 | 31 | 3565 | 3973136 | 36.29 | StatusCode XKaSILRec::execute() *this |
| 12.7 | 7,075 | 2:03.061 | 31 | 295522 | 3969731 | 1.121E+05 | void XK_Algorithm::SILRecN(XK_Tracker &, XK_Condition &, XK_BTrack &) *this |
| 12.1 | 13,357 | 1:57.289 | 176 | 8.63265E+06 | 666417 | 1.017E+05 | bool SlidingWindowFinder::nextcluster() *this |
| 12.0 | 1:55.504 | 1:55.504 | 1.32084E+07 | 0 | 9 | 96.62 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 11.6 | 2 | 1:51.585 | 31 | 527 | 3599532 | 3.242 | StatusCode LArClusterMaker::execute() *this |
| 11.4 | 51 | 1:50.159 | 31 | 19201 | 3553540 | 2097 | StatusCode LArSlidingWindow::execute() *this |
| 10.8 | 26,213 | 1:43.811 | 8.59421E+06 | 1.42704E+07 | 12 | 47.71 | double CaloTowerContainer::et(int, int) const *this |
| 10.4 | 18 | 1:40.485 | 31 | 5208 | 3241467 | 34.61 | StatusCode XKaClusters::execute() *this |
| 10.4 | 33,605 | 1:40.345 | 31 | 2.51611E+07 | 3236945 | 3.159E+05 | void XK_Tracker::Initiate(XK_Condition &, const XKdDetectorVisitor &) *this |
| 9.7 | 530 | 1:33.875 | 51230 | 198821 | 1832 | 263.8 | bool XK_Algorithm::SILTrack(XK_Tracker &, XK_Condition &, XK_Trajectory &, XK_BTrack &) *this |
| 6.3 | 3 | 1:00.809 | 62 | 434 | 980798 | 6.364 | StatusCode LArCellMaker::execute() *this |

### B.4.2 Profile ordered according exclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 98.6 | 3:00.220 | 15:51.882 | 1 | 217 | 951882987 | 2 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 12.0 | 1:55.504 | 1:55.504 | 1.32084E+07 | 0 | 9 | 96.62 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 24.9 | 1:08.716 | 4:00.453 | 62 | 3.17018E+07 | 3878287 | 4.253E+05 | StatusCode LArTowerBuilder::execute() *this |
| 5.6 | 53,901 | 53,901 | 2.67185E+06 | 0 | 20 | 172.6 | bool LArTowerBuilder::cell_segmentation(Identifier &, float &, float &, Identifier &) *this |
| 3.6 | 35,006 | 35,127 | 62 | 1860 | 566571 | 5.443E+05 | StatusCode LArG3CellBuilder::execute() *this |
| 10.4 | 33,605 | 1:40.345 | 31 | 2.51611E+07 | 3236945 | 3.159E+05 | void XK_Tracker::Initiate(XK_Condition &, const XKdDetectorVisitor &) *this |
| 3.0 | 28,600 | 28,600 | 8.25884E+06 | 0 | 3 | 54.28 | Identifier CaloCell::ID() const *this |
| 3.0 | 26,352 | 29,037 | 31 | 645545 | 936679 | 2.631E+05 | void XK_Tracker::ConvertP(XK_Condition &, const XKdDetectorVisitor &) *this |
| 10.8 | 26,213 | 1:43.811 | 8.59421E+06 | 1.42704E+07 | 12 | 47.71 | double CaloTowerContainer::et(int, int) const *this |
| 2.4 | 23,542 | 23,624 | 31 | 2232 | 762080 | 2.001E+05 | StatusCode XKaEventCollection::execute() *this |
| 3.7 | 23,310 | 36,043 | 4.60088E+06 | 9.20176E+06 | 8 | 94.61 | void CaloTower::addcell(CaloCell *, float) *this |
| 2.2 | 21,381 | 21,381 | 2.46219E+07 | 0 | 1 | 24.16 | bool comp_Phi::operator()(const XK_Cluster *, const XK_Cluster *) *this |
| 4.9 | 19,118 | 47,293 | 3.99424E+06 | 1.20532E+07 | 12 | 72.38 | bool CaloCellList::CellSel::operator()(CaloCell *) *this |
| 2.2 | 18,749 | 20,912 | 1129 | 2.49573E+06 | 18523 | 5.608E+04 | unsigned long DataObject::release() *this |
| 1.9 | 18,676 | 18,676 | 2.69619E+06 | 0 | 7 | 94.35 | bool XK_Algorithm::RkutaCP(float *, float *) *this |
| 4.4 | 16,549 | 42,113 | 2.69619E+06 | 5.39238E+06 | 16 | 36.13 | bool XK_Helix::PropagatorPPo(const XK_Noise &, const XK_Surface &, XK_Helix &) *this |
| 5.6 | 14,507 | 53,906 | 50559 | 2.47608E+06 | 1066 | 617.7 | void XK_Trajectory::Extention(char, const XK_Condition &, XK_Tracker &) *this |
| 12.1 | 13,357 | 1:57.289 | 176 | 8.63265E+06 | 666417 | 1.017E+05 | bool SlidingWindowFinder::nextcluster() *this |
| 2.3 | 12,569 | 22,674 | 1.36058E+06 | 6.75358E+06 | 17 | 57.54 | void LArG3Escale::MakeCorrection(LArCell *) *this |
| 1.6 | 10,501 | 15,723 | 1.37371E+06 | 4.23472E+06 | 11 | 70.1 | const XK_Counter *XK_Tracker::Guide(const char &, const XK_Counter *const &, const XK_Helix &, const float &) const *this |

## B.5 Jet Data with Pileup (seeded reconstruction)

### B.5.1 Profile ordered according inclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 100.0 | 278 | 12:31.289 | 1 | 14 | 751289517 | 0 | int main(int, char **) |
| 97.8 | 0.071 | 12:14.447 | 1 | 1 | 734447259 | 0 | StatusCode ApplicationMgr::nextEvent(int) *this |
| 97.8 | 2:49.667 | 12:14.447 | 1 | 217 | 734447188 | 2 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 72.0 | 0.874 | 9:00.993 | 31 | 155 | 17451391 | 7.896 | StatusCode EventLoopMgr::executeEvent(void *) *this |
| 72.0 | 11 | 9:00.990 | 31 | 930 | 17451307 | 1349 | StatusCode MinimalEventLoopMgr::executeEvent(void *) *this |
| 72.0 | 7 | 9:00.978 | 465 | 2325 | 1163394 | 3.454 | StatusCode Algorithm::sysExecute() *this |
| 32.2 | 8 | 4:02.236 | 62 | 1116 | 3907038 | 25.35 | StatusCode CaloTowerMaker::execute() *this |
| 32.2 | 1:09.771 | 4:02.153 | 62 | 3.17018E+07 | 3905703 | 4.033E+05 | StatusCode LArTowerBuilder::execute() *this |
| 15.9 | 13,630 | 1:59.683 | 176 | 8.63265E+06 | 680019 | 1.09E+05 | bool SlidingWindowFinder::nextcluster() *this |
| 15.6 | 1:57.047 | 1:57.047 | 1.32084E+07 | 0 | 9 | 93.23 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 15.2 | 2 | 1:54.340 | 31 | 527 | 3688391 | 15.5 | StatusCode LArClusterMaker::execute() *this |
| 15.0 | 61 | 1:52.842 | 31 | 19201 | 3640091 | 3634 | StatusCode LArSlidingWindow::execute() *this |
| 14.1 | 26,957 | 1:45.921 | 8.59421E+06 | 1.42704E+07 | 12 | 61.97 | double CaloTowerContainer::et(int, int) const *this |
| 7.9 | 3 | 59,615 | 62 | 434 | 961540 | 11.32 | StatusCode LArCellMaker::execute() *this |
| 7.2 | 54,337 | 54,337 | 2.67185E+06 | 0 | 20 | 161.2 | bool LArTowerBuilder::cell_segmentation(Identifier &, float &, float &, Identifier &) *this |
| 7.2 | 10 | 53,979 | 31 | 2110 | 1741279 | 106.1 | StatusCode egammaBuilder::execute() *this |
| 7.1 | 70 | 53,690 | 42 | 6174 | 1278352 | 4940 | StatusCode EMShowerBuilder::execute(egamma *) *this |
| 7.0 | 5,155 | 52,630 | 210 | 3.99424E+06 | 250620 | 5.61E+04 | CaloCellList::Tselect |
| 6.3 | 19,202 | 47,474 | 3.99424E+06 | 1.20532E+07 | 12 | 78.37 | bool CaloCellList::CellSel::operator()(CaloCell *) *this |
| 4.7 | 22,357 | 35,099 | 4.60088E+06 | 9.20176E+06 | 8 | 103.5 | void CaloTower::addcell(CaloCell *, float) *this |

### B.5.2 Profile ordered according exclusive times

| %Time | Exclusive msec | Inclusive total msec | #Call | #Subrs | Inclusive usec/call | Standard deviation | Name |
|---|---|---|---|---|---|---|---|
| 97.8 | 2:49.667 | 12:14.447 | 1 | 217 | 734447188 | 2 | StatusCode EventLoopMgr::nextEvent(int) *this |
| 15.6 | 1:57.047 | 1:57.047 | 1.32084E+07 | 0 | 9 | 93.23 | CaloTower *CaloTowerContainer::getTower(int, int) const *this |
| 32.2 | 1:09.771 | 4:02.153 | 62 | 3.17018E+07 | 3905703 | 4.033E+05 | StatusCode LArTowerBuilder::execute() *this |
| 7.2 | 54,337 | 54,337 | 2.67185E+06 | 0 | 20 | 161.2 | bool LArTowerBuilder::cell_segmentation(Identifier &, float &, float &, Identifier &) *this |
| 4.4 | 33,138 | 33,285 | 62 | 1860 | 536861 | 5.002E+05 | StatusCode LArG3CellBuilder::execute() *this |
| 4.0 | 29,819 | 29,819 | 8.25884E+06 | 0 | 4 | 78.66 | Identifier CaloCell::ID() const *this |
| 14.1 | 26,957 | 1:45.921 | 8.59421E+06 | 1.42704E+07 | 12 | 61.97 | double CaloTowerContainer::et(int, int) const *this |
| 4.7 | 22,357 | 35,099 | 4.60088E+06 | 9.20176E+06 | 8 | 103.5 | void CaloTower::addcell(CaloCell *, float) *this |
| 3.0 | 19,962 | 22,384 | 1103 | 2.49452E+06 | 20294 | 6.262E+04 | unsigned long DataObject::release() *this |
| 6.3 | 19,202 | 47,474 | 3.99424E+06 | 1.20532E+07 | 12 | 78.37 | bool CaloCellList::CellSel::operator()(CaloCell *) *this |
| 2.5 | 18,730 | 18,798 | 31 | 2112 | 606415 | 3.832E+05 | StatusCode XKaEventCollection::execute() *this |
| 15.9 | 13,630 | 1:59.683 | 176 | 8.63265E+06 | 680019 | 1.09E+05 | bool SlidingWindowFinder::nextcluster() *this |
| 3.1 | 12,975 | 23,203 | 1.36058E+06 | 6.75358E+06 | 17 | 126.7 | void LArG3Escale::MakeCorrection(LArCell *) *this |
| 1.5 | 10,629 | 11,502 | 42 | 184255 | 273880 | 2.396E+05 | void XK_Tracker::ConvertP(XK_Condition &, const XKdDetectorVisitor &) *this |
| 1.3 | 9,653 | 9,729 | 1 | 33 | 9729052 | 0 | StatusCode ApplicationMgr::initialize() *this |
| 1.8 | 9,249 | 13,420 | 4.34378E+06 | 4.34378E+06 | 3 | 49.53 | double proxim(double, double) |
| 1.2 | 8,828 | 8,828 | 4.66823E+06 | 0 | 2 | 16.55 | double CaloCell::et() const *this |
| 1.1 | 8,055 | 8,055 | 1.13268E+06 | 0 | 7 | 148.2 | void CaloTowerContainer::setMap(CaloTower *, int, int) *this |
| 1.0 | 7,555 | 7,555 | 8.25374E+06 | 0 | 1 | 31.49 | double CaloCell::eta() const *this |
| 0.9 | 6,614 | 6,614 | 7.54464E+06 | 0 | 1 | 44.74 | double CaloCell::energy() const *this |

## References


[1] The HLT/DAQ/DCS Technical Proposal, CERN/LHCC/2000-17

[2] http://proj-gaudi.web.cern.ch/proj-gaudi

[3] http://atlas.web.cern.ch/GROUPS/SOFTWARE/OO/architecture

[4] HLT Software High-Level Design Document (in preparation)

[5] http://atlas.web.cern.ch/Atlas/GROUPS/DAQTRIG/EF/documents/EFTDREventCnvSvc.pdf

[6] PESA SW group, PESA High Level Trigger selection software requirements, ATL-DAQ-2001-005

[7] EF requirements Document (in preparation)

[8] E.Frank, Private Communication.

[9] J. Baines *et al.* First Study of the LVL2-EF boundary in the high-$p_\perp$ electron/photon High Level Trigger, ATL-DAQ-2000-045

[10] M.Wielers, Private Communication.

[11] http://atlas.web.cern.ch/Atlas/GROUPS/LIQARGON/software/Reconstruction

[12] http://www.acl.lanl.gov/software ; Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation, S.Shende, A.D.Malony, R.Ansell-Bell, Department of Computer and Information Science, University of Oregon; http://www.cs.uoregon.edu/research/paracomp/tau

[13] Performance Analysis of the ATLAS Second Level Trigger Software, J.A.C Bogaerts et al., paper submitted to the Real Time Conference 2001

[14] Summary of Standard Values

[15] TAU User's Guide, version 2.9, Department of Computer and Information Science, University of Oregon, http://www.cs.uoregon.edu/research/paracomp/tau

[16] http://www.acl.lanl.gov/software, Program Database Toolkit, PDT

[17] C++ Front End, Internal Documentation, Edison Design Group, Inc., Dec 26, 2000, Version 2.45

[18] http://www-wisconsin.cern.ch/~wiedenma/Profiling

[19] http://www.acl.lanl.gov/tau → Docs → FAQ