Journal of Physics: Conference Series **331** (2011) 042013 doi:10.1088/1742-6596/331/4/042013

The evolution of CMS software performance studies

M J Kortelainen[1], P Elmer[2], G Eulisse[3], V Innocente[4], C D Jones[3] and L Tuura[3]

[1] Helsinki Institute of Physics, P.O. Box 64, FI-00014 University of Helsinki, Finland
[2] Physics Department Princeton University, Jadwin Hall, Princeton NJ 08544, USA
[3] Fermilab, P.O. Box 500, Batavia, IL 60510-5011, USA
[4] CERN, Geneva, Switzerland

E-mail: matti.kortelainen@cern.ch

Abstract. CMS has had an ongoing and dedicated effort to optimize software performance for several years. Initially this effort focused primarily on the cleanup of many issues coming from basic C++ errors, namely reducing dynamic memory churn, unnecessary copies/temporaries and tools to routinely monitor these things. Over the past 1.5 years, however, the transition to 64bit, newer versions of the gcc compiler, newer tools and the enabling of techniques like vectorization have made possible more sophisticated improvements to the software performance. This presentation will cover this evolution and describe the current avenues being pursued for software performance, as well as the corresponding gains.

1. Introduction

The Compact Muon Solenoid (CMS) experiment at the CERN Large Hadron Collider [1] has now collected data for the first year. The computing infrastructure has performed well for the data taking, reconstruction and physics analysis. Nevertheless, the computing performance still needs attention in order to use the resources efficiently. In general the computing performance can be improved in two ways: by using more or faster resources, or by optimizing the software to run faster. The latter aspect is considered in this paper.

A crucial part of the software optimization process is to measure the quantities of interest. There are many quantities which can be measured, such as CPU time, memory allocations in bytes or in number of allocations, CPU/wall clock time ratio, I/O rates and patterns, event data size, application startup time, compilation time etc. Choosing the most relevant quantities is problem-specific. In this paper we mainly focus on the CPU time since in our CPU bound applications the wall clock time is dominated by it. We also focus on the memory allocations, as there is a strong correlation between the CPU time and the number of allocations.

There are many issues in a software system where improvements can be considered during the optimization process, for example the compiler, the implementation choices, and the design choices and the overall architecture of the software system. The compiler plays an important role in translating the higher level code to assembly or machine code. In the implementation some important aspects for an efficient programming are the algorithms, memory allocations and access patterns, and providing such high level program code that the compiler is able to generate efficient assembly. In order to obtain optimum performance, the high level design has to match the design of hardware architecture it is supposed to run on and performance measurements are

crucial to decide whether or not this is the case. In the era of modern multi-core and many-core CPUs the parallelization of the software becomes more and more significant. This can be achieved in several ways, for example by vectorization, multi-threading or multi-processing [2].

The paper is organized as follows. The evolution of the software performance studies in CMS and our observations are discussed in Section 2. Section 3 describes the challenges experienced in updating the CMS developed sampling profiler IgProf to the 64-bit x86-64 architecture. Section 4 introduces the Intel Performance Tuning Utility (PTU) and our progress in learning CPU performance events based analysis. Section 5 presents a new tool we have developed to share PTU reports on the web in our collaboration. Section 6 gives a summary.

2. Performance analysis

The effort to study the performance of the CMS software started with a dedicated campaign, run by a specially convened Task Force, for cleaning basic C++ errors and in particular to reduce dynamic memory churn. Since then, the model of work has evolved to be more systematic and has become an integral part of the software release process. Standard metrics have been established by making performance measurements on selected physics samples and these are now generated automatically as part of the twice-daily build procedure that integrates changes on a continuous basis (the so-called Integration Builds) [3]. Metrics are produced for CPU usage and memory footprint both as plain numbers and as IgProf profiles. These enable quick detection of any significant regression in performance.

In the beginning it was observed that over 20 % of the CPU time was being spent in memory allocation and deallocation [4]. Common causes for this dynamic memory churn originated from confusion in how `std::vector` works, the copying of large structures, dynamic memory allocation in tight loops, allocation of numerous tiny objects, having multiple copies of objects in memory, and using strings in inappropriate places. Routine monitoring has also helped to find these defects before they are included in the production releases.

Another common cause of software inefficiency has been the code size. Typical CMS applications have from about 500 to over 1000 shared libraries. We have seen in the previous studies [4, 5] that code bloat may be adversely influencing with the CPU memory hierarchy and thereby affecting CPU performance. We are therefore now starting tests in which software is packaged in a small number of large big binaries, such that shared libraries are used only for the external libraries.

We are also studying other factors that may influence performance, namely 64-bit architectures, the GCC compiler version update and vectorization. We are planning to change the production architecture of the CMS software from the 32-bit x86 architecture to the 64-bit x86-64 architecture, because we have seen improvements of 5–20 % in CPU time, as compared to 32-bit. The reasons for the improvement include that the x86-64 architecture provides more, and wider, registers and suffers from a smaller function call overhead, as compared to the 32-bit x86 architecture. The architecture update also includes the transition from the x87 math instruction set to SSE (Streaming SIMD Extensions), which significantly speeds up for virtually all match operations, especially with vectorization. Transcendental functions are a notable exception, the 32-bit x87 versions are currently much faster than any of the 64-bit SSE math libraries we have tried.

Our initial 64-bit tests showed very significant memory footprint increase, roughly doubling the virtual memory size. On closer inspection much of this turned out to be caused by 2 MB virtual memory alignment of shared library segments, versus 4 kB on 32-bit system. As we load hundreds of libraries, virtual memory size ballooned. Real memory footprint increase was 25–30 %, which is significant in itself and quite likely due to our data structures rich in pointers, and not at all optimized for data type alignment. However, because our environment accounts by virtual memory use, we decided to force shared library segment alignment to 4 kB.

At the moment our production compiler is GCC 4.3.4. Tests with GCC 4.5 are beginning. We are evaluating performance potential of the features planned for the next C++ standard (C++0x). The prospects in vectorization are being investigated, both in terms of support by the compiler[1] as well as by direct implementation in some CMSSW algorithms (e.g. geometrical vectors and rotations). In regard to these issues, performance measurements are just starting.

3. IgProf for x64-64 architecture

IgProf is a simple tool, developed by CMS, for measuring memory allocations and memory leaks, plus a sampling performance profiler [6]. It works in Linux, both in 32-bit and 64-bit x86 architectures, and it does not require recompilation of the program to be measured. The support for 64-bit has been added recently. The experience gained in the update are described in the following.

Originally approximately 10 % of the profile hits were lost due to stack unwinding issues. Stack unwinding is needed to determine the function call chain. Issues were found in the GCC compiler [7] and in the *libunwind* library [8].

It was found that GCC versions prior to 4.5.0 did not generate correct unwind information for function epilogues, and sometimes no unwind information was generated at all, which results in incorrect stack unwinds. This manifests as bogus addresses right above the profiling signal frame, or shortly thereafter. It also affects the ability of GDB to show reasonable stack traces. The required fix is to rebuild as much as possible with GCC version 4.5.0 or later. Especially in the computing applications it is important to note that also *libm* needs to be rebuilt. Unwind information for global constructors, both in C++ standard library and each shared library, was not generated by GCC.

The *libunwind* library had several problems. Unwinding through Procedure Linkage Table (PLT) did not work, for which a special handler was added. The library was not robust against bad addresses, for which we contributed a number of fixes. Unwinding through functions that were interrupted at function entry point was failing. This was fixed by tracking separately the *interrupted* and the *call* cases, so that now the library knows whether a frame was interrupted or a caller, and looks for the unwind information with the correct address. This was very important because function entry point is a popular place to trap profile signals, presumably because it comes after expensive cross-library transition. In addition, the unwinding on x86-64 was very slow which especially affected the IgProf memory profiler due to heavy-duty stack walking. We have contributed a patch for speeding up the tracing significantly, by a factor of 5–6.

4. Profiling with the CPU performance events

In the course of these studies we have learned what to do with the dynamic memory allocations and we know how to use tools like Valgrind [9] and IgProf to detect and measure them. A sampling profiler like IgProf is sufficient for providing a good high-level understanding of performance issues, but inadequate for line-by-line source code analysis. Hence tools providing much more detailed and localized performance data are very helpful. In addition, the modern CPUs and the memory architectures are becoming more and more complex, thus complicating interpretation of the reports.

Fortunately modern x86 CPUs from both Intel and AMD have for many years included a so-called Performance Monitoring Unit (PMU). The PMU provides counters for various performance events, such as the number of retired instructions, the number of unhalted and stalled cycles, for cache hits and misses, and for local and remote memory accesses for Non-Uniform Memory Access (NUMA) architectures. Most recently we have been testing two tools for the performance events, namely Perfmon2 for Linux kernel [10], and the Intel Performance

[1] For example `-ftree-vectorize` in GCC

Tuning Utility (PTU) [11], which is discussed further below. In HEP software some functions or classes are typically called multiple times but with different arguments and sometimes it would be necessary to see the impact of the arguments on the performance. PTU is usually used to measure the performance of the software as a whole such that this kind of differentiation is not possible. Perfmon2 can be used to collect the event counts for modules with different arguments with the aid of a new tool [12].

The Performance Tuning Utility is a commercial product from Intel available for Linux and Windows. It is a plugin to the Eclipse IDE, and for Linux it provides a kernel module for the PMU interaction. This is in contrast to Perfmon2 which requires a patch to the kernel. The PTU is able to aggregate the event counts per process, module, source file and function. It can also show the event counts for source lines, basic blocks and even assembly lines, although the precision of the event position is limited in these cases and can be even 50 instructions off, depending on the event in question. By default the event counts are provided in a spreadsheet, which can be exported to a comma separated value (CSV) file. There are also more sophisticated analysis tools, such as a graph of the basic blocks, cache line access distributions and differences of two profiles.

We have found the ability to actually see from the retired instructions and unhalted cycles events what code is executed to be very interesting. We have been able to identify and fix functions consuming large number of cycles combined with a high ratio of cycles per instructions, or CPI. We have also found and reduced the number of divisions and square roots by restructuring the calculations, as these operations are clearly quite costly.

We continue to observe CPU looks to be instruction starved. Now half of the cycles of a typical reconstruction job are spent in front end decoder stalls. This is still poor, but an improvement to 2/3rds time spent in stall we saw earlier [4]. We have not yet analyzed the effect in detail, but the known causes include bad branch prediction performance and high sensitivity to instruction cache misses. Possible sources include code size and locality, and function pointer chasing caused by e.g. virtual functions in C++. Single big binaries are expected to yield more insight to this problem too.

We have found PTU a very valuable and powerful tool, and specifically provides crucial further insight to sampling profiles. Effective use however requires one to learn much more about CPU architectures.

5. Web viewer for PTU reports

The reports produced with PTU can be analyzed in PTU itself, or exported to spreadsheet files in CSV format as mentioned in the previous section. However, neither of these methods is adequate for sharing the reports in a large collaboration. Hence we started to write a tool named *ptuview* [13] to show the basic reports in a web browser. The tool is a CGI script written in Python. It utilizes JavaScript for some user interface improvements like table sorting and collapsing and expanding of some elements, but it is fully usable also without JavaScript. The spreadsheet files exported from PTU are used as the data store. The implemented reports contain the list of events per function, and also the source and assembly code lists. It has also a tree diagram of the total event counts for a predefined set of events to allow intuitive navigation through them. We hope this to be valuable for those users who are not yet familiar with all the details in performance events.

6. Summary

The CMS collaboration has made a strenuous efforts to measure and improve the performance of its offline software. The work was started by a dedicated task force that has coordinated software improvement campaigns. Performance measurements have become an integral part of the release integration and testing process. Tools are an essential ingredient and efforts continue

to improve existing tools, such as IgProf, and to acquire new tools, such as Perfmon2 and the Intel Performance Tuning Utility (PTU). PTU has proved to be particularly valuable and therefore CMS has developed a new tool, *ptuview*, that eases the sharing of performance reports amongst the members of a large collaboration.

Acknowledgments

For support, space and contributions to discussion we would like to thank the rest of the CMS Offline Software Group, the CERN MultiCore R&D group, and Dr. David Levinthal. M J Kortelainen would like to gratefully acknowledge the Waldemar von Frenckell Foundation, the ACEOLE project, and the Graduate School in Particle and Nuclear Physics (GRASPANP) in Finland. This work was in part supported by Department of Energy of the U.S.A.

References

[1] CMS Collaboration 2008 The CMS experiment at the CERN LHC *JINST* **3** S08004
[2] Jones C D, Elmer P, Sexton-Kennedy L, Green C and Baldooci A Multi-core aware applications in CMS *these proceedings*
[3] Sexton-Kennedy L Release Strategies: CMS approach for Development and Quality Assurance *these proceedings*
[4] Tuura L, Innocente V and Eulisse G 2007 Analysing CMS software performance using IgProf, OProfile and callgrind *J. Phys: Conf. Series* **119** 042030
[5] Eulisse G, Tuura L and Elmer P 2010 HEP C++ meets reality *J. Phys: Conf. Series* **219** 032007
[6] Eulisse G and Tuura L A 2004 IgProf profiling tool *CHEP04, Computing in High Energy Physics (Interlaken)* See also `http://igprof.sourceforge.net/`
[7] GCC home page `http://gcc.gnu.org/`
[8] The libunwind project home page `http://www.nongnu.org/libunwind/`
[9] Nethercote and Seward 2007 Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation *Proc. PLDI 2007 (San Diego, California, USA)* See also `http://valgrind.org/`
[10] Eranian S 2006 Perfmon2: a flexible performance monitoring interface for Linux *Proc. of the 2006 Ottawa Linux symposium (Ottawa)* See also `http://perfmon2.sourceforge.net/`
[11] Intel PTU home page `http://software.intel.com/en-us/articles/intel-performance-tuning-utility/`
[12] Kruse D and Kruzelecki K Modular Software Performance Monitoring *these proceedings*
[13] Ptuview home page `http://mkortela.web.cern.ch/mkortela/ptuview/`