Modular Software Performance Monitoring

Daniele Francesco Kruse[1] and Karol Kruzelecki[2]
[1] PH/SFT group, CERN, Geneva, CH
[2] PH/LBC group, CERN, Geneva, CH, on behalf of the LHCb collaboration

E-mail:
[1] daniele.francesco.kruse@cern.ch
[2] karol.kruzelecki@cern.ch

Abstract. CPU clock frequency is not likely to be increased significantly in the coming years, and data analysis speed can be improved by using more processors or buying new machines, only if one is willing to change the programming paradigm to a parallel one. Therefore, performance monitoring procedures and tools are needed to help programmers to optimize existing software running on current and future hardware. Low level information from hardware performance counters is vital to spot specific performance problems slowing program execution. HEP software is often huge and complex, and existing tools are unable to give results with the required granularity. We will report on the approach we have chosen to solve this problem that involves decomposing the application into parts and monitoring each one of them separately. Both counting and sampling methods are used to allow an analysis with the required custom granularity: from global level, up to the function level. A set of tools (based on perfmon2 - a software interface to hardware counters) for CMSSW, Gaudi and Geant4 has been developed and deployed. We will show how this type of analysis has been proven useful in spotting specific performance problems and effective in helping with code optimization.

1. Introduction

The days when Moore's Law guaranteed a stable and transparent computing performance gain each and every year, are over. Processor clock speed cannot be increased anymore and, even if it could, it would not help since memory is still far behind in terms of speed. Memory will not catch up with the processor in the near future, and it will continue to be a bottleneck. Programs are performance-greedy, and as they get larger and more complex, they require improved and faster hardware to run properly. The hardware improvements that are available today include: multiple processors, multiple cores and NUMA architectures. Although all these are very promising, they are definitely non-transparent for programmers, for at least two reasons: firstly programmers need to write multi-threaded code, secondly, as the limited hardware resources (caches, bus, main memory) are shared among cores and processors, programmers need to constantly monitor how their programs use these resources in order to avoid bottlenecks and to speed up performance. Writing efficient and correct multi-threaded code is non-trivial and will not be covered in this article. Instead we will focus on monitoring the performance of single-threaded programs, to find problems and inefficiencies in the code, to optimize it and to get the most out of today's hardware. This article and all the research done, concerns only the most recent Intel processor families: *Core*, *Nehalem* and *Westmere* [1, 2].

The next section will explain the motivations and objectives of our project. In sections 3 and 4 we illustrate the benefits of modularity in performance monitoring and how we applied it to physics software frameworks. In sections 5 and 6 we explain the method and the tools used to monitor an application, and how to get relevant and useful information from raw counter data. Finally we will show how we deployed the monitoring tool and a simple successful usage example.

2. Motivation and Goal

High energy physics (HEP) software is huge and complex and it is developed by hundreds of physicists and programmers often unaware of performance issues. As pointed out in the introduction, the reason comes from the fact that, for many years, expertise in this field was not required, because the ever faster hardware would compensate for it. The software produced is therefore suboptimal in terms of efficiency and speed. Moreover, given the heterogeneous group of developers of HEP software and its large and complex structure (hundreds of libraries and thousands of classes), the job of the performance optimization teams is more difficult than ever. Our goal was then to develop a tool, targeted at HEP software, to help spot the problems and find the parts of code responsible for them, so that they could be solved eventually.

3. *Monolithic* vs. *Modular* Monitoring

Generic performance analysis tools currently available (both open-source and commercial) are countless, and some have proven useful in some occasions. Other tools have been developed ad-hoc for specific software frameworks. As CPU vendors became more aware of the need for application performance analysis, they started to provide hardware-based performance counters able to monitor different aspects of how applications run on processors. Few tools on the market exploit these powerful hardware counters, and, even the ones that do exploit them, provide little insight as to which part of the monitored software is causing problems. Being generic tools, they obviously treat the application as a black box, thus providing what we call a *monolithic* analysis. While for very small programs and kernels, monolithic analysis is often sufficient, when we face large and complex software it becomes less useful. Examples of "traditional" monolithic monitoring tools using performance counters are *PTU* [8] and *pfmon* [6]. Both are able to count and sample CPU events of a running application. Their sampling capabilities allow one to see in which functions those events occur. But they do not tell much more: it is difficult to understand if a function with a high count is causing the problem or if it is the application that is calling that function too often, and when this is the case, it is hard to decide if it is a particular part of the code that is responsible for the calls or if the calls are equally spread all over it. This issue is not so evident in small and single-developer code, but becomes quickly unwieldy when software grows both in size and in number of developers (like in major HEP frameworks). The solution we chose was of the type *divide et impera*: instead of monitoring the application as a whole, we divide it into what we call *modules*, and then monitor each module separately. In other words, modules are parts that the application source code can be divided into. The choice of this division depends mainly on the application structure, and it is made so that the code instrumentation is minimal, if any. Modular monitoring allows a better insight into where the problem is coming from, narrowing down the set of places where we need to look for causes of the inefficient behaviour of the monitored application. We still need to provide both counting and sampling results. Counting results allow us to identify "troubled" modules, the first step in the optimization procedure. Once the module we would like to work on and improve is identified, sampling results tell us in which functions, called within that module, the problems occur. Using modularity we therefore remove the complexity and the multi-developer related problems, dividing the whole application into smaller bits, where each bit has an affordable complexity and often a single developer. This is especially important during the optimization step, when the author of the code can give invaluable advice to the optimization team about his code and how to change it or improve it.

4. Modularity in CMSSW, Gaudi and Geant4

As previously said, the modularity of the analysis varies depending on the structure of the application. The first application to which the tool has been applied is *CMSSW*, i.e. the current official CMS (the Compact Muon Solenoid experiment at CERN) simulation and reconstruction software framework. CMSSW code is organized into *modules* that are sequencially executed during the processing of each event. The CMSSW framework provides hooks to execute user defined actions at the beginning and at the end of each module execution. We exploit these hooks to avoid code instrumentation and provide the monitoring tool as a *service* of the framework. Each physics event goes through a number of software modules depending on its type. We use the hook provided at the beginning of each module to start counting (or sampling) and the one at the end to stop the monitoring process. Gaudi, the second application which the tool has been applied to, is a software architecture and framework for building LHCb (the Large Hadron Collider beauty experiment) and ATLAS (A Toroidal LHC ApparatuS) data processing applications. Gaudi provides a hooking mechanism similar to CMSSW, but in this case modules are called *algorithms*, and unlike CMSSW modules, algorithms are not executed necessarily sequencially, as they are often nested within each other.

Geant4 is the third and last software framework that the monitoring tool has been applied to. Geant4 is a toolkit for the simulation of the passage of particles through matter. For Gaudi and CMSSW we used "code division" modularity, but Geant4 doesn't provide hooks for different parts of its code, so we chose a different approach. Instead of grouping results by code region, we group them by physics conditions during runs. In each Geant4 *run* there are a series of *events*, each event has many *tracks* (one for every particle produced), each track is decomposed into the smaller units called *steps*. Each step is characterised by many variables, but we considered the three most significant in our analysis: the particle type, the energy range that the particle has, and the physical volume it is passing through. Binning the monitoring results into particle/energy/volume combinations provides an interesting insight about how different simulated physical situations are handled by Geant4. For example it allows one to discover if a particular particle type is handled inefficiently, or if particular volume (at any level in the geometry tree) or material is causing performance problems, or if it is a combination of the two that results in slow program execution. While this does not directly point to a particular part of code to optimize, it is very useful to test the performance of Geant4 while handling different physics conditions.

5. The PMU and Perfmon

In what follows, we will understand what hardware-based performance counters are in detail, and we will show how we use them to extract important information on how the monitored application performs.

Performance monitoring can be defined as *the action of collecting information related to how an application or system performs* [7]. The aim of performance monitoring is to identify bottlenecks and remove them to improve software performance.

Hardware performance counters are the resource we use for application performance monitoring. But what are performance counters? All new micro-architectures include a special hardware unit called PMU or Performance Monitoring Unit that contains a set of registers called performance counters. Access to the PMU requires kernel support to read and write its privileged registers. These counters are able to detect and count certain micro-architectural events from several hardware sources, like the pipeline, system bus or cpu caches. These include events such as CPU cycles, cache references, misses, instruction TLB (Translation Lookaside Buffer) misses, system bus utilization, and other kinds of possible events and states of the processor. Such events provide facilities to characterize the interaction between programmed sequences of instructions and microarchitectural subsystems. In contrast to machine simulations, they are available on most of today's hardware and they do not require software to be modified or recompiled. When properly used, they incur in a very low overhead. There is an enormous amount of event types that can be counted, so a big effort has to be put on deciding which events we should count and how we should use them to understand the performance of an application.

In order to tell the PMU which events we want to monitor and when we want to start or stop monitoring, we use an API called *Perfmon2* [6]. This interface was developed and it is currently maintained by Stephane Eranian of Google Corporation (though he did most of *Perfmon2* related work when he was working at HP). It is portable across many recent micro-architectures, it supports system-wide and per-thread monitoring and, besides counting events, it also supports sampling.

Figure 1 shows the layering of perfmon components. At the bottom we see the CPU Hardware that contains the PMU. *Perfmon2* interacts with the PMU using a patched Linux kernel. In fact the vanilla kernel does not include support for perfmon. The *Perfmon2* library (libpfm) is divided in two parts: architectural and generic. The architectural part is specific for the microarchitecture used in the machine (in our case Intel Core Microarchitecture), while the generic part provides a common interface to the user of the library. On top of *Perfmon2* library there are the user space applications that make use of the library. As said before, `pfmon` is such an application, but also the performance analysis tool that we developed for CMSSW is an other example.

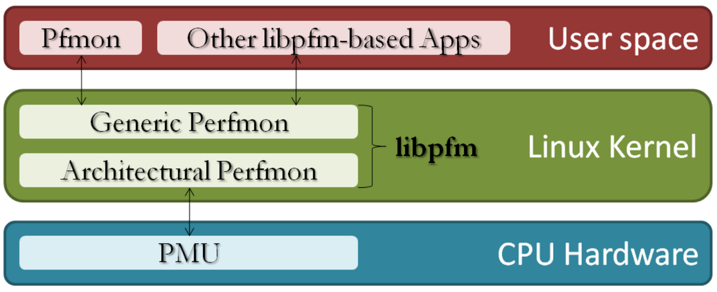


Figure 1. Layering of perfmon components.

6. Cycle Accounting Analysis

Now that we have described the counters and the API needed to access them, we will explain what kind of relevant information we are looking for and which events we need to monitor to get it.

The Cycle Accounting Analysis is a methodology to analyse the performance of an application and to find its weak points. It is specific for the Intel Core Microarchitecture and was developed by David Levinthal of Intel Corporation [3]. It is also the recommended methodology of the Intel Architectures Optimization Reference Manual. Figure 2 illustrates the cycle decomposition schema of an application execution. According to it, the total execution time, i.e. the total execution cycles of an application, can be divided into cycles in which the front-end is issuing μops and cycles in which it is not. The cycles in which the front-end is issuing μops can be further divided into cycles which are retiring μops (i.e. when the processor completes the execution of instructions) and cycles which are not. One example of μops issued but not retired is when branch misprediction occurs, μops which were issued and executed do not get retired since they belong to a speculative execution which did not prove correct eventually. So basically we have three possible types of cycles: cycles retiring μops, i.e. doing useful work, cycles issuing μops but not retiring them, i.e. doing useless work, and finally cycles which are not issuing μops at all, i.e. stalled cycles doing no work. As you can see in figure 2, the stalled cycles can be further decomposed into 5 major components to better understand who is responsible for the lost cycles: data load related stalls, floating point exceptions, cycles stalled due to long-latency divisions and/or square root operations executing, instruction fetching related stalls, stalls due to jumps and branches [4]. The aim of software optimization is therefore:

(i) To bring the stalled cycles close to 0% by improving code and data locality for example.

(ii) To do the same for cycles that are not retiring μops by making the existing branching more predictable.

(iii) To reduce the number of cycles which are retiring μops by using vector instructions where possible, and using faster and more efficient algorithms of course.

Doing so will result in fewer total cycles and therefore a faster application.

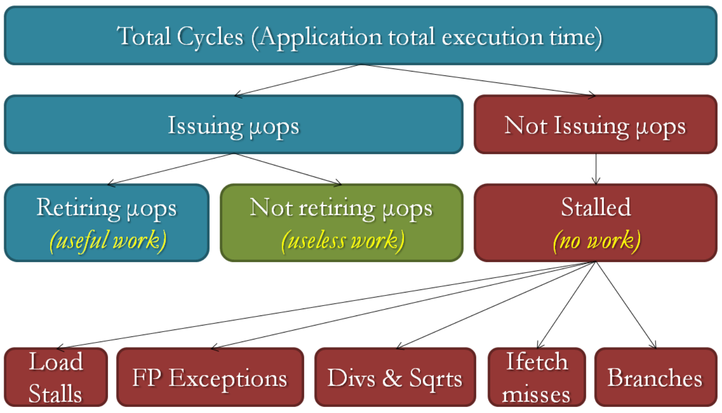


Figure 2. Cycle Accounting Analysis - Cycle Decomposition.

While giving detailed reports of all kinds of statistics needed for cycle accounting, our modular analysis focuses on the five different components of the stall cycles. That is the part of the execution cycles that we can (and should) improve more, shrinking it as much as possible. For each component, we monitor one or more CPU event to quantify how much each one of them contributes to the total number of stalls. In Table 1 we show which events are monitored in each category.

This information will be fundamental when optimizing the code, because it will tell us what kind of change we need to apply to the code to make it more efficient. In summary, modularity tells us the part of code to look at (and thus probably also the developer to contact), and the result of the stall analysis tells us what kind of problem needs to be fixed in that code.

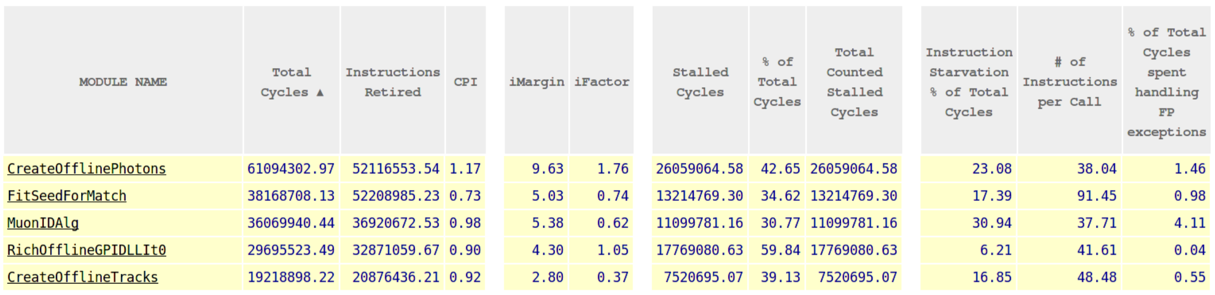
7. Deployment, Usage and Results within the Gaudi Framework

Our goal was to make the tool as easy to use as possible, and therefore it was necessary to make all the steps of software profiling and postprocessing automatic. The Gaudi framework, the base of the LHCb experiment software, provides an easy way to configure all the parameters of a job through Python scripts. We created a wrapper called `GaudiProfiler` that can be used instead of the default executable of Gaudi, `gaudirun.py`. The wrapper takes care of the on-the-fly generation of specific options for the profiler, the definition of the groups of events to monitor, and finally, after collecting all the necessary data, the postprocessing which results in browsable HTML result tables.

BASIC STATS	Total Cycles Instructions Retired CPI
IMPROVEMENT OPPORTUNITY	iMargin iFactor
BASIC STALL STATS	Stalled Cycles % of Total Cycles Total Counted Stalled Cycles
INSTRUCTION USEFUL INFO	Instruction Starvation # of Instructions per Call
FLOATING POINT EXCEPTIONS	% of cycles handling FP exceptions
LOAD OPS STALLS	L2 Hit L3 Unshared Hit L2 Other Core Hit L2 Other Core Hit Modified L3 Miss -> Local DRAM Hit L3 Miss -> Remote DRAM Hit L3 Miss -> Remote Cache Hit
ITLB MISSES	L1 ITLB Miss Impact ITLB Miss Rate
DTLB MISSES	L1 DTLB Miss Impact L1 DTLB Miss % of Load Stalls
DIVISION & SQUAREROOT STALLS	Cycles spent during DIV & SQRT Ops
L2 IFETCH MISSES	Total L2 IFETCH misses IFETCHes served by Local DRAM IFETCHes served by L3 (Modified) IFETCHes served by L3 (Clean Snoop) IFETCHes served by Remote L2 IFETCHes served by Remote DRAM IFETCHes served by L3 (No Snoop)
BRANCHES, CALLS & RETS	Total Branch Instructions Executed % of Mispredicted Branches Direct Near Calls Indirect Near Calls Indirect Near Non-Calls All Near Calls All Non Calls All Returns Conditionals
INSTRUCTION STATS	Branches, Loads, Stores, Packed

Table 1. Types of micro-architectural events monitored

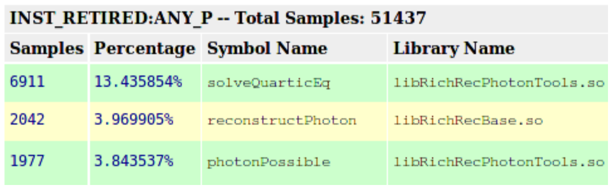
During the testing phase of the tool, we have chosen the LHCb event reconstruction application *Brunel* as an example piece of software to monitor and improve. The first step was to run the overall analysis and collect data for all software modules, *algorithms* in Gaudi's vocabulary. After having considered only the algorithms with high *improvement margin* and high *improvement factor* (`iMargin` and `iFactor` in Table 1), we decided to work on the `CreateOfflinePhotons` algorithm (see Figure 3). The `iMargin` is a metric that quantifies how much an improvement to that particular algorithm would benefit the entire application if the number of clock cycles per instruction ratio were to reach the theoretical minimum (0.25 for Core and Nehalem processors). The `iFactor` is an index from 0 to 3 that roughly indicates how much the single algorithm can be improved, regardless of the effect on the entire application; this metric takes into consideration the number of stalled cycles, the level of branch misprediction, and the fraction of packed instructions over all intructions.



MODULE NAME	Total Cycles ▲	Instructions Retired	CPI	iMargin	iFactor	Stalled Cycles	% of Total Cycles	Total Counted Stalled Cycles	Instruction Starvation % of Total Cycles	# of Instructions per Call	% of Total Cycles spent handling FP exceptions
CreateOfflinePhotons	61094302.97	52116553.54	1.17	9.63	1.76	26059064.58	42.65	26059064.58	23.08	38.04	1.46
FitSeedForMatch	38168708.13	52208985.23	0.73	5.03	0.74	13214769.30	34.62	13214769.30	17.39	91.45	0.98
MuonIDAlg	36069940.44	36920672.53	0.98	5.38	0.62	11099781.16	30.77	11099781.16	30.94	37.71	4.11
RichOfflineGPIDLLIt0	29695523.49	32871059.67	0.90	4.30	1.05	17769080.63	59.84	17769080.63	6.21	41.61	0.04
CreateOfflineTracks	19218898.22	20876436.21	0.92	2.80	0.37	7520695.07	39.13	7520695.07	16.85	48.48	0.55

Figure 3. Partial view of LHCb modules list with data collected by the profiler.

The following step was to find a hotspot function using the detailed symbol view of `CreateOfflinePhotons` (see Figure 4). The sampling result tables are generated for each module of the framework. For each monitored event, they provide the number of samples collected for each symbol (function or method) in the module, their percentage of the total number of samples for that module, the symbol name, and the library where the symbol belongs. After a quick investigation on which of the methods, among those taking a considerable time of execution time, would be easier to improve, we made some small changes in the body of `solveQuarticEq`, which resulted in an improvement of 2% of the overall algorithm speed. Given that the algorithm considered was one of the main ones (high *improvement margin*), the improvement was also measureable in the overall application execution.



INST_RETIRED:ANY_P -- Total Samples: 51437			
Samples	Percentage	Symbol Name	Library Name
6911	13.435854%	solveQuarticEq	libRichRecPhotonTools.so
2042	3.969905%	reconstructPhoton	libRichRecBase.so
1977	3.843537%	photonPossible	libRichRecPhotonTools.so

Figure 4. Functions contributing to the instruction count of the `CreateOfflinePhotons` algorithm.

8. Conclusions

In this paper we discussed the usefulness of modular software monitoring for high energy physics software. We showed how it helps answering the two most important questions for developers and performance optimizers of huge and heterogenously developed software frameworks: where is the bottleneck? what kind of problem is it? By doing so, we pointed out that modularity also helps narrowing down the number of developers to contact as responsible for the part of source code that is slowing the execution.

We developed a set of tools based on *Perfmon2*, which exploit the hardware performance counters, to allow the developer (or the end user) to modularly monitor the performance of his application within three frameworks: CMSSW, Gaudi and Geant4. Besides modularity, as an additional advantage to existing free and commercial tools, the results are given in the form of easy-to-understand derived statistics, instead of raw counter data, to allow a more effective understanding of the ongoing processes and bottlenecks inside the CPU during program execution.

We showed how in the Gaudi framework, the improvements made to a single function selected using the tool we developed, reduced noticeably the clock cycle count and therefore also the total program execution time. We believe that systematic usage of the tool, and consequent code improvement, will result in significant CPU time savings, making HEP production software more efficient and lowering the cost of physics data processing.

9. Bibliography

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*
`http://www.intel.com/Assets/PDF/manual/253665.pdf`

[2] *Intel 64 and IA-32 Architectures Optimization Reference Manual*
`http://www.intel.com/Assets/PDF/manual/248966.pdf`

[3] David Levinthal, *Cycle Accounting Analysis on Intel Core 2 Processors*
`http://assets.devx.com/goparallel/18027.pdf`

[4] David Levinthal, *Introduction to Performance Analysis on Intel Core 2 Duo Processors*
`http://assets.devx.com/goparallel/17775.pdf`

[5] David Levinthal, *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors*
`http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`

[6] Stephane Eranian, *Perfmon2: a standard performance monitoring interface for Linux*
`http://perfmon2.sourceforge.net/perfmon2-20080124.pdf`

[7] Stephane Eranian, *Perfmon2: a standard performance monitoring interface for Linux*
`http://cscads.rice.edu/workshops/july2007/perf-slides-07/Eranian-Perfmon.pdf`

[8] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, D. Ryabtsev, *Parallelization Made Easier with Intel Performance-Tuning Utility*
`ftp://download.intel.com/technology/itj/2007/v11i4/2-parallelization/2-Parallelization_Made_Easier.pdf`