Systematic profiling to monitor and specify the software refactoring process of the LHCb experiment

Ben Couturier
CERN, CH-1211 Geneva 23, Switzerland
E-mail: ben.couturier@cern.ch

Emmanouil Kiagias
University of Athens, Greece
E-mail: emmanouil.kiagias@cern.ch

Stefan B. Lohn
CERN, CH-1211 Geneva 23, Switzerland
E-mail: stefan.lohn@cern.ch

Abstract. The LHCb upgrade program implies a significant increase in data processing that will not be matched by additional computing resources. Furthermore, new architectures such as many-core platforms can currently not be fully exploited due to memory and I/O bandwidth limitations. A considerable refactoring effort will therefore be needed to vectorize and parallelize the LHCb software, to minimize hotspots and to reduce the impact of bottlenecks. It is crucial to guide refactoring with a profiling system that gives hints to regions in source-code for possible and necessary re-engineering and which kind of optimization could lead to final success.
Software optimization is a sophisticated process where all parts, compiler, operating system, external libraries and chosen hardware play a role in. Intended improvements can have different effects on different platforms. To obtain precise information of the general performance, to make profiles comparable, reproducible and to verify the progress of performance in the framework, it is crucial to produce profiles more systematically in terms of regular profiling based on representative use cases and to perform regression tests. Once a general execution, monitoring and analysis platform is available, software metrics can be derived from the collected profiling results to trace changes in performance back and to create summary reports on a regular basis with an alert system if modifications led to significant performance degradations.

1. Introduction

In large-scale software solutions of high energy physics (HEP), performance is critical for efficient result extraction. Software evolves quickly because of considerations of maintainability, usability, due to necessary feature implementation or urgent bug fixes. Hence, performance has often less priority, also because of missing information about software behavior. The many available profiling tools do not change the game, because it appears that performance information is not clearly and reliably available to determine the importance of optimization measures. The LHCb performance & regression (PR) project tries to address this issue for the LHCb experiment.

During its development phase, the LHCb software was constantly optimized; the profiling was however the responsibility of each developer, with no "official" profiling test suite defined and no record of the results. While this approach was effective during the framework development phase, there is no record of the evolution of software performance, nor of the current baseline. With a refactoring of the code under way, it is therefore now necessary for LHCb to put systematic profiling tests in place, in order to ensure that there is no performance degradation.
This paper gives an overview of LHCb software and the objectives of the LHCb PR project in chapter 2, the work-flow with some technical aspects in chapter 3 and briefly goes into upcoming challenges in chapter 4.

2. LHCb computing

2.1. LHCb software

The LHCb experiment software is based on Gaudi[1], a C++ framework using generic and object-oriented features of C++ for computing intensive tasks, and python for configuring and structuring modules (algorithms, tools...). Gaudi provides core services and tools for applications to hide complexity and make future development and changes more transparent for users. It is a large-scale framework and is additionally used by ATLAS, Glast, Harp and other experiments. Online and Offline applications are built on top of Gaudi. Moore is the implementation of the High-Level Trigger (HLT) to decide whether event data will be stored or not, Brunel is responsible for offline reconstruction, DaVinci is the physics analysis framework, Gauss simulates the particle transport and interaction through several detector modules using Geant4, and Boole performs the digitization.

2.2. Computing environment

The LHCb computing environment consists of computing resources of the Worldwide LHC Computing Grid (WLCG), of Cloud Infrastructure and of the HLT farm located at the experiment. Additional, virtualization is used and just recently volunteer computing has been added using Boinc. Some 100k CPUs are involved in the data processing.

2.3. Integrated Profiling

Instrumentation is a crucial advantage for profiling source-code of large-scale frameworks. Multiple profiling measures have been implemented in the Gaudi framework using the AuditorService [2] which provides an interface for executing code between events or algorithms processed.
Timing information from the operating system's process information is collected using the TimingAuditor. Likewise information can be collected using the Memory- or MemStatAuditor for changes in the memory footprint. Recent work [5] conducted by Mazurov and Couturier shows how to improve precision in profiling the event-loop using instrumentation routines of Intel's VTune™Amplifier API, which can be added using Gaudi's IntelAuditor. Another strategy is to collect information from the performance monitoring unit (PMU) of modern CPU architectures to collect information about hardware usage such as cache-misses, branch-misprediction and more, as done by Kruse and Kruzelecki [6]. Many of such kind of work has been performed to provide tools and to enable developers to profile their code. Still, systematic usage or comparative profiling has been barely observed.

2.4. Systematic Profiling

Three important aspects are crucial for *systematic profiling*. Profiles must be *comparable*, *reproducible* and *representative* to allow regression analysis and to trace back changes in performance to regions in source-code. The way to detect anomalies in performance by regression

analysis, which can then be examined for responsible regions in code like modules, algorithms, functions or other entities of source-code locations, is here called a *top-down* analysis.
Profiling should be limited to a small number of default use cases and a fixed set of reference data to facilitate finding changes in performance which are crucial for production and to point out responsible source-code regions, that are *good candidates for performance optimization*. Representative reference data are important to avoid variance due to different types of physics. This way differences in execution behavior between two revisions must originate in related source-code changes. Changing the reference set of physics events could later on be used to evaluate the needs of computing resources for upcoming data-taking periods for changing raw event information.
Furthermore, profiles must be reproducible in order to compare test configurations of executed test jobs. This affects the *job configuration*, which is given by version/platform/host information, as well as the *run configuration*, which is providing the setup of a run by an option file. As soon as tests are reproducible with reliable results evaluated through multiple runs, tests become comparable for further examinations.
Hence the LHCb PR project has the following requirements.

(i) Performance information must be centrally collected and become easily accessible. A web application as interface to support brief analysis of collected data could achieve this while additionally facilitating the propagation of issues.

(ii) Profiling is a changing subject with new interesting technologies. A solution must be flexible to include new profiling tools. Hence, information must be collected by parsing reports from profilers to put them onto the central database.

(iii) Regular execution is necessary to comply with the objective of reliable regression analysis. An automated chain of triggering, setup, execution and data collection would simplify systematic profiling attempts.

(iv) To reduce work generic ways of navigation and visualization are beneficial.

3. LHCb PR project

3.1. Work-flow

The LHCb PR project provides support to conduct systematic profiling. To fulfill the objectives and requirements, the work-flow can technically be divided into three major parts, as shown in figure 1. In (A), for reproducibility, the job and run configuration must be referenced to the collected profiling results from a central SQL database. The job and run configuration is characterized by a job description ID provided by the database, while option files are stored in the version controlled PRConfig package. Jenkins [8], a continuous integration system, is now used to manage the test configuration, trigger and submission of test jobs for a certain automation in the profiling cycle.
Before execution, as shown in (B), a wrapper is downloaded and executed to hide the profiler specific setup. While execution, the profile is gathered and stored with log-files on a distributed file system (AFS). After profiling, data handlers to parse reports and collect results are called. Then, results are inserted onto the SQL database. In the end of the chain (C), the LHCb PR web analysis interface, based on Django [7], facilitates profile comparisons and helps detecting anomalies in performance.

3.2. LHCb PR web interface

The core of the LHCb PR framework is a web interface based on Django to analyze profiles. The backbone of Django is using python to speed up development while keeping a certain amount of flexibility due to a variety of auxiliary modules available. Processing intensive tasks can be performed on the server-side while keeping the interface, on client-side, quick and smart.

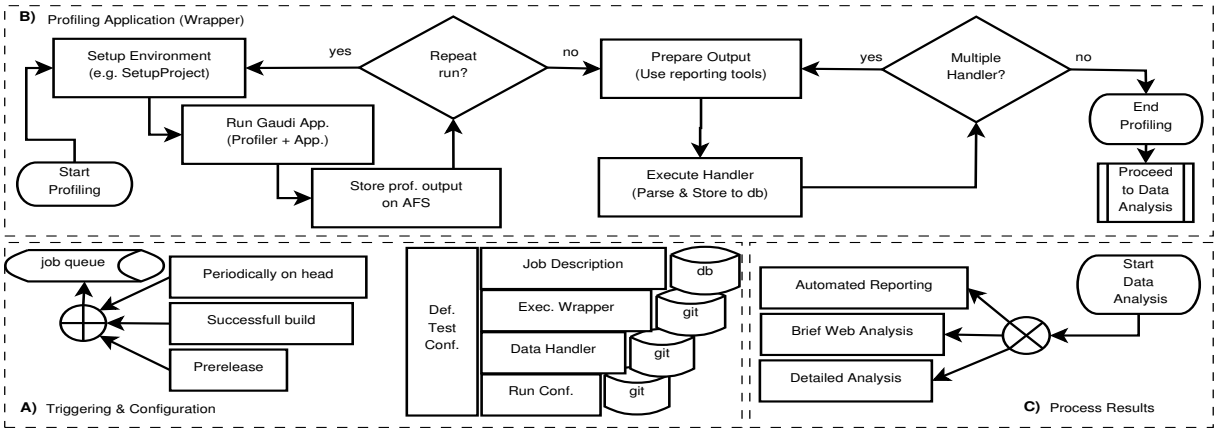

Figure 1. *The work-flow has three main parts. A) the test definition, triggering and job submission. B) the rob execution, repetition and data collection using data handlers. C) quick access to profiling results.*

The front-end can roughly be divided into two main aspects, navigation and visualization. To *navigate* easily through data, results can directly be accessed by their category, where jobs from a specific job and run configuration belong to the same *category*. A generic selection menu allows to select specific categories, which one wishes to compare. The generic menu is supplemented by a customized part, to specify profiling groups (different profiling information), attributes or filtering options, which are important to the specific visualization. Data access is also available on job level via a job table that is accessible through a list of all available categories.
Different analysis types are referring to different *visualization* methods, in which data *attributes*, generic items of performance information, are shown. Attributes can be runtime, resident memory, possibly lost memory or a more complex software metric. The Trend-Analysis, figures out the progress of attributes cross versions or revisions and has been added to observe changes in the general performance during the code evolution. Significant changes become immediately conspicuous, but requires single attributes, e.g. one of many algorithms, to be tracked. The Overview-Analysis is to show several attributes between two versions, platforms or run configurations. Both, Trend and Overview, show entries with their statistical variance. To get more precise information about the distribution of attribute values around their average the Basic-Analysis can be used.
The Visualization uses ROOT histograms, which are created with the python interface pyRoot to simplify the access, or Google Charts, currently created on the client-side with javascript. The Basic-Analysis is using ROOT histograms with the attributes dimension as x-axis and normalized about the amount of entries on the y-axis, e.g. in figure 4. Visualizations like job tables, trends, treemaps or overviews are implemented using Google Charts, for which data are preprocessed on the server-side.

3.3. Job distribution and triggering

To facilitate *regular profiling* in a series of equal tests to permit statistical evaluations, Jenkins helps managing the job distribution to test hosts. The test configuration, which is the composition of the job and run configuration, can be prepared by creating generic parametrized jobs on the Jenkins UI. There, Preprocessing, e.g. compiling packages with specific compiler flags or using different revisions of packages (software modules) within a specific version, can be added. Jenkins organizes the inclusion of further hosts and it prevents interferences between multiple jobs running on the same machine by limiting job slots.
A cron-jobs like plugin in Jenkins can be used for triggering induced by the release or build cycle.

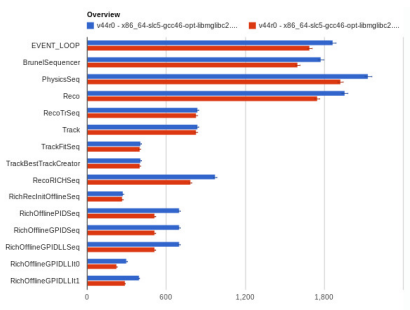

Figure 2. *Overview analysis of Brunel to get a fast impression of attributes behavior to others and cross versions, platform or differing configurations (options).*

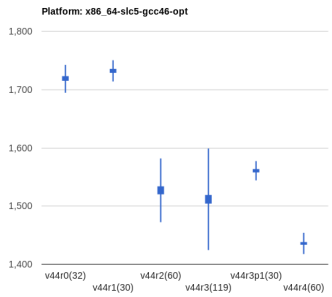

Figure 3. *Trend analysis (runtime [ms] ×versions) of Brunel to monitor changes of specific attributes cross versions to observe the general performance.*

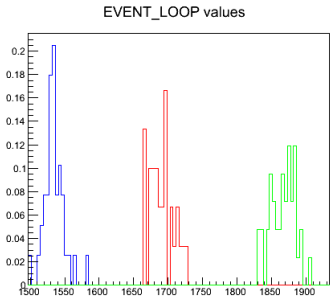

Figure 4. *Shows runtime distribution ([ms] per event) using different math libraries. (blue) Intel, (red) elder libm, (green) updated libm of a new glibc version.*

This ensures that from triggering to execution and data collection, no human intervention is necessary. This simplifies data collection and opens the way to point out reliable and significant changes in the software profile.

3.4. Data collection

Data collection is performed by *data handlers*, which are defined for distinct types of profile reports or which can provide additional information like a comment, a profile class or an AFS path per job. There are two different kinds of routines to transfer collected information. One to save numbers or strings to be stored onto the database and a second to upload files which contain further performance information and could later become available from within the interface. The post-processing of a profiling job must prepare parsable reports. The LHCb PR Handler project hosts the various data handlers to parse and collect the data.

3.5. Test cases

Use cases are important to trace performance changes back to the evolving algorithms. *Test cases* are based on *default use cases*, and try to be a best approximation to the production usage. Still, the sophisticated computing environments can influence test runs on a non-predictive non-deterministic way, which hampers conclusions from regression analysis. Currently, the highest priority is to find software related issues, that can be addressed or have to be taken into account for upcoming decisions in resource allocation.

A few default cases for Brunel have been quickly evaluated, defined and are running now on a regular basis after each successful nightly build. LHCb PR has demonstrated its importance already by observing simple timing information obtained by the TimingAuditor. The Overview-Analysis gives a general idea on how single algorithms runtime falls into account, like shown in figure 2. For time consuming algorithms, significant changes become immediately clear by observing the trend information cross versions as shown exemplary in figure 3. After observing performance degradation, further tests can be defined to use more precise profilers, like the IntelAuditor, for more details. One real example shows the changes found in the external math library, as shown in Figure 4. Such information needs to be extracted before a release is done.

Unfortunately, the HLT framework (Moore) can not simply be reduced to a few common default cases, what makes it more difficult to trace back performance issues using a top-down analysis.

The further complications are:

(i) The computing environment at the HLT farm is highly sophisticated with processes cloned to run multiple of them on a node and with events coming from the BufferManager.

(ii) Single or a few default cases, as required, are not available because of constantly changing trigger configuration keys (TCKs), which define the algorithms involved.

The first problem can only be addressed if developers agree to provide a representative standard configuration. Since this is currently not available, no enhancements can here be made. The second problem can partially be addressed by splitting one use case into multiple test cases and focusing these cases on specific regions in the source-code. This also simplifies tracing back performance issues to source-code locations and to provide separated development efforts with information. Finally the Overview-Analysis was adapted to be able to compare different run configurations, to evaluate how separate test cases differ.

4. LHCb PR and beyond

4.1. Profiling accuracy

In the current state, the web-interface does not provide a resolution beyond algorithm level. More detailed information is only accessible via the collected profile information stored on AFS. The resolution of the web-interface is limited for several reasons.

(i) Systematic profiling with a top-down point of view on performance is to *validate performance*, but not to point to specific lines in code. Hence, LHCb PR currently informs the developer groups about general performance and allows to examine how a specific implementation contributes to the overall performance. Additionally, developers are provided with profiles for detailed analysis for the given default use cases if this is requested. Competing with the variety of good visualization and analysis tools of highly sophisticated profilers would however not be a feasible approach.

(ii) Default cases are interesting to *estimate the general performance expectations*, but not to analyze the efficiency of uncommon use cases. Systematic profiling will not completely replace customized profiling by single developers, since uncommon use cases are not covered by the systematic profiling approach.

(iii) Regression analysis is gathering, e.g. in the case of thousands of algorithms of a Gaudi application, already big amounts of data with ordinary information. One reason to limit the amount of performance data is, because of the already massive amount of collected data. Statistical evaluations are currently more in focus to provide *reliable and precise information*. Additional limitations are given by the fact that the general performance is characterized by a selection of measurements. Not every measurement will and can be provided.

The higher precision leads already now to further issues, as for instance from the unpredictable side effects of recent hardware features. They can influence runtime on an unreproducible way, e.g. by using frequency scaling or automated over-clocking. On the one hand, these aspects can now precisely be determined, but on the other hand, needs to be neutralized during regular profiling to keep tests reproducible and to detect software performance anomalies.

4.2. Complementary information

Still, it could be reasonable to increase the resolution to function level. A call stack of functions with corresponding runtime would enable us to see if algorithm were using lazy initialization. This is important in particular for Gaudi, which allows a flexible order in which algorithms are started. E.g. tracking could be triggered by the first algorithm which requests these information.

Later, tracks would only be read. This aspect makes algorithm vary in their runtime. This appears currently as unreasonable change regarding performance of single algorithms.
Furthermore, gathering performance data could be extended by additional information. Correlating specific physics analysis with runtime is frequently performed, but not included into LHCb PR. Likewise different information, like software quality metrics, could be correlated to runtime behavior. Subsequently it would be possible to estimate the impact of design decisions and to find methodologies for new concepts. Open questions could be answered as knowledge base for further developments. Questions like how exactly the call-stack depth causes more cache-misses within a multi-threaded application that is sharing cache among threads, could then be quantified by concrete numbers.

5. Conclusions

Using a flexible, extensible and customizable platform to collect and summarize profiling results enables the LHCb collaboration to systematically compare profiles. The LHCb PR project has already demonstrated to be highly valuable. Performance due to changing software and advancing technology is partially examined and issues could be addressed. Since implementing instrumentation is in many aspects already performed and since many profilers can be applied for data collection and as Django and Jenkins is reducing the necessary work to set up a performance monitoring system, the remaining effort is limited to setup a framework for performance and regression analysis. Still, improvements have to be discussed and put into perspective, but due to the easy adaptivity of the web interface and testing framework, more and more fields for application, like including software metrics, become tangible.

References

[1] G. Corti, M. Cattaneo, P. Charpentier, M. Frank, P. Koppenburg, P. Mato, F. Ranjard, S. Roiser, I. Belyaev and G. Barrand, *"Software for the LHCb experiment"*, IEEE Transactions on Nuclear Science, vol. 53, nb. 3, P.1323-1328, 2006

[2] P. Mato and others, *"Status of the GAUDI event-processing framework"*, Intl. conference on computing in high energy and nuclear physics, Beijing, China, 2001

[3] Hegner, B.; Mato, P.; Piparo, D., *"Evolving LHC data processing frameworks for efficient exploitation of new CPU architectures"*, Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), IEEE, P.2003-2007, 2012

[4] M. Frank, C. Gaspar, E. v Herwijnen, B. Jost, N. Neufeld, and R. Schwemmer, *"Optimization of the HLT Resource Consumption in the LHCb Experiment"*, J. Phys.: Conf. Ser. 396 012021, 2012

[5] A. Mazurov and B. Couturier, *"Advanced Modular Software Performance Monitoring"*, J. Phys.: Conf. Ser. 396 052054, 2012

[6] D. F. Kruse and K. Kruzelecki, *"Modular Software Performance Monitoring"*, J. Phys.: Conf. Ser. 331 042014, 2011

[7] "Django is a high-level Python Web framework", url: https://www.djangoproject.com/

[8] "Jenkins, An extendable open source continuous integration server", url: https://www.jenkins-ci.org/