# Analysing CMS software performance using IgProf, OProfile and callgrind

**L Tuura[1], V Innocente[2], G Eulisse[1]**

[1] Northeastern University, Boston, MA, USA
[2] CERN, Geneva, Switzerland

**Abstract.** The CMS experiment at LHC has a very large body of software of its own and uses extensively software from outside the experiment. Understanding the performance of such a complex system is a very challenging task, not the least because there are extremely few developer tools capable of profiling software systems of this scale, or producing useful reports.

CMS has mainly used IgProf, valgrind, callgrind and OProfile for analysing the performance and memory usage patterns of our software. We describe the challenges, at times rather extreme ones, faced as we've analysed the performance of our software and how we've developed an understanding of the performance features. We outline the key lessons learnt so far and the actions taken to make improvements. We describe why an in-house general profiler tool still ends up besting a number of renowned open-source tools, and the improvements we've made to it in the recent year.

## 1. The performance optimisation issue

The Compact Muon Solenoid (CMS) experiment on the Large Hadron Collider at CERN [1] is expected to start running within a year. The computing requirements for the experiment are immense. For the first year of operation, 2008, we budget 32.5M SI2k (SPECint 2000) computing capacity. This corresponds to some 2650 servers if translated into the most capable computers currently available to us.[1] It is critical CMS on the one hand acquires the resources needed for the planned physics analyses, and on the other hand optimises the software to complete the task with the resources available. We consider this latter aspect in this paper.

The CMS software still requires considerable optimisation to fit the resource budget which allows a single event reconstruction to take 25'000 SI2k seconds, or approximately 8.3 seconds on the above-mentioned current reference server. In the CMS CSA06 challenge [2] less than a year ago, in the reconstruction of ttbar samples considered representative of the computation need at the start-up, our software used approximately 20'000 SI2k seconds per event, *however this was with incomplete algorithms and no pile-up.* The most recent CMS reconstruction takes 12'000 SI2k seconds per event for QCD 20–30 GeV "standard candle" sample, again without pile-up, or about 3.8 seconds per event on our top-performing reference server. The reconstruction times for real events will be several times the time for reconstruction without pile-up. Thus we exceed the allowed envelope by a large margin.

Optimising CMS software is therefore necessary, but a challenging task for several reasons:

[1] An Intel Xeon 5160 at 3 GHz, 3065 SI2k per core, two dual-core CPUs per server.

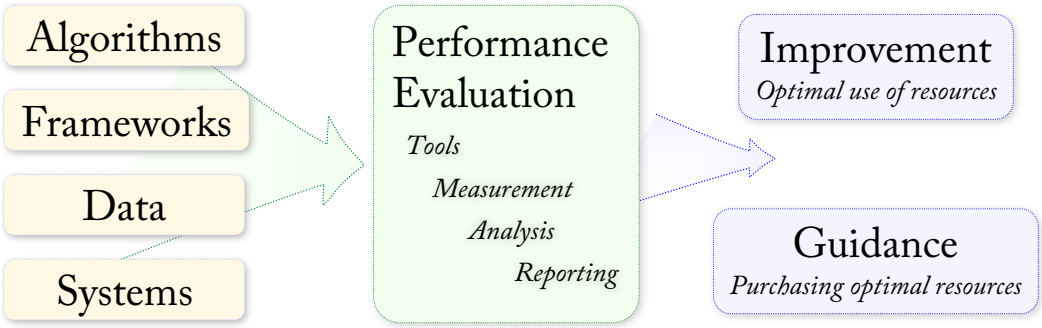



**Figure 1.** The CMS performance improvement task.

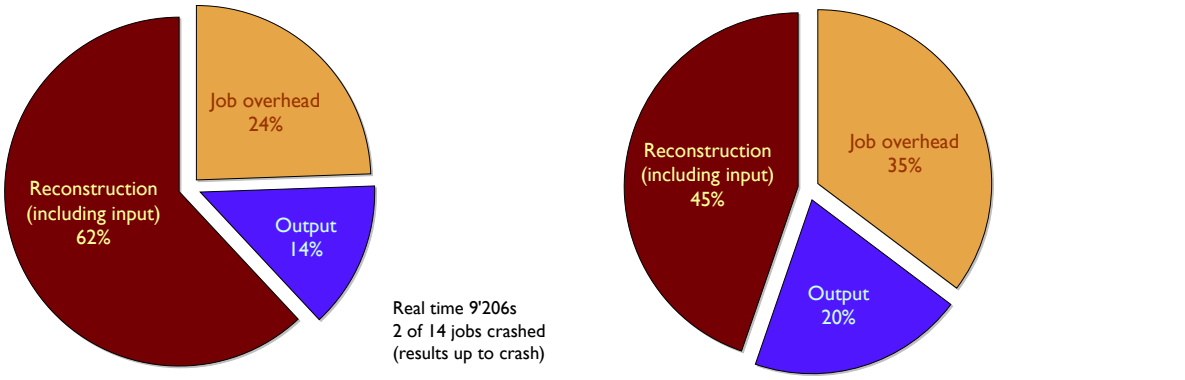
- The CMS software release consists currently of approximately 1.5 million lines of code, not including the external packages we use. In the current development phase existing code evolves and new code is added much faster than it is possible to analyse and improve what is already there.
- There is no immediately obvious *technical* metric to indicate when the code is sufficiently well optimised. The available computing budget as discussed above of course sets practical limits, as does the desire to out-perform competing experiments. These do not really help us give a large number of non-professional software developers technical advice on which specific aspects need improvement and why, or when their software can be considered well-optimised.
- There is limited practical experience in high-energy physics community in producing *high-quality and high-performance* designs for object-oriented C++ applications. This is mostly because C++ is still relatively new to high-energy physics, which lead the past discussion to focus much more on the options and merits of different object-oriented designs, and less on making designs well adapted to the systems we have.
- Present software development tools are not for the kind of software CMS develops. CMS software is in C++, we run large processes for a very long time, use multi-threading, and load dynamically hundreds of shared libraries. Precious few tools are capable of dealing with the combination of all these factors. We have not come across on any platform a call tree profiler, open source or commercial, other than our own modest tools, which would be able to extract useful insight about our applications with reasonable effort and within reasonable amount of time. Some tools appear to emerging but most are not yet mature for professional use.

## 2. Summary of studies and issues

### *2.1. Scope of the present efforts*

CMS begun a focused, organised project for code performance improvement in the beginning of 2007. The scope and the structure of the project is illustrated in Figure 1. Effectively everything concerning the computational algorithms, frameworks, data structures and how we use persistent data, as well as the actual computer systems we use are under scrutiny.[2] The project is responsible for establishing the tools and methodologies for performance analysis and for generating understandable reports in a timely manner for each software release. The group

[2] This leaves out mainly various database-related and build-system related optimisation issues. These are addressed more directly in other projects.




(a) Actual time division

(b) Time division if reconstruction doubles in speed

**Figure 2.** The division of time spent in CMS reconstruction with 14 reconstruction jobs, each reconstructing 100 events from different physics samples, and what the chart would look like if reconstruction doubled in speed but all the rest stayed the same.

advices developers on making improvements such that CMS resources are used in an optimal manner, up to and including giving help with the actual re-design and re-coding. The group also documents which resources are best suited for CMS software at any one time, and why.

We emphasise the project's essential property is to establish the means to *reliably measure and analyse* the behaviour of our software. We do not want to guess where problems might be, but to make well-informed changes to the system, investing optimisation effort where it is known to make a difference, and to address early the areas which pose the highest risk to CMS.

### *2.2. Progress accomplished*

So far, the project has established a suite of tools to be used at this time, and has completed the first round of analysis. We will describe below the main findings and the corrective actions triggered. We are currently working on analysing both the impact of these changes and new code introduced in the mean time. Separate studies are going on in areas such as how we store and access our persistent data, but these are not yet sufficiently advanced to be reported on.

While a broad range of analyses have been carried out to hone our methods and tools, our most significant results by far originate from *memory profiling* analyses. These have driven practically all the major improvements that we will discuss below.

### *2.3. Principal findings*

Our first finding is that the CMS software is effectively free of buffer overrun problems. These were in fact rather frequent in the past, where clever but incorrect buffer optimisations caused data from one event to poison the data in another event. These turn out to be rare now, much to our delight. We credit this to both valgrind [4] and the designs which effectively avoid this by construction. We have however traded for another problem in the process as we will see shortly.

Our second main finding is that the CMS software, as well as practically all the external software it uses, has considerable room for improvement, and specifically in the area of memory management. On a positive note, just about everywhere we have looked there are optimisation opportunities, which is encouraging considering we are currently well over our the CPU capacity we have budgeted for. On a slightly negative note, this means we get to revise numerous design decisions and to change practically all of our 1.5 million lines of code, as well as negotiate

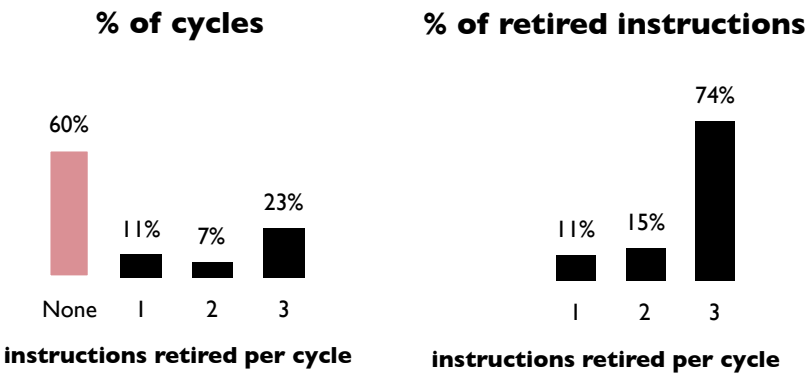



**Figure 3.** Statistics on how many instructions are retired on average per clock cycle, and what fraction of all instructions are represented by each retirement group. This shows the CPU is stalled most of the time, and tends to work in bursts, filling the CPU pipelines to capacity only to be drain soon on a stall.

improvements with the providers of the external packages we use. Of course as the code is changing anyway, this is not completely fatal and can be addressed to a large extent with an effective execution plan.

At the highest level we find our software is not spending as much time in effective work as we would like it to be. As Figure 2(a) indicates, overhead from both job setup and I/O is disturbingly high. Considering we are hoping to significantly speed up the reconstruction performance, the irreducible overheads are a major concern as illustrated by Figure 2(b). The per-job overheads could of course be mitigated by running longer jobs, but in practise are a real pain and represent significant time lost every day. We are studying what can be done to reduce the I/O costs.[3]

Our fourth main finding is that *a single factor dominates the application performance: a staggering memory allocation and deallocation rate.* Practically every CMS application allocates and deallocates memory at the dizzying rate of 700k–1M blocks per second, or about 1 GB and 10 M allocations per event. A quarter of all time is spent in memory allocation *(operator new, malloc, free...),* and a third if we include memory and string shuffling *(strcpy, memcpy, ...).*

While we have identified a couple of particularly egregious causes, this problem is not confined to any particular package or coding style. Nearly all high energy physics C++ code we have examined, both CMS' own code and the externals we use, have large-scale and wide-spread issues with memory allocation and usage patterns. Looking at the profile data it is not at all evident the applications should be crunching numbers. There is in particular an odd affection for strings almost everywhere.

In our system level studies we have made three interesting discoveries. As expected from the memory management issues, a large first level and page table cache substantially improves performance of CMS software; this is the primary reason why our code performs so much better on AMD CPUs and Intel's Core-2 CPUs than on Intel's Pentium-4 line. We also find that when the operating system claims the CPU is 100% busy, the CPU is, in fact, completely idle for about two thirds of the time, as shown in Figure 3. We theorise this also leads to the

[3] Do note the graph shows *CPU time spent,* that is, the slice for I/O is the actual overhead in the job to format and prepare the data for I/O. The CPU was 100% occupied in these tests so there is no I/O idle time to consider here.

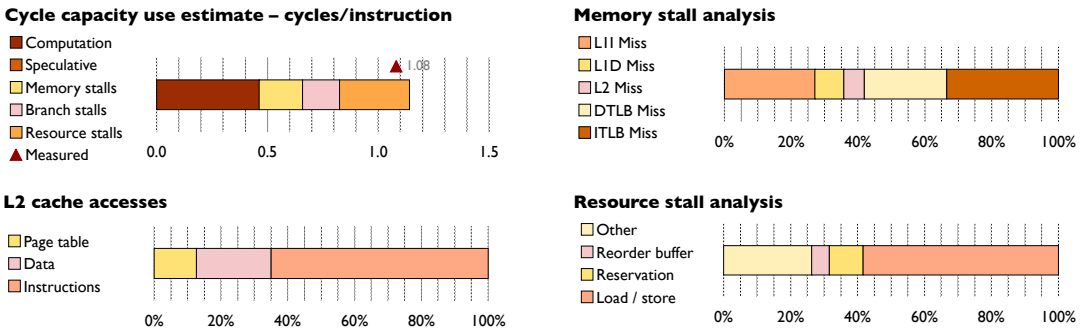



**Figure 4.** System level CPU cycle usage measurements on AMD Opteron 270. On top left we see this wide and resource-rich CPU is not at all computation bound. On right we see the breakdown of the main two stall categories, memory and resource stalls. 60% of memory stalls are due to instructions and 60% due to page table accesses, while most resource stalls are in fact memory-related stalls as well. On bottom left we see that instructions dominate the accesses to the second level cache.

deceptively low CPU-memory bus[4] utilisation, the bus is used well below 1% of its capacity. In any case this seems to indicate there is a possibility to extract much more performance from existing hardware.

The most interesting discovery from the system level analysis was that on a wide and resource-rich CPU our main problem appears to be *excessively large code size*. Figure 4 shows how the CPU is not at all computation bound, and that the prime reason for the stalls is memory references—but that 60% of memory stalls originate from code, 60% are for page tables, and instructions dominate second level cache accesses. We concluded from this that we may have considerable issues with both the absolute code size and poor locality of both code and data. On further investigation we found that a reconstruction process consuming 500 MB of memory could have as much as 150 MB of code present in memory: nearly a third of just code. At present it looks like we have far too many shared libraries, far too large a "code working set" in many key algorithms leading to page table overflow, the code organisation for the libraries is poor and there is much redundancy at the machine code level.

### *2.4. Principal remedies*

So far our optimisation efforts have focused on resolving the worst memory management issues. We find it frequently that the memory management costs are not clearly visible in normal performance profiles, however the memory profiles point very clearly to the most suspect pieces of code. Once the memory management is brought under control, other issues are unmasked and the more traditional performance profiles begin to become valuable again. Hence the work progresses in cycles of sorts.

Our first major optimisation was very low-hanging fruit, if not easy to tackle. We discovered that 66% of all memory allocations were due to a matrix and vector package used in reconstruction algorithms. We had long considered and debated the replacement of this package, however the cost of the migration made this very unattractive. Our findings finally convinced us to bite the bullet and overcome the various technical and political arguments. The first wave of the migration, for the parts most easily changed, required changes to some 50 core reconstruction packages, and a number of delicate changes such as switching from 1- to 0-based

[4] HyperTransport in the case of AMD and the more traditional CPU bus for the current Intel CPUs.

array and vector indexing. Work is ongoing to complete the migration in code requiring more extensive effort.

Together with other optimisation opportunities discovered in the process, such as caching computed values and magnetic field lookups, the performance of the tracking code was improved by about a factor of three. Reconstruction as a whole was unfortunately sped up less than this due to unrelated less optimised algorithms introduced independently at the same time.

### *2.5. Discussion*

The memory churn we have discovered appears to have numerous origins. Packages developed with very different styles and methods exhibit the problem, each in their own way. We identify some common causes below.

- Excessive use of strings everywhere, and in just about every possible wrong way.
- Poorly chosen method names, spaghetti logic, poor encapsulation and unclear object ownership rules. As a result, developers are confused about which object owns what data, or what state objects might be in, or the interactions between different methods are too complex to track through, or it's unclear what a method does or returns. The result is that developers don't reuse previous data or clone objects excessively. In some rare cases there is little wrong with the APIs used, and it is the developers who need to be educated to make less mistakes.
- Manipulating very expensive objects by value. One very common pattern is using containers of objects that contain, or are themselves, containers, such as vectors of (objects containing) vectors, maps of strings, and so on. Not passing big parameters and return values by value is mentioned in every text book, yet rather common.
- Constantly recalculating values. Could be because the local scope knows too little to effectively reuse state or to cache expensive computations. Frequently `x().y().z()` call chains where intermediate calls are not as cheap as the developer thought. Can involve encapsulation gone too far and limiting useful horizon in the system. Can be a side effect of shattering the code in thousands of three-line routines all over the place such that the compiler is unable to eliminate common sub-expressions.

## 3. Analysis and reporting methods

### *3.1. Tool suite overview*

CMS uses both open source and in-house performance analysis tools. For performance analysis we use mainly three different tools, depending on the desired precision. At the very coarse end CMS framework providers timers which allow the execution of each algorithm to be timed. This is the profiling tool every physicist will understand and use easily. Most of the time the more advanced developers use the intermediate-precision tool, IgProf [3], because it is fast and provides a suitable level of detail. For maximally detailed code scrutiny we use callgrind from the valgrind family of tools [4]. We make little use of CPU performance counters in algorithmic performance evaluation due to the nature of our performance problems, but hopefully that will change one day.

For memory profiling we use mostly IgProf and valgrind a little bit. We usually measure three quantities: the total memory allocated, the total memory leaked, and each algorithm's largest working set size. The IgProf memory profiler has been instrumental in our analyses as it brought into light the memory management problems of our applications—the root cause was not evident in the normal performance profiling we did.

For system level analysis with the CPU performance counters we started with OProfile [5], mainly because it is well known, conveniently available for the Scientific Linux kernels we use and because there is plenty of documentation on how to make simple analyses. However as

our analyses gained depth, we found ourselves patching and improving OProfile and begun investigating alternatives. The new Linux perfmon [6, 7] would have been the ideal choice, but is not available for the kernels we have to use. We settled on perfctr [8] whose user interface is simple and raw, but allows us to make all the necessary measurements. We augment the data with `numastat` numbers. The biggest disadvantage of these kernel tools is that they are rarely installed, let alone enabled, requiring specially configured test servers. Furthermore the use of OProfile requires super-user privileges, rather a big minus in practise.

Our tool suite for analysing data persistency issues is growing slowly. At present the most important tools profile the structure and the sizes of the various persistent data products.

We generate digestible performance reports from IgProf and callgrind data using a fairly general PerfReport tool we wrote ourselves. Its purpose is to generate comprehensible summary reports, the most important of which is a reference report for every software release. This tool is currently being extended to provide an interactive profiling data repository available as a web service.

Finally, we restructured the CMS release validation suite such that every process step can be tested independently, and can be instrumented for various forms of performance analysis. This is also integrated with the reporting. We have established a "standard candle" sample and defined a reference processing chain that is used to evaluate every CMS release. The reference reports include comparisons with other releases for easier tracking of improvements and regressions.

### *3.2. IgProf*

IgProf was developed in CMS for measuring and analysing application memory and performance characteristics. It was born mainly out of frustration with other tools and to fill a dire gap between valgrind and system level profilers. It incorporates ideas from many other tools as we have described previously [3]. It requires no changes to the application or the build process, and no special privileges to run. IgProf is one of the few profilers capable of correctly profiling CMS' C++ software. It is fast, light-weight and correctly handles dynamically loaded shared libraries, threads and sub-processes started by the application. It generates full call tree profiles which can be filtered in many ways in a later analysis stage.

The main strengths of IgProf are its speed and efficiency. The *statistical performance profiler* adds about 40 MB to the memory usage and negligibly ($\leq$1%) to the run time. The *accurate memory profiler* adds 50–75% to the run time and about 250 MB to the memory use for a typical CMS task loading 400 or so shared libraries, running for an hour, using some 500 MB memory and making around 1M memory allocation calls per second. IgProf is typically 10–100 times faster than valgrind or callgrind.

### *3.3. Callgrind*

Callgrind collects a very precise execution time profile by simulating the execution of the code on a virtual processor. The profile is accurate, not statistical as the IgProf performance profile. The tool is very robust and rather flexible, and deals well with the complexity of CMS software. The most productive way to use callgrind is to enable it only for a particular algorithm module; we provide a simple framework service that allows users to easily specify the parts to profile. We also extended callgrind to include complete call stack information as we noticed that it tended to mis-estimate function call costs due to the gprof-like profile structure it maintained. We find it is not really practical to profile entire CMS jobs with callgrind, the runs take far too long and the analysis of the entire CMS release validation would take days.

We initially used KCachegrind [9] as an analysis studio for studying callgrind results. However we saw several ambiguous call costs attributed to functions, and found they were due to the gprof-style representation of the call graph. While we introduced complete callstacks to callgrind, in our estimation introducing them to KCachegrind would have implied a near-complete rewrite,

and we switched to developing our own reporting tools, for example to generate the reports without a GUI.

## 4. Conclusions

In the last half a year we have taken our first steps at organised and determined software performance scrutiny and improvement. We are clearly only at the start of a very long road. It could be said we have so far mainly learned a productive method for understanding the performance of our software, and how to make controlled improvements. This in itself is very encouraging however.

We conclude there appears to be room for competitive improvement on existing hardware and even an option to extend the physics range. Our unexpected findings confirm it is important to first measure and analyse.

There is clearly a lack of mature and effective tools for analysing the performance of modern, complex software systems. We are making publicly available the tools we find useful ourselves: IgProf, PerfReport and our improvements to callgrind. Our methodology for measuring system level performance is also available to any interested party. We find it now much easier to identify bottlenecks in our software than when we started.

We continue to watch with interest developments elsewhere: strategies and tools of large open source projects such as KDE, Samba and Linux, compiler evolution, and C++ support infrastructure. We follow closely the gradual but significant developments taking place on the CPU market.

## Acknowledgments

Identifying the problems is only useful if the issues can be addressed. The credit for addressing the CMS performance issues should largely go to the developers having completed these difficult migrations so promptly. The authors thank the CMS software developers for their constructive response to the difficult issues we have found at a stressful time, and the project management for supporting this demanding task.

## References

[1] CMS Collaboration, "Technical proposal", CERN/LHCC 94-38, 1994.
[2] Fisk, "CMS Experiences with Computing Software and Analysis Challenges", these proceedings.
[3] Eulisse and Tuura, "IgProf profiling tool", Proc. CHEP04, Computing in High Energy Physics, Interlaken, 2004
[4] Nethercote and Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", Proc. PLDI 2007, San Diego, California, USA, June 2007. See also `http://valgrind.org`
[5] OProfile home page, `http://oprofile.sourceforge.net`
[6] Perfmon home page, `http://www.hpl.hp.com/research/linux/perfmon`
[7] Jarp and Jurga, "Perfmon2 - A leap forward in performance monitoring", these proceedings.
[8] Mikael Pettersson, PerfCtr home page, `http://user.it.uu.se/~mikpe/linux/perfctr`
[9] Weidendorfer, "Performance Analysis of GUI applications on Linux", KDE developer conference Kastle, 2003. See also `kcachegrind.sourceforge.net`