Proceedings of the First EELA-2 Conference
R. mayo et al. (Eds.)
CIEMAT 2009
© 2009 The authors. All rights reserved

Performance Monitoring of the Software Frameworks for LHC Experiments

William A. Romero R.[1], J.M. Dana[2]

[1]*Systems and Computing Department. Universidad de los Andes*
wil-rome@uniandes.edu.co
[2]CERN
Jose.Dana@cern.ch

Abstract

This paper presents the taken methodology and experienced results of the performance monitoring of the software framework for LHC experiments: LHCb, CMS, and ALICE. As monitoring tool, pfmon was used. The performance monitoring and tuning tasks are composed by the following steps: pfmon deluxe analysis, pfmon profiling and application improvement. The objective is to improve the identified weakness in order to enhance the application performance. A new functionality has been added to pfmon in order to resolve the symbols generated in the profiling for the 32-bit version of the software frameworks.

1. Introduction

Performance monitoring is a necessary practice in High Performance Computing. An appropriate monitoring allows to identify well-known signs about how the application is being executed and key processes in that execution. In this way, it is possible to find the functions, methods (in terms of the Object-Oriented programming) or procedures that should be modified in order to improve the application performance according to the technology used.

On the other hand, as it is necessary to save on resources, it is important for the software designer/programmer to check on the hardware results, finding out weaknesses and having in mind the comparison between manpower and hardware capacity in order to empower the hardware results through a lower price [8].

For the Large Hadron Collider (LHC), the High Energy Physics (HEP) community has developed huge C++ software frameworks for event generation, detector simulation, and data analysis. The scope of this work is to study the performance given by the software frameworks, analyze their bottlenecks and isolate the more important parts in order to analyze them independently, and improve the application execution.

This work presents the experience at monitoring the software frameworks for LHCb [9], CMS [12] and ALICE [1] experiments. As monitoring tool *pfmon* was used, as CERN openlab [3] previous experience had supported the efficiency of the application.

This paper is organized as follows: Section 2 introduces the monitoring tool used. In the Section 3, the methodology developed is described. Section 4 briefly summarizes the execution stages in the analysis frameworks for LHC experiments. Section 5 shows the obtained results from the monitoring of the software frameworks, followed by the main related issues and conclusions.

2. Monitoring Tool: *pfmon*

In order to get information about how the application is being executed by the processor and to understand how the application performs, a monitoring tool is necessary. In this work *pfmon* [14] was used; a command-line program that, through *perform2* [13] and *libpfm*, allows access to the Performance Monitoring Unit (PMU) of the processor and its performance counters.

Perform2 is a Linux kernel interface that provides a uniform abstract model to access PMU counters for most modern processors such as Intel Itanium, Intel Xeon and AMD Opteron, among others. In this way, *pfmon* with *libpfm* library, access to the interface with the purpose of collecting simple counts and profiles by sampling PMU registers. It also provides support for per-thread and system-wide measurements [4][5].

One of the advantages of *pfmon* is the non-intrusive method for profiling. It does not require labels into the program code or special compilation modes for the program. With this tool it is possible to get the names of processes executed by the processor. It is feasible to set a sampling period (sampling mode) in order to check the function calls and the percentage of utilization in the application execution.

3. Performance Monitoring and Tuning Task

The performance monitoring and tuning tasks consist of three basic steps: *pfmon* deluxe Analysis, *pfmon* profiling and application improvement. At the end, a tightly verification is made (though the first and second step) to check the new application performance.

3.1. *pfmon deluxe* Analysis

As it was mentioned above, *pfmon* can do some measurements on the PMU but, in order to understand the performance behaviour of the application execution, some additional calculations are necessary. At CERN openlab a python script was implemented to provide a specific mode of analysis according to a certain group of events, this script is called *pfmon deluxe*. The analysis modes are: *standard*, *simd1*, *simd2*, *simd_uop* and *stalls*. For the purposes of this work, the *standard* and *simd1* modes were used. The table 1 shows some of the information calculated by the *standard* analysis of *pfmon deluxe*.

Table 1. Some of the *pfmon deluxe* standard information.

Percentage of	Formula
Load and store instructions	`((INST_RETIRED:STORES + INST_RETIRED:LOADS) / INSTRUCTIONS_RETIRED ) * 100`
branch instruction mispredicted	`(MISPREDICTED_BRANCH_RETIRED/BRANCH_INSTRUCTIONS_RETIRED)*100`
L2 loads missed	`(LAST_LEVEL_CACHE_MISSES/LAST_LEVEL_CACHE_REFERENCES)*100`
Bus utilization	`((BUS_TRANS_ANY:ALL_AGENTS)* 2/CPU_CLK_UNHALTED:BUS)*100`
Comp. SIMD instructions (new FP)	`(SIMD_COMP_INST_RETIRED:PACKED_SINGLE: SCALAR_SINGLE:PACKED_DOUBLE:SCALAR_DOUBLE/ INSTRUCTIONS_RETIRED)*100`
Comp. x87 Instructions (old FP)	`(X87_OPS_RETIRED:ANY/INSTRUCTIONS_RETIRED)*100`

As an example, the *pfmon* outcome shown below (figure 1) reflects that the program execution has 94.554% of L2 cache misses and, therefore, there is an issue related with memory management. In order to identify where the problem is, it is necessary to take the next step: *pfmon* profiling.

```
Ratios:
                                          CPI:  2.0529
                          load instructions %:  24.888%
                         store instructions %:  14.751%
                 load and store instructions %:  39.639%
                resource stalls % (of cycles):  53.562%
                        branch instructions %:  18.223%
              % of branch instr. mispredicted:  0.714%
                         % of l2 loads missed:  94.554%
                            bus utilization %:  8.158%
                       data bus utilization %:  4.631%
                              bus not ready %:  0.000%
               comp. SIMD instr. ('new FP') %:  1.585%
                comp. x87 instr. ('old FP') %:  0.000%
```

Figure 1. ***pfmon deluxe*** **results.**

3.2. *pfmon* Profiling

This step allows determining the percentage of the total time spent in a function. The objective is to identify the specific function that could be improved in order to optimize the whole program. Figure 2 shows the standard output of *pfmon* profiling.

When *pfmon* is used in profile mode, every *n*-quantity of occurrences of an event within the CPU (clock cycles), the PMU would dump the address of the instruction pointer (IP). Therefore, it is possible to get the set of addresses that are visited frequently by the program and identify which code is being used.

```
# results for [27703<-[27641] tid: 27703]
(/gauss/soft/lhcb/GAUSS/GAUSS_v30r5/Sim/Gauss/v30r5/slc4_amd64_gcc34/Gauss.exe)
# total samples          : 64913963
# total buffer overflows : 31696
#

 counts   %self   %cum                              code addr symbol
 2776941  4.28%   4.28%  0x00002b5c990926c0   CLHEP::RanluxEngine::flat()</data4/wilrome/ga
 2365853  3.64%   7.92%  0x00002b5ca2dcb2e0   G4ElasticHadrNucleusHE::GetLightFq2(int,
 2066022  3.18%   11.11% 0x000000306150e370   __ieee754_exp</lib64/tls/libm-2.3.4.so>
 1964096  3.03%   14.13% 0x0000003061511930   __ieee754_log</lib64/tls/libm-2.3.4.so>
 1622689  2.50%   16.63% 0x000000306126b5f0   __GI___libc_malloc</lib64/tls/libc-2.3.4.so>
 1508825  2.32%   18.95% 0x00002b5c9d34e5e0   MagneticFieldSvc::fieldVector(ROOT::Math::Pos
 1401687  2.16%   21.11% 0x0000003061269510   __cfree</lib64/tls/libc-2.3.4.so>
 1345044  2.07%   23.19% 0x00002b5c9ca8cae0   G4Navigator::LocateGlobalPointAndSetup(CLHEP:
```

Figure 2. Results generated by *pfmon* profiling.

The addresses themselves are meaningless to the average user, but they are translated into program symbols, which map onto function and/or data names (labels within the code). Sometimes the monitored programs open shared libraries using *dlopen*, and in this case *perform2* has to intercept the moment of the opening in order to know which library was loaded and where it was placed in memory.

```
# results for [27703<-[27641] tid: 27703]
(/gauss/soft/lhcb/GAUSS/GAUSS_v30r5/Sim/Gauss/v30r5/slc4_amd64_gcc34/Gauss.exe)
# total samples          : 64913963
# total buffer overflows : 31696
#

 counts   %self   %cum                              code addr symbol
   145173  0.22%  75.01%  0x000000306152b090   __GI___isnan</lib64/tls/libm-2.3.4.so>
  1622689  2.50%  16.63%  0x000000306126b5f0   __GI___libc_malloc</lib64/tls/libc-2.3.4.so>
   344666  0.53%  54.06%  0x00000030612723a0   __GI_memcpy</lib64/tls/libc-2.3.4.so>
   177884  0.27%  70.50%  0x0000003061270a00   __GI_strlen</lib64/tls/libc-2.3.4.so>
   243524  0.38%  61.50%  0x0000003063da9a80   __gnu_cxx::__exchange_and_add(int
   199310  0.31%  68.52%  0x00000030615095b0   __ieee754_atan2</lib64/tls/libm-2.3.4.so>
  2066022  3.18%  11.11%  0x000000306150e370   __ieee754_exp</lib64/tls/libm-2.3.4.so>
  1964096  3.03%  14.13%  0x0000003061511930   __ieee754_log</lib64/tls/libm-2.3.4.so>
   317859  0.49%  57.59%  0x00000030615135a0   __ieee754_pow</lib64/tls/libm-2.3.4.so>
   181292  0.28%  69.95%  0x0000003061527760   __log</lib64/tls/libm-2.3.4.so>
   300545  0.46%  59.01%  0x000000306151c2e0   __sin</lib64/tls/libm-2.3.4.so>
   333070  0.51%  55.11%  0x00002b5c9c9f1918   _init</data4/wilrome/gauss/soft/lhcb/GEANT4/G
   190218  0.29%  69.10%  0x00002b5c9c4551a0   _init</data4/wilrome/gauss/soft/lhcb/GEANT4/G
   140059  0.22%  75.44%  0x00002b5c9ccccc58   _init</data4/wilrome/gauss/soft/lhcb/GEANT4/G
   133222  0.21%  76.07%  0x00002b5ca2cae188   _init</data4/wilrome/gauss/soft/lhcb/GEANT4/G
   582164  0.90%  41.58%  0x0000003061268c50   _int_free</lib64/tls/libc-2.3.4.so>
  1120478  1.73%  24.91%  0x00000030612695d0   _int_malloc</lib64/tls/libc-2.3.4.so>
   199403  0.31%  68.21%  0x00002b5c9cfb6c70   CLHEP::Hep3Vector::operator()(int)
   161171  0.25%  73.60%  0x00002b5c9cfb6490   CLHEP::Hep3Vector::rotateUz(CLHEP::Hep3Vector
   170173  0.26%  72.08%  0x00002b5c99087d80   CLHEP::HepRandom::getTheEngine()</data4/wilro
   636781  0.98%  36.95%  0x00002b5c9cfa2030   CLHEP::HepRotation::rotateAxes(CLHEP::Hep3Vec
  2776941  4.28%  4.28%   0x00002b5c990926c0   CLHEP::RanluxEngine::flat()</data4/wilrome/ga
   322374  0.50%  56.61%  0x00002b5c9c9ff970   G4Box::DistanceToIn(CLHEP::Hep3Vector
   343197  0.53%  54.59%  0x00002b5c9c9ffe10   G4Box::DistanceToOut(CLHEP::Hep3Vector
```

Figure 3. *pfmon* profiling results ordered by code addr symbol column.

It is feasible to organize the results in order to identify important execution elements of the application such as classes and packages, among others. For example, in figure 3 it is possible to see a certain group of calls to both the IEEE Standard library for Binary Floating-Point Arithmetic and CLHEP (a Class Library for High Energy Physics).

In object-oriented programs it is possible to identify, through a sorted profiling, the percentage of the total time spent in an object instance, as well as the invocations between methods of the same class. A method could be at the top of the profiling results, but the real bottleneck may be in one of the used classes.

3.3. Application Improvement

As already described, the HEP software is written using several existing frameworks such as ROOT and Geant4, among others. Which means, several programming languages are used to develop the required applications; for the LHC software frameworks the main one is C++. The C++ standard compiler tool at CERN is GCC [6]. The GCC versions have optimization options (O1, O2, O3, Os, etc.) in order to enhance the application performance, but sometimes the compiler does not identify possible code pieces that could be improved. The default optimization level for LHC software frameworks is O2.

From profiling results the objective is to identify the code pieces that are difficult to improve for the compiler.

Two examples are shown in tables 2 and 3. In the first one, the developer is allocating all the memory and, after that, operating over the matrixes. In order to improve the memory management and reduce the number of cache misses, it would be better to allocate the memory just for the first matrix, operate and, after that, allocate the second matrix.

Table 2. Code improvement example 1.

Original code

```
for(i=0;i<N;i++) {
  imageA[i]=loadimg(fileA[i]);
  imageB[i]=loadimg(fileB[i]);
  a1[i]=funcA(imageA[i]);
  a2[i]=funcB(imageA[i]);
  b1[i]=funcA(imageB[i]);
  b2[i]=funcB(imageB[i]);
}
```

Improved version

```
for(i=0;i<N;i++) {
  imageA[i]=loadimg(fileA[i]);
  a1[i]=funcA(imageA[i]);
  a2[i]=funcB(imageA[i]);
  imageB[i]=loadimg(fileB[i]);
  b1[i]=funcA(imageB[i]);
  b2[i]=funcB(imageB[i]);
}
```

In the second example, it is shown the usual memory allocation for a matrix. However, we can never be sure about the alignment of the data in memory. The compiler, usually, will try to allocate the memory as aligned as possible, in order to reduce the number of memory accesses. However, it won't be able to do that in any possible situation. In this case, it is better to allocate the memory manually and be sure that, every time we go through all the elements of the matrix, the number of memory accesses will be minimum.

Both of them are typical situations where the compiler cannot improve the code as much as we would like to.

Table 3. Code improvement example 2.

Original code

```
matrix=(unsigned char**)malloc(height*sizeof(unsigned char*));

for(i=0;i<height;i++)
  matrix[i]=(unsigned char*)malloc(width*sizeof(unsigned char));
```

Improved version

```
matrix=(unsigned char**)malloc(height*sizeof(unsigned char*));
matrix[0]=(unsigned char*)malloc(height*width*sizeof(unsigned char));

for(i=1;i<height;i++)
  matrix[i]=matrix[i-1]+width;
```

4. Overview of the Execution Stages in the LHC Analysis Frameworks

In general, software frameworks for LHC experiments are a chain of specialized processes. These processes correspond to how an experiment is executed: 1) events are produced by a collision, 2) the particles cross through the detector, 3) a data acquisition system (DAQ) collect the produced signals and 4) the signals are transformed in information according to the physics theory. The software frameworks are the result of modelling the process described above; the objective is to validate methods for the experiment calibration and tuning (detectors, DAQ system, etc.).

According to this model, the software framework is composed by execution stages; each one depending on the outputs generated by the previous stage. These execution stages are shown below (figure 4):

- Generation: Event generation (e.g. using a Monte Carlo method). The software is based on PYTHIA [10], Alpgen [2], etc.
- Simulation: Particles through detector; the signals produced by the detectors and electronics devices are stored as RAW data. The software in this case is based, mainly, on Geant4 [7].

- Digitization: In this stage, the RAW data is transformed to information.
- Reconstruction: To process the information to get new one according to the physics theory. The framework used in the last two steps is called ROOT [11].

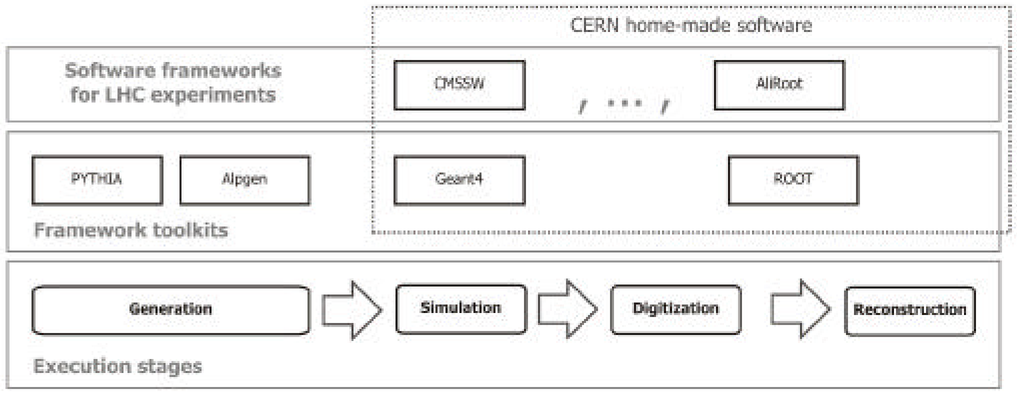

Figure 4. Execution stages and related software.

There are cases in which two stages are implemented in one. For example, the ALICE software framework has only two stages: Simulation and Reconstruction. But not all frameworks are designed in the same way. Their specific design is a consequence of their purposes and the development team decisions.

5. Monitoring Process Conclusions

Generally speaking, these frameworks are used to be launched and simulate processes of the order of thousands. Though, for the purposes of the monitoring process 5, 50 and 150 order events are used. Some extracts are presented as a brief recapitulation of the processed monitoring. The performance monitoring tasks were developed on the stages: Generation (CMS), Simulation (LHCb, CMS, ALICE), Digitization (CMS) and Reconstruction (CMS, ALICE). The monitoring was performed on a machine with Intel Xeon architecture, two processors Dual-Core 2.66 GHz 64 bits, 4 MB L2 cache, 8 GB of RAM and the Scientific Linux CERN 4.7 operating system.

5.1. LHCb

There are 2 functional versions for simulation stage: 32-bit and 64-bit. The execution can be made using a number of threads defined by the user. In general, the behaviour is similar between both versions: cycles per instruction, load and store instructions, etc. An important difference is at the use of SIMD instructions; it is a consequence of the compilation mode process. For the 32-bit version, the compiler does not know the specific processor architecture and implements the x87 instruction subset. On the other hand, for the 64-bit version, the compiler knows that the processor architecture supports SIMD instructions. Finally, as a specific sign, the percentage of bus utilization increases with the number of threads.

From the profiling results the method `ScanAndSetCouple` (from the Geant4 class G4ProductionCutsTable) has been isolated in order to test and analyze if an improvement could be made. Basically, the method is a recursive algorithm used for propagating a new value for an attribute in the `G4LogicalVolume` object. A snippet has been developed to test the performance of this method with a huge `G4LogicalVolume`. This method is important only for the first information load and with few events, process where it takes a representative percentage. For a bigger number of events, the percentage is not representative.

5.2. CMS

There is no 64-bit implementation of this framework. Therefore, the 32-bit version was used for this work. There is a uniform performance when the number of events increases. A high percentage of L2 cache misses was observed for the generation stage (over 5%), unlike other stages. For example, the simulation stage has a percentage of L2 cache misses lower than 1%.

A final issue in the profiling results was a very high level of activity within one library: *pthread*. It is possible that the framework would try a run with more events and then come back to the program root with the results so that we can analyze them.

5.3. ALICE

There is a 64-bit version of ALICE software framework. A percentage over 6% of SIMD instructions were observed into simulation stage, as had been expected for this version. For the reconstruction stage, this amount is around 1%. Through a *pfmon deluxe* analysis into *simd1* mode, it is possible to determine the type of SIMD instructions executed: scalar simple and scalar double. From the *pfmon deluxe* standard analysis, there are not significant differences between simulation and reconstruction stages. Neither functions of interest were identified in the partial profiling results.

6. Related Issue

A problem with a lot of unresolved symbols was detected in the profiling results. It was caused by the fact that *pfmon* was never prepared to monitor 32-bit *dlopen* calls. As it was presented above, the CMS software framework used was the 32-bit version and that's when the failure was discovered. This problem does not happen with 64-bit versions. The functionality has been added (pfmon-3.4.x5) and in effect, it was possible to find more symbols resolved.

As far as the occasional 10, 20 or 50 unresolved addresses are concerned, in the typical case this is caused by rogue samples received very close to context switches. The first hipotesis is that latency issues make it impossible to be more accurate in this case. Anyhow, for 100.000 samples, having 10 or 20 unresolved, is a very good result.

7. Conclusions

A *pfmon* based methodology was developed in order to monitor the software frameworks for LHC experiments. Three basic steps compose the monitoring and tuning tasks: *pfmon* deluxe, *pfmon* profiling and application improvement.

The authors have presented a complete analysis of the software frameworks and the results have been sent to software developers in the different experiments in order to let them know about the requirements and bottlenecks of their tools.

As a solution to the results obtained from monitoring 32-bit version frameworks, a *pfmon* limitation was detected and a new functionality was added in order to avoid this issue.

8. Acknowledgment

The authors would like to thank Andrzej Nowak, who implemented the *pfmon* functionality to monitor the 32-bit frameworks, and Sverre Jarp (CERN openlab Chief Technology Officer) for their collaboration in this project.

References

[1] ALICE Off-line Project. Retrieved January 29, 2009, from http://aliceinfo.cern.ch/Offline/index.html

[2] Alpgen. Retrieved January 29, 2009, from http://mlm.home.cern.ch/mlm/alpgen/

[3] CERN openlab. Retrieved January 29, 2009, from http://openlab.cern.ch/

[4] Eranian, S. The perfmon2 interface specification. 2005

[5] Eranian, S. Quick overview of the perfmon2 interface. Retrieved January 29, 2009, from http://www.gelato.unsw.edu.au/archives/linux-ia64/0512/16211.html

[6] GCC, the GNU Compiler Collection. Retrieved January 29, 2009, from http://gcc.gnu.org/

[7] Geant4 Simulation Toolkit. Retrieved January 29, 2009, from http://geant4.cern.ch/

[8] Jarp S., Jurga R., Nowak A. Perfmon2: A leap forward in Performance Monitoring. International Conference on Computing in High Energy and Nuclear Physics, 2007.

[9] LHCb Computing. Retrieved January 29, 2009, from http://lhcb-comp.web.cern.ch/lhcb-comp/

[10] PYTHIA. Retrieved January 29, 2009, from http://home.thep.lu.se/~torbjorn/Pythia.html

[11] ROOT - An Object Oriented Framework For Large Scale Data Analysis. Retrieved January 29, 2009, from http://root.cern.ch/

[12] The CMS Offline SW Guide. Retrieved January 29, 2009, from https://twiki.cern.ch/twiki/bin/view/CMS/SWGuide

[13] The perfmon2. Retrieved January 29, 2009, from http://perfmon2.sourceforge.net/

[14] The pfmon tool. Retrieved January 29, 2009, from http://perfmon2.sourceforge.net/pfmon_usersguide.html