Journal of Physics: Conference Series **396** (2012) 052054 doi:10.1088/1742-6596/396/5/052054

Advanced Modular Software Performance Monitoring

A Mazurov[1,2,3] **and B Couturier**[1]

[1] CERN, European Organization for Nuclear Research, Geneva, Switzerland
[2] University of Ferrara, Ferrara, Italy
[3] Institute for Nuclear Research, Troitsk, Russia

E-mail: alexander.mazurov@cern.ch

Abstract. The LHCb software is based on the Gaudi framework, on top of which are built several large and complex software applications. As the LHCb experiment is now in the active phase of collecting and analyzing data, performance problems arise in various parts of the software, from the High Level Trigger (HLT) programs to data analysis frameworks. It is not easy to find hotspots in the code — only specialized tools can help to understand where CPU or memory usage are not reasonable. There exist many performance analyzing tools, but the main problem is that they show reports in terms of class and function names and such information usually is not very useful — the majority of algorithm developers use the Gaudi framework abstractions and usually do not know about functions which lie at the lower level. We will show a new approach which adds to performance reports a higher abstraction level based on knowledge of framework architecture and run-time object properties. A set of profiling tools (based on Intel® VTune™Amplifier XE) and visualization interfaces has been developed and deployed.

1. Introduction

In LHCb, as in all High Energy Physics (HEP) experiments, complex software is used to process the data recorded by the detectors. Performance is an essential characteristic of this software, especially when dealing with the High Level Trigger (HLT): its role is to filter events coming from the hardware based trigger in order to identify those with interesting physics, and to write them to disk in real-time. The number of events processed per second (event rate) is therefore one of the crucial characteristics of the HLT, as it has to keep up with the data rate delivered by the hardware triggers (10^6 events per second) in order to avoid data loss. To reach such high throughput, the processing is performed on many nodes in parallel by highly optimized algorithms. In order to optimize the algorithms, and to keep track of the evolution of the event rate when changes are applied to the HLT, it is necessary to measure the overall peformance of the code but also to understand which algorithms are costly in term of CPU and memory.

In this paper our focus is on the analysis of frequency and duration of function calls in algorithms (this type of analysis is commonly named CPU profiling). Profiling helps to identify parts of the code that take a long time to execute. In performance analysis, those places often are referenced as hotspots. Obviously, hotspots affect the event rate of event processing software. So, one of the main goals of profiling HEP software is to point out to application developers the places in the code that need to be tuned to increase the event rate.

The first study on CPU profiling at LHCb was carried out by Daniele Francesco Kruse and Karol Kruzelecki in their work Modular Software Performance Monitoring [1]. They conclude

that instead of profiling the application as a whole it would be better to divide it into modules and profile those modules separately. In general terms, a module can be defined as an applications structural component that is used to group logically related functions. Grouping performance results by module allows a better insight into where the performance issues are coming from. Since each module is under the responsibility of a specific developer, the provided reports can be delivered to the right person. For example, in the Gaudi [2] framework at LHCb each algorithm used for event processing is such a module which can be profiled independently. More details on Gaudi will be given in Section 3.1.

This design principle was first implemented in a set of profiling tools based on perfmon2 [3] library. These tools have several drawbacks. First, the produced analysis reports used the hardware event counters metrics. Only developers with a good knowledge of the hardware could read and interpret those reports. Since the major part of developers in HEP are physicists, the number of users of those tools are very low. Second, since the current tools do not use the counters multiplexing feature of perfmon2 library, the target program should be run several times to collect all required hardware counters. As a result, the profiling time is significantly increasing.

To fill some of the the gaps of the previous tools we created the Gaudi Intel Profiling Auditor. This profiling tool uses the same module principle that was described in [1], but is based on Intel® VTune™Amplifier XE [4]. VTune™Amplifier XE is the newer performance profiling tool, that provides better functionality than perfmon2 library.

In the next section we briefly review Intel® VTune™Amplifier XE . Then we discuss how the Gaudi Intel Profiling Auditor can integrate VTune™Amplifier XE to the Gaudi framework and show examples of using those tools to profile LHCb's HLT.

2. Intel® VTune™Amplifier XE

In this section we give an overview of the modern performance profiling tool Intel® VTune™Amplifier XE , describe its basic features and analysis reports.

2.1. Overview

Intel® VTune™Amplifier XE is a commercial application for software performance analysis that is available for both Linux and Windows operating systems. VTune™Amplifier XE belongs to the runtime instrumentation class of profiling tools. This means that the code is instrumented before execution and the program is fully supervised by the tool. A target application can be profiled without any modification of the codebase.

Intel® VTune™Amplifier XE has various kinds of code analysis including hotspot analysis, concurrency analysis, locks and waits analysis. In our tools we use a hotspot analysis based on the user-mode sampling feature of VTune™Amplifier XE . User-mode sampling allows to profile a program by exploring a call stack of a running program and produce one simple metric — amount of time spent in the function.

The amount of time spent in a function (CPU time) is calculated by interrupting a process and collecting samples of call stacks from all active threads. CPU time value is calculated by counting the number of a function's appearances at the top of a call stack. This means that stack sampling is a statistical method and does not provide a 100% accurate measurement. However, for a large number of samples the sampling error does not have a serious impact on the accuracy of analysis. More details about sampling accuracy will be provided in section 2.2.

VTune™Amplifier XE also supports the hardware event-based sampling and provides advanced metrics based on event counters inside a processor. Reports that use those metrics require knowledge of hardware architecture unlike the user-mode sampling reports that can be understood by any application developers. Furthermore, while the user-mode sampling can be performed on any 32 and 64-bit x86 based machine, the hardware event-based sampling is

targeted only for a specific Intel® microarchitecture and requires a special driver to be installed on the operating system. The advantage of the hardware event-based sampling that it can be used for fine tuning of algorithms in places where the user-mode sampling could not point out the reasons for the hotspot.

The goal of our profiling tool is to provide analysis reports to a wider audience of software developers and, therefore, for implementation we chose the user-mode sampling method over hardware-mode sampling.

2.2. Sampling interval

The sampling interval is an important parameter of the user-mode sampling method. It can impact on results accuracy and on total profiling time. Intel® recommends to use a 10 ms interval. Using this value the average overhead of the sampling is about 5% in the most applications. The minimum sampling interval value depends on the operating system. For example, a 10 ms interval is the minimum value for the old Linux kernel 2.4, whereas 1 ms is the minimum value for the modern Linux $\geq$ 2.6 kernels.

To determine an appropriate sampling interval, consider the duration of the collection, the speed of your processors, and the amount of software activity. For example, if the duration of sampling time is more than 10 minutes, consider increasing the sampling interval to 50 milliseconds. This reduces the number of interrupts and the number of samples collected and written to disk. The smaller the sampling interval, the larger the number of samples.

2.3. Tools

VTune™Amplifier XE has two major interfaces — a command-line tool *amplxe-cl* and a Graphical User Interface tool *amplxe-gui*. *Amplxe-gui* generally plays a role of analysis results presenter, but can also be used as a wrapper to the command-line tool. *Amplxe-cl* is used to execute the profiling supervisor with appropriate parameters. The second important function of *amplxe-cl* is to export CPU usage reports to CSV text format. This feature allows to use collected data not only inside VTune™Amplifier XE , but also in external user applications.

2.4. Profiling reports

In this section we review essential profiling reports that are available in VTune™Amplifier XE . These reports can be obtained either from *amplxe-gui* or *amplxe-cl tool*, but for short we present only GUI screenshots.

An ordered functions CPU time usage report is a basic report of almost at all performance profilers (Figure 1).

Hotspots - Hotspots

Analysis Target | Analysis Type | Summary | Bottom-up | Top-down Tree

/Function /Call Stack	CPU Time	Module
PatForwardTool::fillXList	22.483s	libPatAlgorithms.so
FastVeloTracking::makeSpaceTracks	16.963s	libFastVelo.so
Tf::HitCreatorGeom::OTModule::loadHits	8.465s	libTfTools.so
tc_new	8.157s	libtcmalloc.so.0.1.0
Tf::TStationHitManager<PatForwardHit>::createHit	7.374s	libTfTools.so
PatForwardTool::fillStereoList	6.875s	libPatAlgorithms.so
PatFwdTool::xAtReferencePlane	5.817s	libPatAlgorithms.so
std::__introsort_loop<__gnu_cxx::__normal_iterator<PatF	4.530s	libPatAlgorithms.so

Figure 1. Functions CPU Time report. The first column contains function names. The second column is a CPU time usage and the last column contains the names of the shared libraries where the functions are defined.

VTune^{TM}Amplifier XE provides many grouping options:

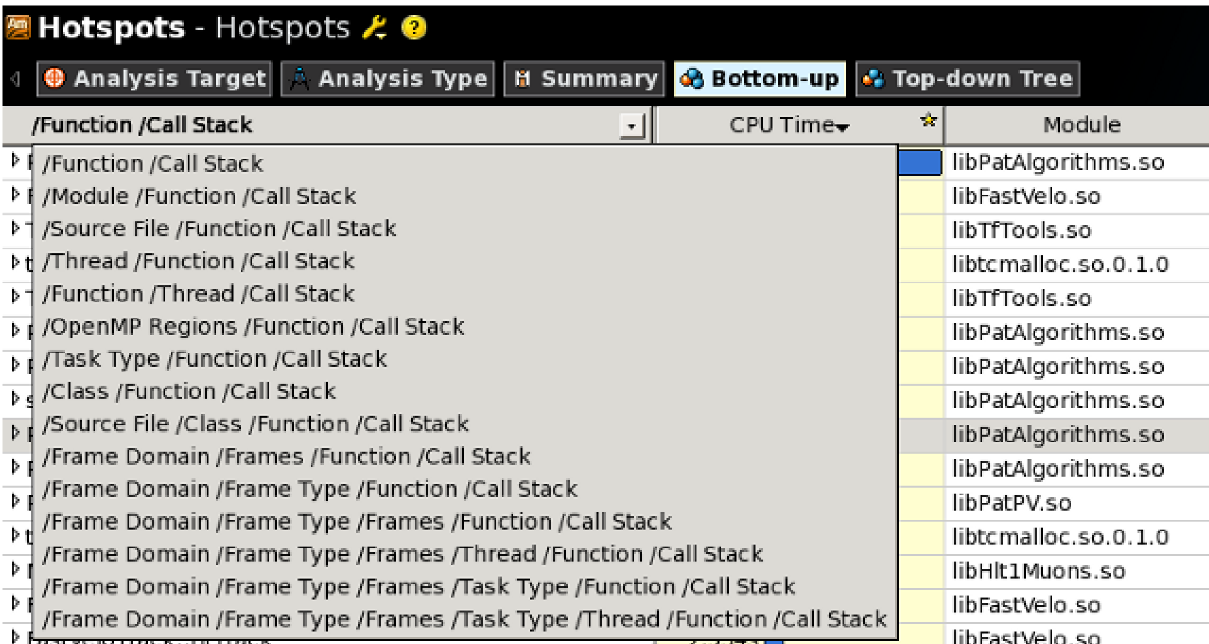

Figure 2. Various grouping options.

Example of grouping by shared library:

Hotspots - Hotspots

Analysis Target | Analysis Type | Summary | Bottom-up | Top-

/Module /Function /Call Stack	CPU Time
libPatAlgorithms.so	56.804s
libFastVelo.so	38.316s
libTfTools.so	22.738s
libPatPV.so	11.689s
libtcmalloc.so.0.1.0	11.553s
libTrackFitEvent.so	9.607s
libHlt1Muons.so	7.725s
libGaudiKernel.so	7.087s

Figure 3. Shared libraries CPU time report. First column contains a name of shared library.

The striking feature of VTune$^{\text{TM}}$Amplifier XE is an ability to filter data based on a selection in the timeline. This feature does not exists in other popular profilers:

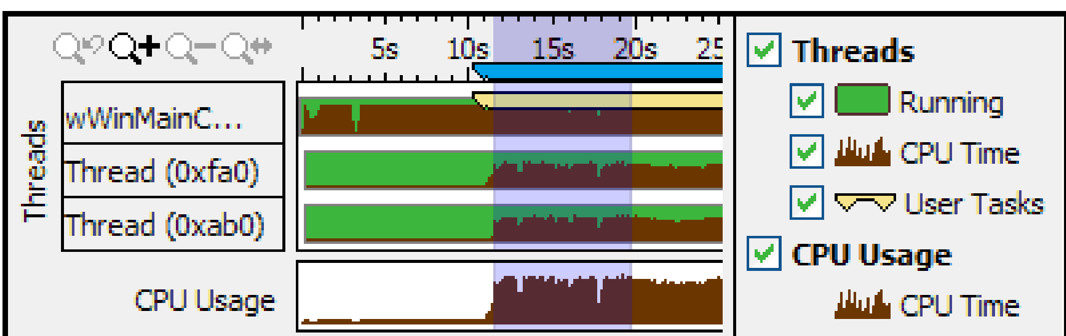

Figure 4. Filter data on a selection in timeline.

CPU usage by code line can be created if a target application was compiled with debug symbols:

Source | Assembly

Line	Source	CPU Time
970	`info() << format( "x%7.3f y%7.3f R%7.3f dSin%7.3f, MC: %7.3f %7.3f R%7.3f`	
971	`x, y, sqrt( x*x + y*y ), dSin,`	
972	`xMc, yMc, rMc, (*itH)->distance( xMc, yMc ) );`	
973	`printCoord( *itH, ":" );`	
974	`}`	
975	`if ( fabs( dSin ) > maxDSin ) continue;`	0.050s
976		
977	`(*itH)->setZ( sensor->z( x, y ) );`	0.100s
978	`(*itH)->setPhiWeight( rPred );`	0.059s
979	`if ( 0 > firstSensorWithHit ) firstSensorWithHit = sensor->number();`	0.020s
980	`goodPhiHits[module].push_back( *itH );`	1.651s
981	`}`	

Figure 5. CPU time usage by code source line.

2.5. Detecting code dependency

Besides finding hotspots, another useful function of the profiling tool is to reveal the code dependencies. Usually HEP applications have a lot of lines of code and were developed by many people during a long period of time. Therefore, determining relations between parts of code is very difficult. Since VTune^{TM}Amplifier XE has a top-down tree report of functions calls (Figure 6.), we can determine the code dependency in the application and see CPU usage in the call chain.

Call Stack	CPU Time:Total	Module
▽Selection::Line::Stage::execute	96.3%	libSelectionLine.so
Algorithm::isEnabled	0.0%	libGaudiKernel.so
StatusCode::~StatusCode	0.0%	libGaudiKernel.so
▽GaudiAlgorithm::sysExecute	96.3%	libGaudiAlgLib.so
▽Algorithm::sysExecute	96.3%	libGaudiKernel.so
▷Gaudi::Guards::AuditorGuard::AuditorGuard	0.4%	libGaudiKernel.so
▽GaudiSequencer::execute	95.4%	libGaudiAlgLib.so
▽GaudiAlgorithm::sysExecute	95.4%	libGaudiAlgLib.so
▽Algorithm::sysExecute	95.4%	libGaudiKernel.so
▷L0DUFromRawAlg::execute	3.0%	libL0DU.so
▷LoKi::HltUnit::execute	90.6%	libLoKiTrigger.so
▷LoKi::L0Filter::execute	0.4%	libLoKiHlt.so

Figure 6. Top-down tree report.

3. Gaudi Intel Profiling Auditor

In the previous section we show that Intel® VTune^{TM}Amplifier XE is a powerful performance profiling tool. What are the disadvantages of the tool? Why do we need to implement something else? In the current section we are going to answer those questions. First, we talk about the Gaudi framework and then show how the Gaudi Intel Profiling Auditor can enhance VTune^{TM}Amplifier XE reports.

3.1. Gaudi

3.1.1. Overview. Gaudi is a C++ software framework used to build event data processing applications using a set of standard components. Gaudi is a core framework used by several HEP

experiments, in particular LHCb and ATLAS at LHC. All of the event processing applications, including simulation, reconstruction, high-level trigger and analysis are based on this framework. By design, the framework decouples the objects describing the data and those implementing the algorithms. Due to this design, developers can concentrate only on physics related tasks in algorithms and usually do not care about other parts of the framework.

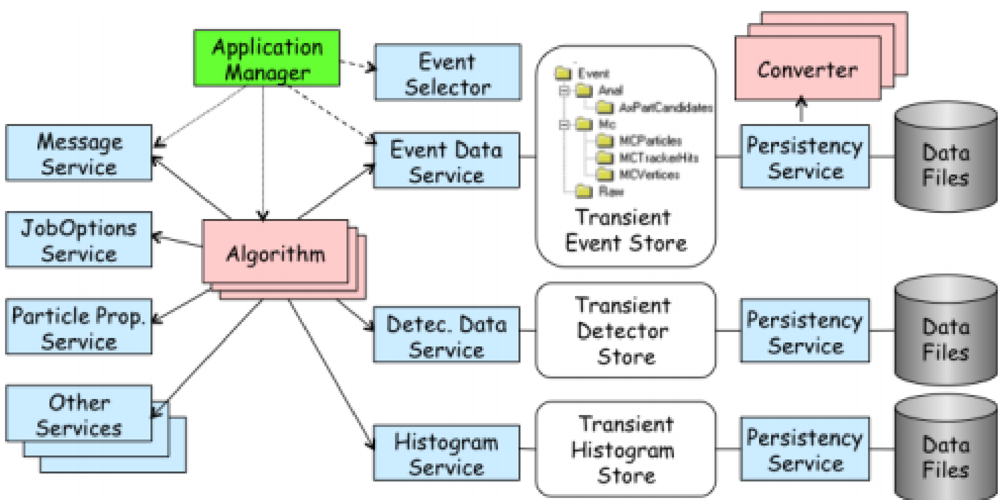

Figure 7. Gaudi Architecture (as described in [2]). Applications are made by composing sequences of Algorithms and adding specific Services and Tools.

The Gaudi framework is a highly customizable framework. Any component of the system can be configured by user options.

3.1.2. Gaudi Auditors. The Application Manager is one of the major components of Gaudi. It takes care of instantiating and calling algorithms. A supplement to this component is the Auditor Service that enable to add auditors to a Gaudi application. The auditor is a set of user functions that are called on some workflow events in the Manager. For example, we could add custom action that is called when the Manager wants to execute some algorithm or when an algorithm is finished. There are many different events types and we can add as many auditors as needed. In other frameworks and programming languages, this type of functions are often referenced as callback functions.

In the following section we show how we can use an auditor to build a profiling tool.

3.2. Profiling Auditor

3.2.1. Objectives. A Gaudi application can be profiled by VTune™Amplifier XE without any modifications of the codebase. This tool can collect any data about CPU consumption in code lines, functions, classes, shared libraries, threads, but it has one disadvantage. VTune™Amplifier XE knows nothing about Gaudi framework's algorithms. However, algorithms are the central point of any framework application, all major event processing occurs there. In principle, a general task of framework users is just to write algorithms that solve a problem and usually nothing more. So, if the profiler could generate a report that can group functions CPU usage by algorithm then application developers could look to the profiling result

from a new point of view. This point of view can help to reveal previously invisible hotspots. In order to provide such report the Gaudi Intel Profiler Auditor was developed.

3.2.2. User API of Intel® VTune™ Amplifier XE . Each Gaudi algorithm has a name that is assigned to an algorithm at run-time. VTune™Amplifier XE , in turn, is supplied with a C library that allows to import those names to the target report. In order to use the library from user applications, the public User API is provided in VTune™Amplifier XE . The API enables to control the data collection process and set marks during the execution of the code. The possibility to mark code regions at runtime is the striking feature of our new profiling tool, because a CPU usage in the region between algorithms start and finish points is what we need for the report that group functions by algorithm. Event API is a part of the User API that is in charge of marking.

_itt_event __itt_event_create(const __itt_char name, int namelen);
: Create a user event type with the specified name. This API returns a handle to the user event type that should be passed into the following APIs as a parameter. The namelen parameter refers to the number of characters, not the number of bytes.

int __itt_event_start(__itt_event event);
: Call this API with an already created user event handle to register an instance of that event. This event appears in the Timeline panel display as a tick mark.

int __itt_event_end(__itt_event event);
: Call this API following a call to __itt_event_start() to show the user event as a tick mark with a duration line from start to end. If this API is not called, the user event appears in the Timeline pane as a single tick mark.

3.2.3. Implementation. An auditor is a good component for implementing the required profiling tool. In this case, we do not need to modify the algorithms code and need only to write two callback functions: at algorithm start and finish. In order to generate the target report those functions need to call Event API functions of VTune™Amplifier XE .

An appropriate auditor was created and named Gaudi Intel Profiling Auditor. It was deployed to the GaudiProfiling package of Gaudi framework as a shared library. Below we show how this profiling tool marks regions and what reports can be generated.

Gaudi has a special type of algorithms — Sequence. Each instance of Sequence can execute other algorithms or sequences. So, an application's event loop could have not only a flat but also a tree structure. Moreover, the same algorithm instance can occur in different sequences. Therefore, it was decided that an algorithm's region between its start and finish should be marked by the branch identifier. In this case, we get more detailed information about usage of the algorithm in the application. A branch identifier is constructed from an algorithm name and its parents in the sequence tree. For example, let's profile an application that has the following sequence tree:

```
Hlt
    HltDecisionSequence
        Hlt1
            Hlt1DiMuonHighMass
                Hlt1DiMuonHighMassFilterSequence
                    Hlt1DiMuonHighMassStreamer
                        FastVeloHlt
                        MuonRec
                        Velo2CandidatesDiMuonHighMass
```

```
        GECLooseUnit
            createITLiteClusters
            createVeloLiteClusters
    Hlt1DiMuonHighMassL0DUFilterSequence
        L0DUFromRaw
        Hlt1DiMuonHighMassL0DUFilter
```

In VTune^{TM}Amplifier XE the report that use information on marked regions can be obtained by choosing the "Task Type / Function / Call Stack" grouping options as seen on Figure 8.

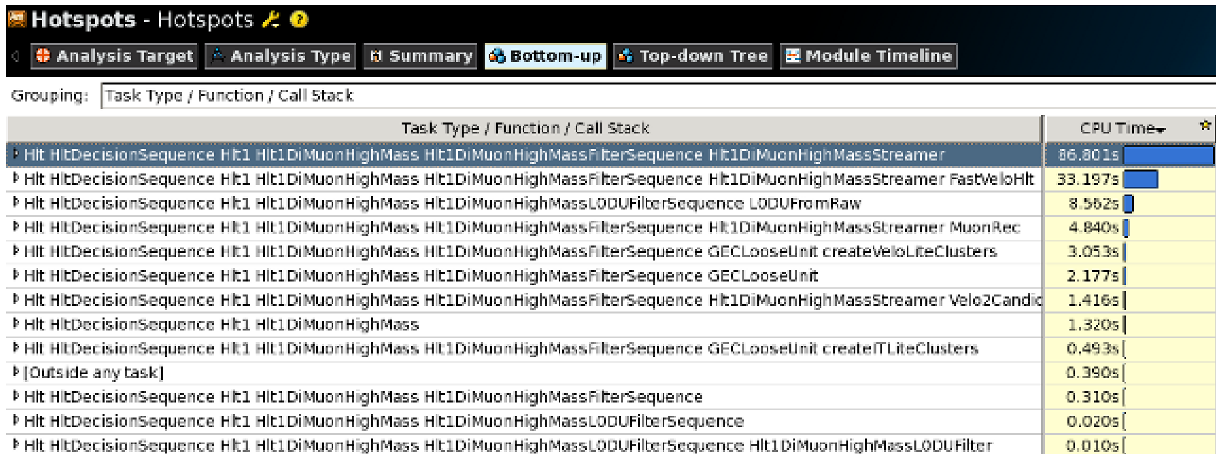

Figure 8. Group and order CPU usage by branch identifier.

For example, the selected branch identifier *"Hlt HltDecisionSequence Hlt1 Hlt1DiMuonHighMass Hlt1DiMuonHighMassFilterSequence Hlt1DiMuonHighMassStreamer"* in the report on Figure 8 is constructed from the names of algorithms that were executing when the VTune^{TM}Amplifier XE supervisor sampled a call stack. Each algorithm name in the branch is separated by the space.

For each branch we could see a CPU usage by function (Figure 9):

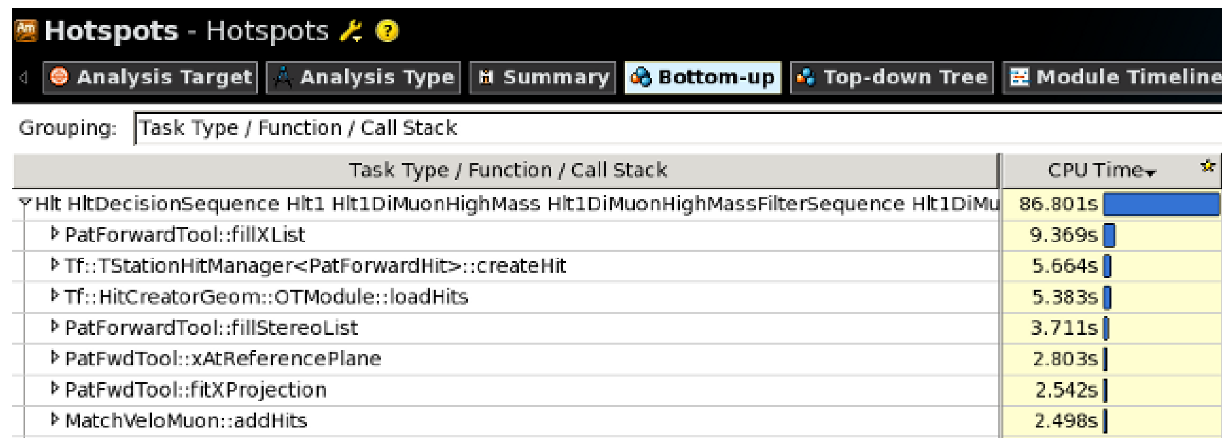

Figure 9. Group and order CPU usage by branch identifier.

On Figure 9 we see the functions's CPU usage in the algorithm *Hlt1DiMuonHighMassStreamer* in the branch *"Hlt HltDecisionSequence Hlt1 Hlt1DiMuonHighMass Hlt1DiMuonHighMassFilterSequence"*.

As can be observed the main goal was achieved — we get the report that groups function CPU usage by algorithm. So, the next step is only to interpret profiling results by application developers and, if needed, to tune algorithms.

In addition to reports on algorithms, in the Gaudi Intel Profiling Auditor were added options that allow to skip unimportant regions of the code during profiling. Information about functions in those regions is not collected and, as a result, we get clearer reports and a decrease of total profiling time. For example, usually time critical processes happens in the event loop. Thus, initialization and finalization phases are not interesting for developers. Due to this, the auditor has options that trigger the start of profiling on the first event in the event loop and stop it after the last event.

3.2.4. Usage example. For user convenience the command-line tool *intelprofiler* that is wrapper of the *amplxe-cl* was created . It does the dirty work of initializing the environment and setting the common parameters of profiling. For example, we can start a profiling by executing the simple shell command:

```
$> intelprofiler -o /path/to/collected/data/  gaudi_program.py
```

where the option -o points to the directory where to store collected data.

Generally a Gaudi program is configured from option files that are written in Python language. To enable the profiler we need to add at least the following three lines:

```
from Configurables import IntelProfilerAuditor
profiler = IntelProfilerAuditor()
AuditorSvc().Auditors +=  [profiler]
```

At the first line we imported the auditor library. Then instantiated the profiler and at the third line added the auditor to the Auditor Service component.

If you would like to skip unnecessary events in the event loop you need to add the following lines:

```
profiler.StartFromEventN = 5000
profiler.StopAtEventN = 15000
```

where we collect profiling data only for events in the range between 5000 and 15000.

From all above we can see that Gaudi Intel Profiling Auditor gives to application developers a full control on the profiling process.

4. HLT Profiling Examples

In the previous section we demonstrated how Gaudi Intel Profiler Auditor can assist in profiling Gaudi applications. The original motivation for creating this auditor was a profiling of HLT applications of the LHCb experiment. As was stated in the introduction, trigger programs are most sensitive to the event processing time. Therefore, a performance profiling is an essential tool in the hands of trigger's applications developers. In this section we show three examples of using VTune^{TM}Amplifier XE and Gaudi Intel Profiling Auditor to profile the Moore application — a Gaudi based HLT framework at LHCb.

4.1. Memory Allocation Functions

In the first example we profile a Moore program twice. The first time a program was executed with the standard memory allocation function *operator new* from libstdc++ library:

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module
PatForwardTool::fillXList	23.460s	libPatAlgorithms.so
FastVeloTracking::makeSpaceTracks	19.788s	libFastVelo.so
operatornew	18.696s	libstdc++.so.6
Tf::HitCreatorGeom::OTModule::loadHits	10.870s	libTfTools.so
Tf::TStationHitManager<PatForwardHit>::createHit	7.981s	libTfTools.so

Figure 10. Hotspot functions in the Moore application with the standard memory allocation function.

and the second time it was executed with the memory allocation function *tc_new* from tcmalloc library developed by Google® [5]:

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module
PatForwardTool::fillXList	22.483s	libPatAlgorithms.so
FastVeloTracking::makeSpaceTracks	16.963s	libFastVelo.so
Tf::HitCreatorGeom::OTModule::loadHits	8.465s	libTfTools.so
tc_new	8.157s	libtcmalloc.so.0.1.0
Tf::TStationHitManager<PatForwardHit>::createHit	7.374s	libTfTools.so

Figure 11. Hotspot function in the Moore application with the memory allocation functions from tcmalloc library.

The figures indicates that *tc_new* function is twice faster than *operatornew*. Moreover, the total application time reduction by 5% was observed if we replace standard allocation functions with function from tcmalloc library.

4.2. Measuring Profiling Accuracy

To check the CPU time measurement accuracy we compared the results obtained by the Gaudi Intel Profiler Auditor and by the Gaudi Timer Auditor. The Timer Auditor proceed in the same way as the Profiler Auditor — it calculates the difference between the algorithms finish time and the time at the start of the algorithm. Unlike the Gaudi Intel Profiler Auditor, the Timer Auditor calculates the exact time spent in the algorithm. So, we can assume a CPU time observed by the Timer Auditor as a reference value. The limitation of the Timer is that it creates reports only for algorithms times and could not provide results for a low level of granularity (for functions or code instructions). Therefore, only the algorithms CPU times were compared.

Since the VTune Amplifier XE instruments the code before execution, the absolute CPU time measured by the Profiler can differ from the time measured by the Timer auditor. But the time distribution of all algorithms should stay the same in both auditors. So, for the test we took a real HLT application and run it twice. The first time we used the Timer auditor and the second time the Gaudi Timer Auditor was used. Then we selected five hotspot algorithms and calculated their time distribution relative to the top hotspot algorithm. The process was repeated three times with different numbers of events: 10 (Table 1), 100 (Table 2) and 1000 events (Table 3):

Table 1. 10 events

Algorithm name	Timer (%)	Profiler (%)	**Difference**
L0Muon	100	100	-
Hlt1TrackAllL0Unit	63.71	63.571	**0.139**
FastVeloHlt	33.065	7.143	**25.922**
L0Calo	8.065	0	**8.065**
HltPVsPV3D	4.032	0	**4.032**

Table 2. 100 events

Algorithm name	Timer (%)	Profiler (%)	**Difference**
L0Muon	100	100	-
Hlt1TrackAllL0Unit	36.985	42.353	**-5.368**
FastVeloHlt	29.648	28.235	**1.413**
L0Calo	7.94	15.294	**-7.354**
HltPVsPV3D	2.613	4	**-1.387**

Table 3. 1000 events

Algorithm name	Timer (%)	Profiler (%)	**Difference**
L0Muon	100	100	-
Hlt1TrackAllL0Unit	35.872	35.147	**0.725**
FastVeloHlt	29.648	28.235	**1.413**
L0Calo	30.478	29.736	**0.742**
HltPVsPV3D	2.491	2.25	**0.241**

As expected, our test shows that the hotspot algorithms are the same in both auditors and the accuracy of the CPU time distribution measured by the Profiler is increasing while increasing the number of events. As a result, we can be confident that the Profiler can identify the hotspots with the high precision.

4.3. Custom reports

In the second example is demonstrated how custom reports can be created. Basic profiling reports can be picked up in VTune^{TM}Amplifier XE , but if a custom report is required then a user tool needs to be created . This application can get the CPU usage data from VTune Amplifier XE by using its export function. For example, if we export CPU Time data that is shown on Figure 8 then the following pie chart report can be created.

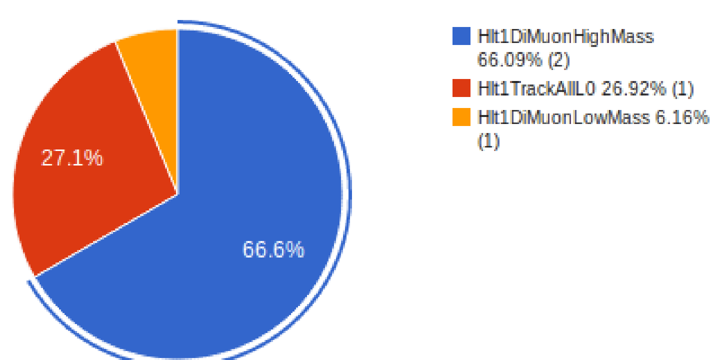

Figure 12. CPU Time percentage of top-level algorithms in the Gaudi sequence tree.

The report on Figure 12 was produced by a user application that took an exported CSV data and compiled it to javascript code that can be inserted to any web page [6].

5. Conclusions

In this paper we presented the Gaudi Intel Performance Auditor — a CPU profiling tool that is used in LHCb experiment at CERN. This tool integrates the functionality of Intel® VTune™Amplifier XE performance profiler to the LHCb core framework Gaudi. The key advantage of the auditor is an ability to produce reports that use the framework's modules to present performance analysis results. Those reports help developers to identify hotspots in the code and improve the application performance. Besides the reports, the Gaudi Intel Performance Auditor provides the options that allow to control the Intel® VTune™Amplifier XE supervisors process from the Gaudi applications.

The results have further strengthened our confidence in the profiling sampling technique. This technique gives us a reasonable overhead of total profiling time (5% at Intel® VTune™Amplifier XE) in comparison to the tools that count the functions calls. For example, the popular profiling tool Valgrind [7] counts every code instruction and programs running under this tool usually run from five to twenty times slower than running outside Valgrind. Though Valgrind provides precise measurements, using the sampling technique we can get accurate results by tuning the sampling interval or increasing the number of processing events.

We hope that this example of using Intel® VTune™Amplifier XE in Gaudi framework would help to implement other advanced profiling tools.

References

[1] "Modular Software Performance Monitoring", Daniele Francesco Kruse and Karol Kruzelecki 2011 J. Phys.: Conf. Ser. 331 042014.
[2] Barrand G et al. 2001 Comput. Phys. Commun. 140 45-55
[3] "Perfmon2: a standard performance monitoring interface for Linux", Stephane Eranian http://perfmon2.sourceforge.net/perfmon2-20080124.pdf
[4] Intel® VTune™Amplifier XE http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/
[5] Google® Performance Tools http://code.google.com/p/gperftools/
[6] Custom pie chart reports http://amazurov.ru/cern/hltprofilingresults/
[7] Valgrind http://valgrind.org/