Analysing CMS software performance using IgProf, OProfile and callgrind

Lassi A. Tuura

Northeastern University

With V. Innocente and G. Eulisse
on behalf of the CMS experiment

CHEP 2007

Victoria, BC, Canada

2-7 September 2007

We use a lot of money for computing

1% of 2008 CMS CPU budget
≈ 27 today's top performing servers
(2 × dual core Intel Xeon 5160 @ 3 GHz)

The CMS computing model allows
25k SI2k × s / event for reconstruction
≈ 8.3 s on a 3 GHz Intel Xeon 5160

In the CMS CSA06 challenge we used
20k SI2k × s / event on ttbar, however
with incomplete algorithms and no pile-up

The very latest CMS reconstruction takes
~12k SI2k × s / event (3.8 s on 3 GHz 5160)
for QCD 20-30GeV "standard candle"
with no pile-up

With pile-up the reconstruction
is still well over the time budget!

Possible solutions

Decide inclusive QCD
wasn't so interesting after all.

Possible solutions

Drop a detector upgrade or two
and buy more computers.

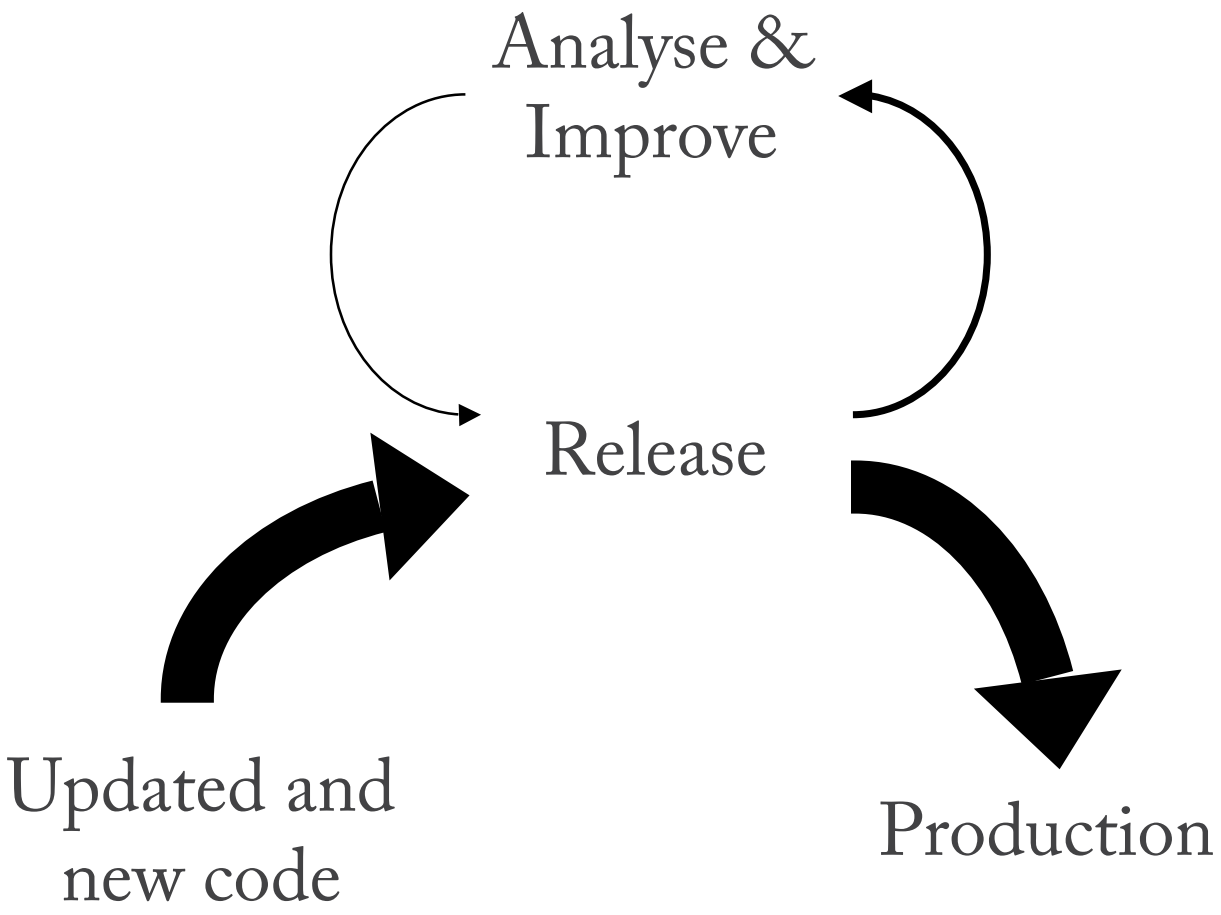
Possible solutions

Improve software performance and
maybe even gain more physics capacity!

But that's actually quite hard.

CMS release is currently 1.5M lines of code,
not including the external packages we use.
In the current phase of development, existing
code is modified and new code is added faster
than we can analyse and improve what is there.

The essential optimisation challenge



(The other optimisation challenge)

"But in our enthusiasm, we could not resist a radical overhaul of the system, in which all of its major weaknesses have been exposed, analyzed, and replaced with new weaknesses."

Bruce Leverett

Register Allocation in Optimizing Compilers

Moreover…

Current state of the art video editing software is capable of applying multiple video effects in real time. This is a good example of well optimised code.

Is it realistic to assume we could reach a similar degree of code performance?

If not, *what is* a realistic goal?
Which *reality checks* benchmark us?

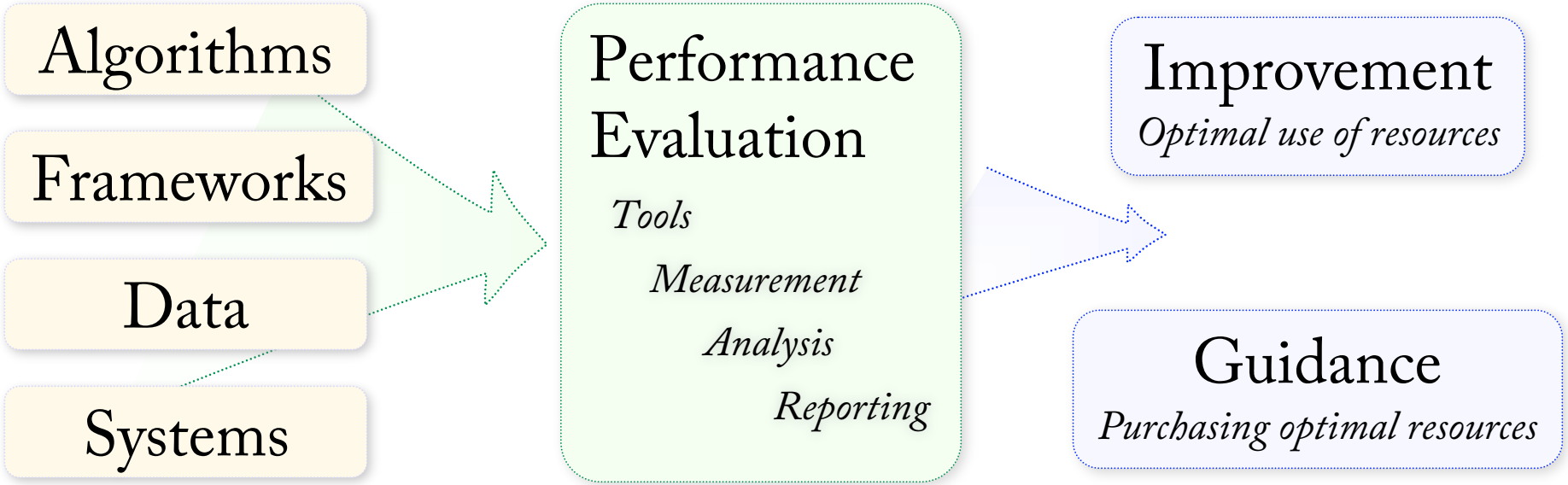
Some interesting facts

Each major bug cuts production efficiency instantly by O(50%). That's a tough envelope.

Memory is relatively cheap. Or was. Chasing pointers is poison to the CPUs, page table working set size is limited and >> 1 GB per job slot gets tenuous.

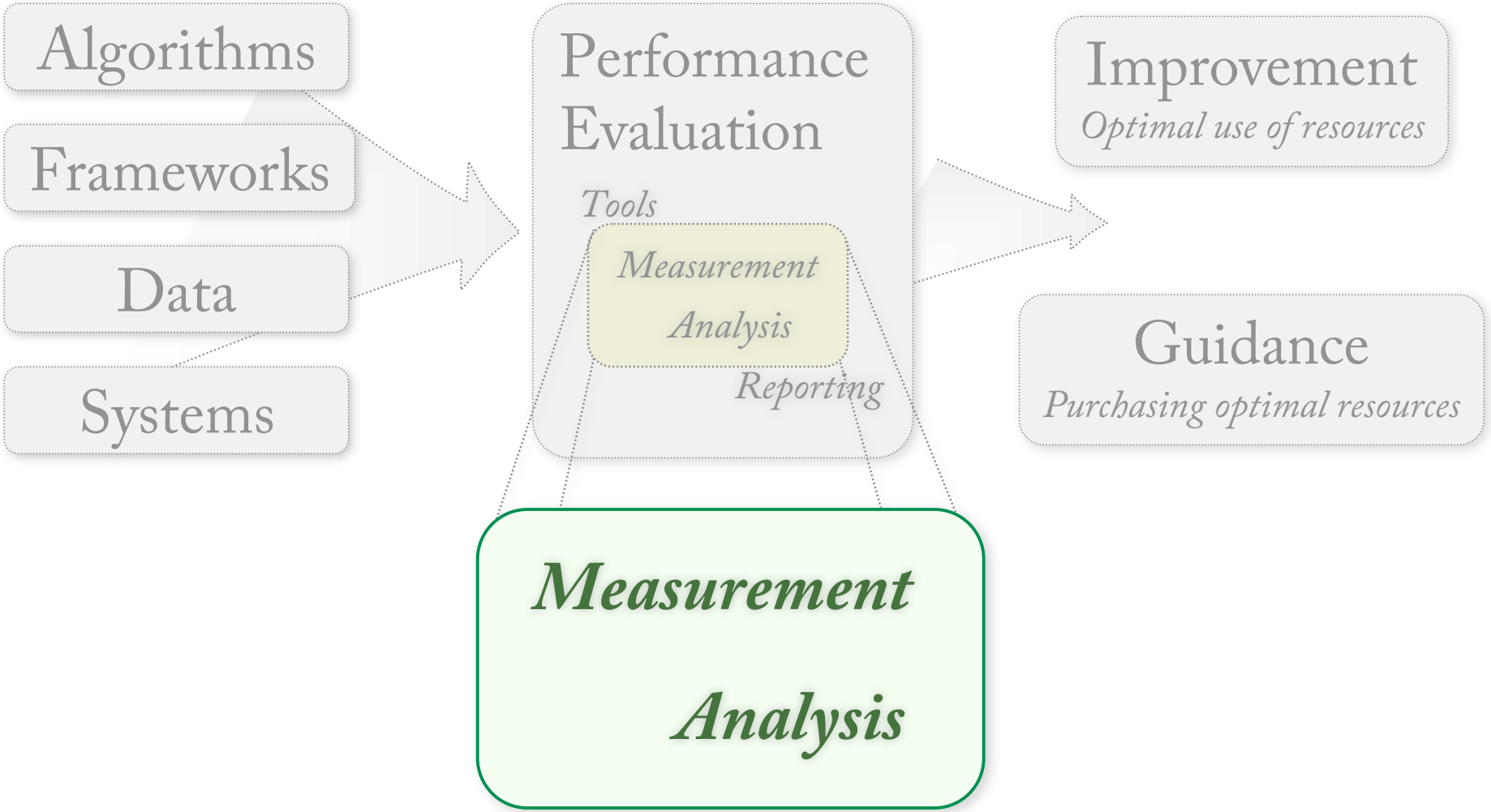
C++ is still relatively new in physics and in terms of tool maturity. A lot of attention went in the past into debating and honing OO designs —not necessarily high-performance designs.

Recent CMS code performance projects



"Measure, don't guess!"

Use reliable hard data and proper methodologies



20’000 ft view

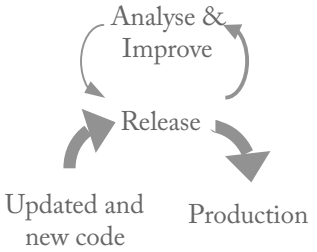
Excellent news: Remember the problems with “clever” optimisations causing data from one event poisoning another? Well, gone. Turns out to be a negligible problem in CMS software, both by design and thanks to valgrind.

Brilliant news: We are making good progress towards the computing model envelope! The most important tracking algorithms were speeded up by × ~3 and the high-level trigger has demonstrated reconstruction within the allotted 40ms/event budget.

So were there any bad news?

Well, think children and candy stores.
It's an optimisation paradise.
We have much to gain.

It means we get to revise many design decisions and to change nearly all the 1.5M lines of code one way or another—not that the code wouldn't be changed anyway…



And then there are questions on some externals we use.

Let's dive into the details.

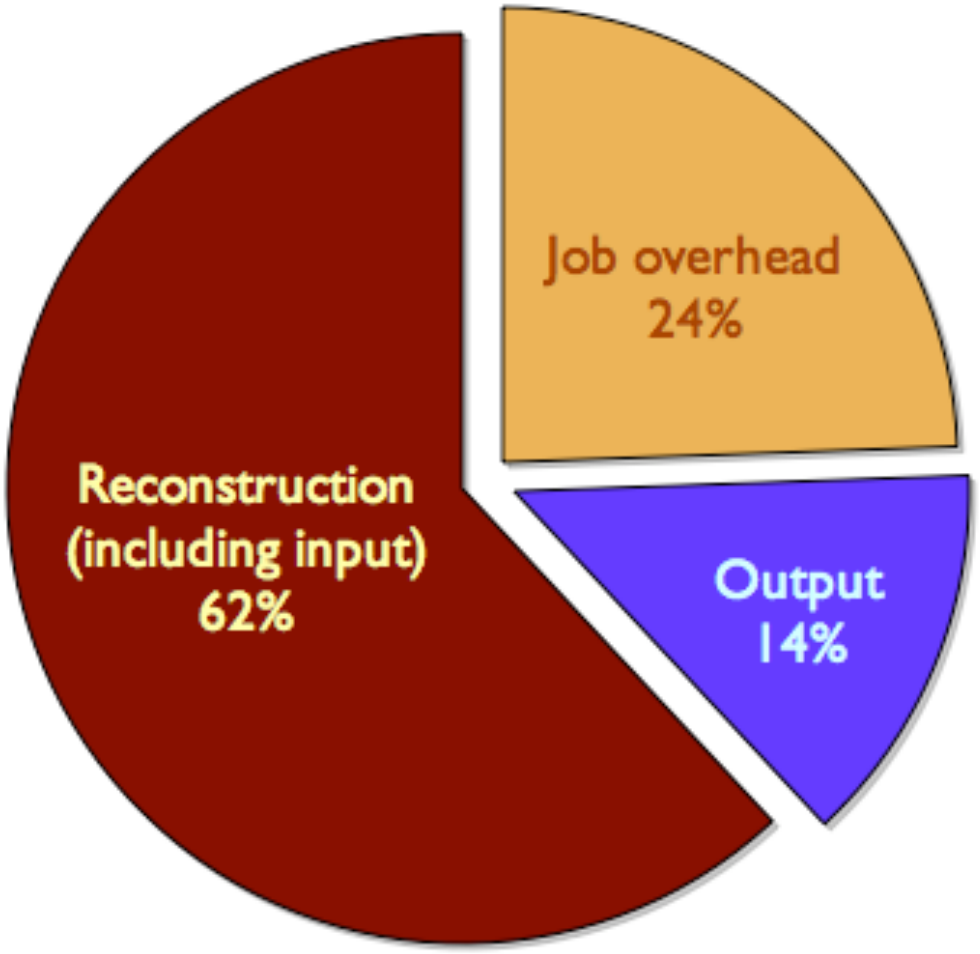
- What we found out.

- What we did about it.

- Tools and methods we used.

- A glimpse at ongoing analyses.

The starting point



Real time 9'206s
2 of 14 jobs crashed
(results up to crash)

We have discovered a single dominating factor:
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.

1/4th of all time spent in memory allocation *(operator new, malloc, free…)*, 1/3rd 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.*

Profile data rarely reveals significant number crunching in the applications. There is in particular an odd affection for strings almost everywhere.

In recent months our top optimisation priority has consistently been the identification and addressing of memory issues.

This often implies significant code changes. On the other hand the process usually reveals further important optimisation opportunities: the memory usage patterns frequently mask genuinely interesting problems elsewhere.

The first optimisation step was very low-hanging fruit, if not easy. 66% of all memory allocations were by a matrix and vector package used in the reconstruction algorithms.

Replacing the package with a more memory-efficient one in the most easily accessible performance sensitive parts required changes to ~50 packages and delicate changes such as switching from 1- to 0-based indices. Work is ongoing to complete the transition in code requiring more extensive effort.

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

➙ #241 ➙ #374

The memory churn has many sources; packages developed with very different styles and methods exhibit the problem. We identify some factors below.

–Strings everywhere, and in just about every possible wrong way.

–Bad method names, spaghetti logic, poor encapsulation and unclear object ownership rules. 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. Result is not reusing previous data and excessive cloning.

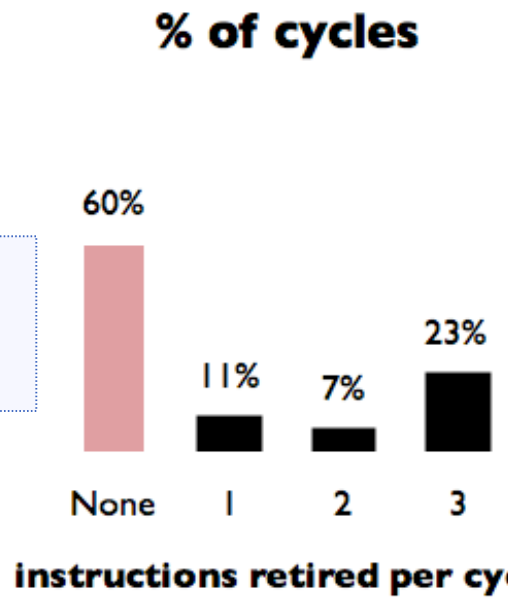
–Manipulating very expensive objects by value. One very common pattern is using containers of objects that contain, or are themselves, containers, such as vectors of (objects containing) vectors, maps of strings, and so on. Not passing big parameters and return values by value is mentioned in every text book, yet rather common.

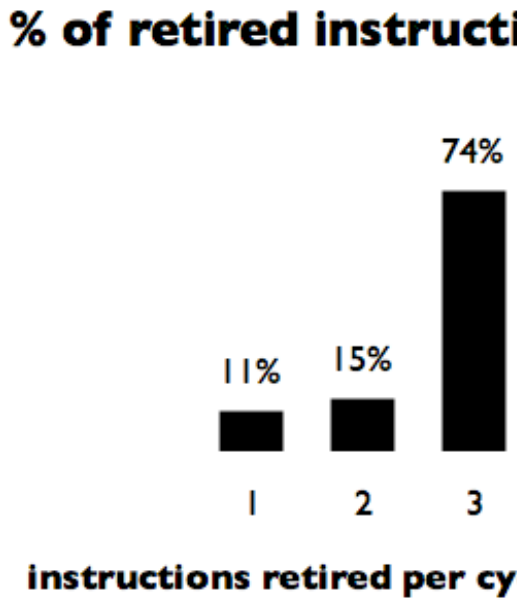
–Constantly recalculating values. Could be because the local scope knows too little to effectively reuse state or to cache expensive computations. Frequently "x().y().z()" call chains where intermediate calls are not as cheap as the developer thought. Can involve encapsulation gone too far and limiting 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.

At the system level, *large Level-1 and page table caches (TLBs)* improve CMS software performance substantially. Deceptively low CPU ⇔ memory bus use, <<1% of capacity, is perhaps best explained by the fact that *the CPU is mostly stalled.*

These are, effectively, just restatements of the woeful state of memory management affairs. The first level cache obviously takes the hit from dereferencing all the millions of pointers we create. Large page table caches help mitigate the large, sparse memory page working sets.

Most of the time, the CPU does nothing…

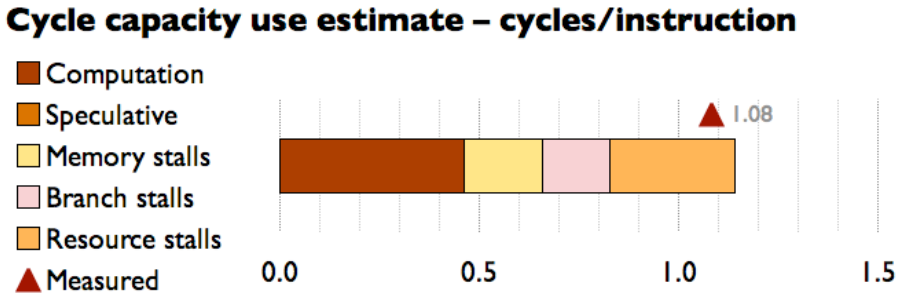




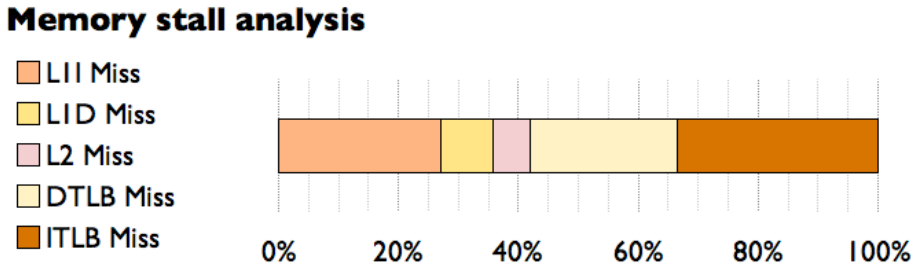
…but when it does work, the entire pipeline is filled up for a short while

Could we extract three times the performance from *existing hardware?*

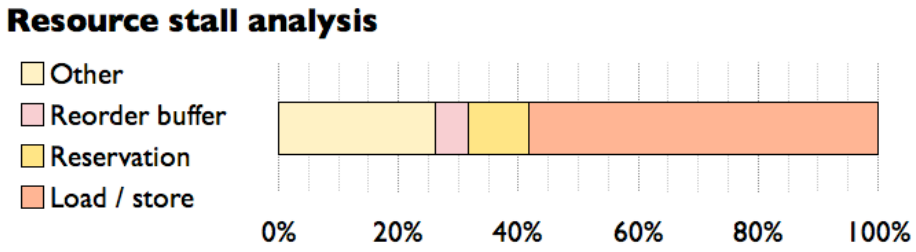
What *really* happens?



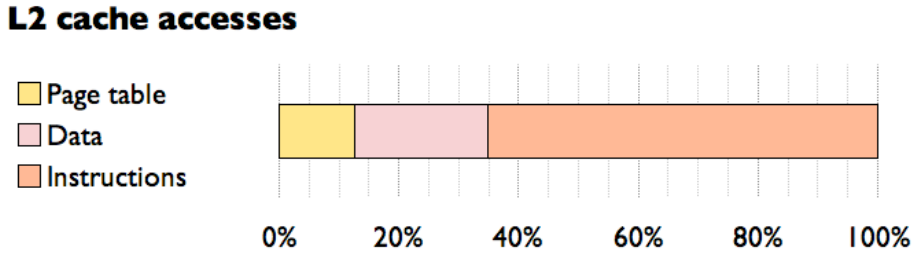
On AMD Opteron 270, a "wide" and resource-rich CPU, our apps are by far not compute bound. Analysis of stalls follows.



Surprise! Memory stalls are for *instructions* (60%) and *page table accesses* (60%). Number of data pages aside, data accesses appear not to be problem or are masked by other issues.



Resource stalls indicate a "wider" CPU with more resources could run the code faster. Except our stalls are for memory. Oops.



Another surprise! Instructions dominate the number of Level-2 cache accesses. We have found *a process of 500 MB can include as much as 150 MB of code pages* and several *algorithms have too large code working sets.*

Two other major "core" issues, I/O performance and per-job overheads, are still being analysed in further detail. We expect to report at a later occasion on both, as well as on our analysis on code size, compiler selection and impact of compiler options.

Other recent performance improvements

–Reduction in framework overheads; AOD analysis target rate is 2kHz.

–Use of shower profile files optimised in simulation: simulation time reduced to 25% in affected detector regions. In addition improved physics settings and benefits from Geant4 optimisations in simulation. Ongoing effort to optimise use of the magnetic field.

–Improved performance for conditions data access.

We use three different tools for performance analysis, depending on the desired precision of results.

> At the coarse end we simply time algorithm execution using CMS *framework timer facilities*. This is easily understood and done by every physicist. Most of the time we use *IgProf* as it is fast and provides a suitable level of detail. For maximally detailed code scrutiny we use *callgrind* from the valgrind family of tools.

For memory profiling the only tools practically accessible to us are *IgProf* and the valgrind family. We usually use IgProf. Its memory profiler has been instrumental in our analyses so far.

Most of our system level analysis was done using *perfctr*. We have also used *OProfile*, which is more widely installed, certainly very usable for basic analysis, but lacks several critical features for the level of detail we gathered. We are also learning to know *pfmon*.

IgProf is a profiler tool developed by L. Tuura and G. Eulisse for measuring and analysing application memory and performance characteristics. It requires no changes to the application or the build process, and no special privileges to run.

Few profilers are capable of correctly profiling CMS' C++ software. IgProf is a 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 analysis stage.

The main strengths of IgProf are its speed and efficiency.

The *statistical performance profiler* adds ~40 MB to the memory usage and negligibly (≤1%) to the run time. The *accurate memory profiler* adds 50–75% to the run time and ~250 MB to the memory use for a typical CMS task loading ~400 shared libraries, running for an hour, using ~500 MB memory and making ~1M memory allocation calls per second. IgProf is typically 10–100 times faster than valgrind or callgrind.

IgProf fills a dire gap between valgrind and system level profilers.

In addition we have tools for profiling the size of persistent data. We developed a *PerfReport tool suite* for easy generation of digestible performance reports from IgProf and callgrind.

We *extended callgrind to handle full call-stacks*, not just the gprof-style call graphs. The functionality was taken from IgProf.

For each release we run a *release validation suite*. We will soon begin to generate a standard performance report for each release, including canonical performance numbers for a "standard candle" sample analysed with a well-defined reference process.

Performance report summary page

producer	input	eventsetup	remainder		TOTAL
			(total)	(thereof mem mgmt)	
cms::CkfTrackCandidateMaker	0.022	0.011	119.713	23.701	119.746
SeedGeneratorFromRegionHitsEDProducer	0.022	1.870	16.398	3.222	18.290
HcalSimpleReconstructor	10.734	0.088	2.024	0.011	12.846
MuonIdProducer	0.242	0.044	10.624	2.013	10.910
SoftElectronProducer	0.132	0.088	10.294	1.474	10.514
cms::SiStripClusterizer	5.246	0.000	2.387	1.078	7.633
TrackProducer	0.022	0.011	4.905	1.001	4.938
cms::SiStripRecHitConverter	0.011	0.000	4.718	1.496	4.729
EcalWeightUncalibRecHitProducer	3.805	0.165	0.506	0.033	4.476
ElectronPixelSeedProducer	0.011	0.000	2.804	0.715	2.815
cms::BaseJetProducer	0.011	0.011	1.947	0.319	1.969
PixelMatchGsfElectronProducer	0.011	0.000	1.595	0.011	1.606
CSCRecHit2DProducer	0.451	0.000	0.935	0.165	1.386
cms::SiPixelClusterProducer	0.407	0.011	0.418	0.220	0.836
ESRecHitProducer	0.605	0.000	0.143	0.000	0.748
DTRecHitProducer	0.539	0.000	0.044	0.000	0.583

Performance report release comparison

Top 20 libraries

		self cost [seconds]		inclusive cost [seconds]	#times called	name	
	−	164.34	(28.350 %)	196.97	14'943	libc.so.6 [more]	1
		(-51.208 %)	(-2.555 %)	(-99.467 %)	(-51.208 %)	The item appears not to have moved.	
		62.56	(10.792 %)	216.81	5'689	libCint.so [more]	2
		58.14	(10.029 %)	269.71	5'287	libCore.so [more]	3
		32.98	(5.689 %)	39.62	2'999	libDetectorDescriptionCore.so [more]	4
		27.04	(4.665 %)	116.96	2'459	libstdc++.so.6 [more]	5
		26.39	(4.552 %)	30.75	2'400	libz.so.1 [more]	6
		23.61	(4.073 %)	48.61	2'147	ld-linux.so.2 [more]	7
↳	−	17.89	(3.086 %)	17.89	1'627	libm.so.6 [more]	8
		(-67.390 %)	(-1.947 %)	(-99.674 %)	(-67.388 %)	This item has moved down the list by 4 ranks. Former absolute position: 2.	
		11.32	(1.953 %)	133.62	1'030	libCintex.so [more]	9
		11.08	(1.911 %)	29.99	1'008	libTrackingToolsTrajectoryState.so [more]	10
		8.47	(1.461 %)	55.39	771	libRecoTrackerTkDetLayers.so [more]	11
↱	+	7.66	(1.321 %)	17.89	697	libTrackingToolsKalmanUpdators.so [more]	12
		(+131.420 %)	(+1.018 %)	(-99.799 %)	(+131.561 %)	This element was not in the top 20 before. Former absolute position: 35.	
	−	6.79	(1.171 %)	38.81	618	libTrackingToolsMaterialEffects.so [more]	13
		(-56.222 %)	(-0.252 %)	(-99.812 %)	(-56.201 %)	The item appears not to have moved.	
		6.07	(1.047 %)	7.50	552	libMagneticFieldInterpolation.so [more]	14
↳	−	5.54	(0.956 %)	12.79	504	libTrackingToolsAnalyticalJacobians.so [more]	15
		(-60.173 %)	(-0.321 %)	(-99.785 %)	(-60.158 %)	This item has moved down the list by 2 ranks. Former absolute position: 15.	
		4.77	(0.823 %)	7.15	434	libGeometryTrackerGeometryBuilder.so [more]	16
↱	+	4.75	(0.819 %)	8.99	432	libGeometryCaloGeometry.so [more]	17
		(+202.548 %)	(+0.675 %)	(-96.695 %)	(+202.098 %)	This element was not in the top 20 before. Former absolute position: 51.	
↱	+	4.70	(0.811 %)	5.18	428	libDataFormatsMath.so [more]	18
		(+46900.000 %)	(+0.810 %)	(-87.953 %)	(+42700.000 %)	This element was not in the top 20 before. Former absolute position: 179.	
↱	+	4.52	(0.780 %)	9.05	411	libTrackPropagationSteppingHelixPropagator.so [more]	19
		n/a	n/a	(-9.500 %)	n/a	This element was not in the top 20 before. Former absolute position: 233.	
		4.50	(0.776 %)	186.48	410	libTree.so [more]	20
		78.63	(13.564 %)	n/a	n/a	(others)	21

Total cost in this list (formerly): 1089.87
Total cost in this list: 579.69 (-46.811 %)

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.

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.

The bottlenecks in your software may now be easier to find!

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.