Software performance optimization

Peter Elmer
Princeton University
Workshop CCR INFN-Grid, La Biodola
17 May, 2011

Things I'll talk about

- In this presentation I'll talk about some technical aspects of software performance optimization
- I work on CMS and thus this is (mostly) about our experience and plans in CMS
- I'll be talking about:
  - Compiler (and OS) version evolution
  - The transition to 64bit
  - Multicore and “whole node” job scheduling
- A number of people in CMS have contributed to this work: Giulio Eulisse, Vincenzo Innocente, Chris Jones, Shahzad Muzaffar, Lassi Tuura + many CMS software developers

Things I won't talk about

- I won't be talking about the I/O part of the performance optimization
  - see the talk by Giacinto
- I won't talk here about "Physics Choice" software optimization
  - I note however that the combination of experience with data taking (at LHC) and actually hitting resource limits in 2011 (unlike 2010) may lead to more of this type of optimization this year and next (offset however by pile-up!)
- I won't talk much about "routine" performance optimization we are doing with the software (often memory mgmt issues, poorly implemented or redundant algorithms, etc.) or the tools we use to do this: see older talks by me or the other CMS people, google "igprof sourceforge", etc. (or ask me later)

Transition to gcc4/SLC5

- (Going backwards in time a bit for some context!) Approximately two years ago the LHC experiments (including CMS) went through a transition from SLC4 to SLC5 and from gcc34 to gcc43.
- The computing centers transitioned to SLC5 on WN's, and specifically to SLC5/64bit for the OS
- CMS transitioned from "slc4_ia32_gcc345" builds to "slc5_ia32_gcc434" (and we could run jobs of both types on the SLC5/64bit WN's)
  - Over time people transitioned to using the newer software releases and slc5_ia32_gcc434
- The gcc3.x to gcc4.x transition was a large one in terms of internal evolution: in some sense it represents the transition from a "cathedral model" to a "bazaar" model
  - We chose gcc43 rather than gcc41 (the RHEL5 system compiler) as it was the "current" compiler version at that time. there were o(8-10%) performance improvements in typical applications.

Transition gcc4/SLC5

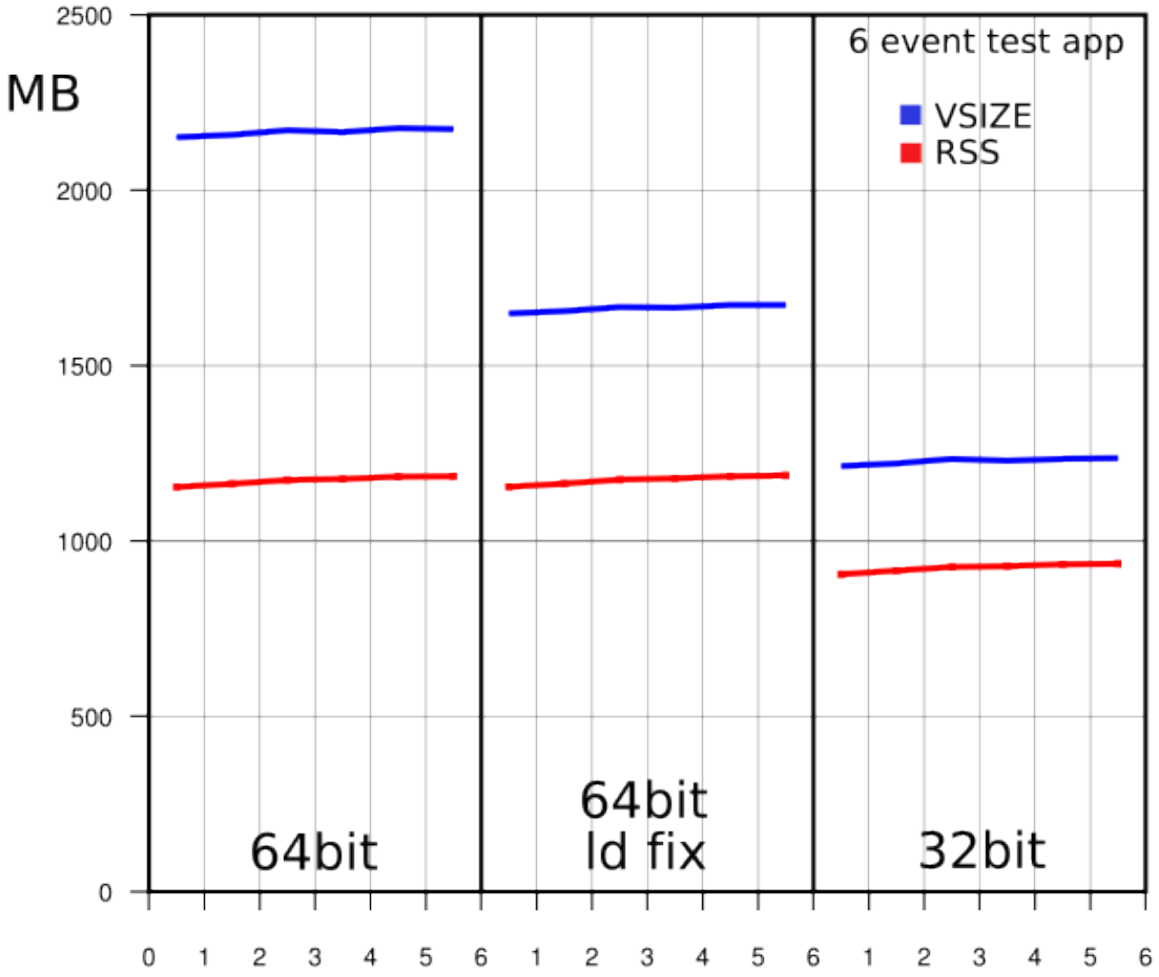
- The transition model was in which the WN's move forward (in this case to SLC5/64bit) and the experiments (including CMS) “catch up” by transitioning the jobs we run on those WN's over time from slc4_ia32_gcc345 to slc5_ia32_gcc434.
- As “CMS moves” from the old architecture to the new one we exploit better the hardware
  - “Production” activities (MC production, data reconstruction) move faster than “analysis” activities
- Obviously the missing pieces here are:
  - Transitioning to native 64bit builds
  - Transitioning to newer compiler versions and exploiting newer features of the compiler and C++ (e.g. gcc 4.6.0 was released 6 weeks ago)
  - An important constraint we have is that we want the CMS High Level Trigger (online) to use the same architecture as offline “production” activities, this constrains us to new build architecture changes during shutdowns (once/year)

64bit (AKA x86-64)

- Advantages:
  - Better architecture: additional registers, better calling convention, SSE FP math support, reduced -fPIC cost
  - CMS sees typical CPU performance improvements from 18% (Geant4) to 30% (reconstruction, HLT)
- Disadvantages:
  - Somewhat more memory hungry, typically 25-30% in CMS (see next slides for additional notes)
  - Some coding assumptions are no longer valid (e.g. data sizes) thus the transition requires a bit more care
  - SSE math (and IEEE754 compliance) results in somewhat slower transcendental math functions from linux libm (software implementation)

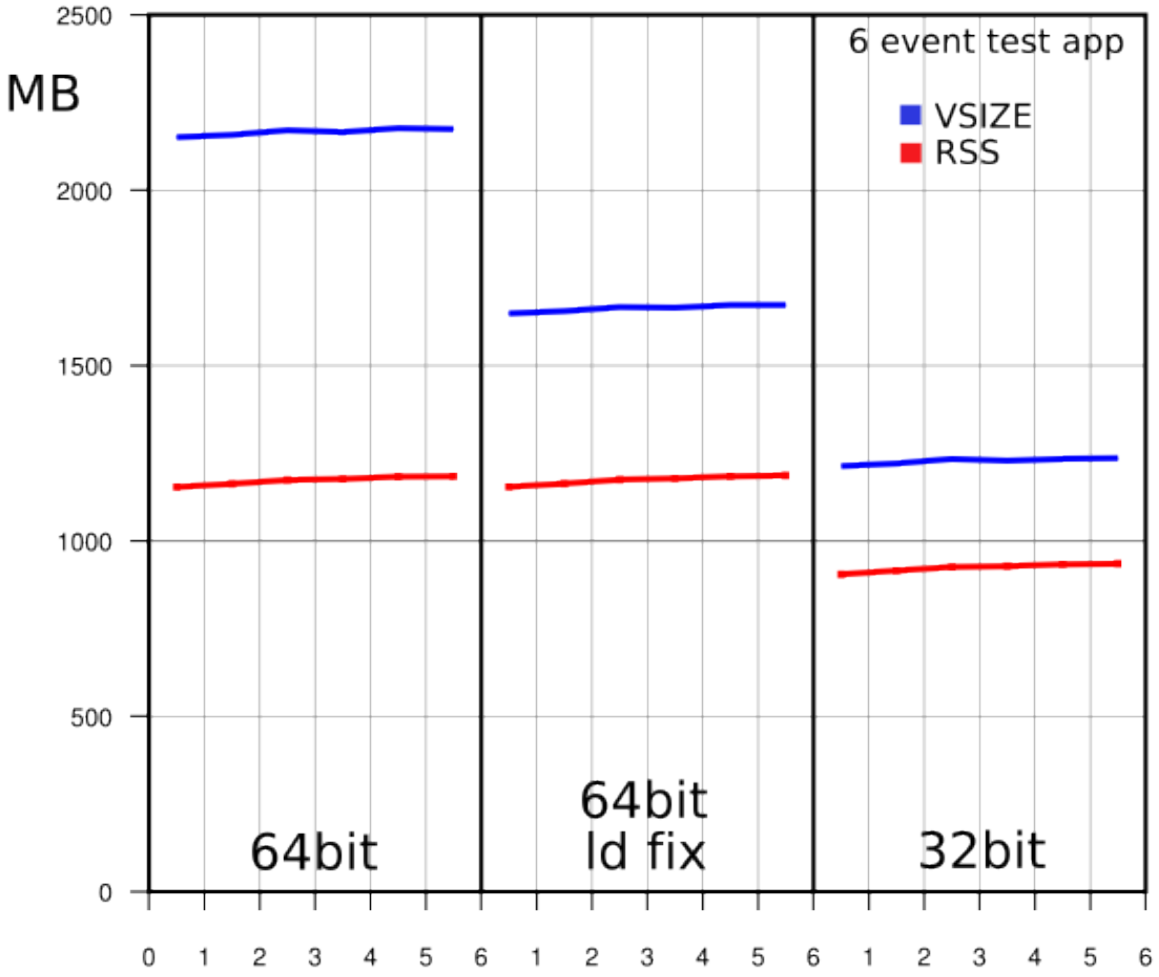
Memory footprint myth-busting

- An early observation (much quoted) was that the VSIZE doubles for the 64bit applications, while the RSS increase is much more modest (25-30%) But where does any of this increase come from?
- The actual code (text/data) size itself only increases by a small amount (~5%) from 32bit to 64bit
- Some data increases come from the doubled pointer size and alignment padding of data structures, but these don't explain the VSIZE.



Memory footprint myth-busting

- It turns out there are two effects:
- First, recent versions of the binutils linker will by default align code pages to 2MB. Thus there is extra "padding" in the virtual memory space. We have 500+ shared libraries, so this is a big effect.
- Second, since at 64bit the virtual memory space is huge, many things (e.g. glibc when reading the locale file) now just memory map files into the memory space and let the OS do the work of paging things in/when needed.

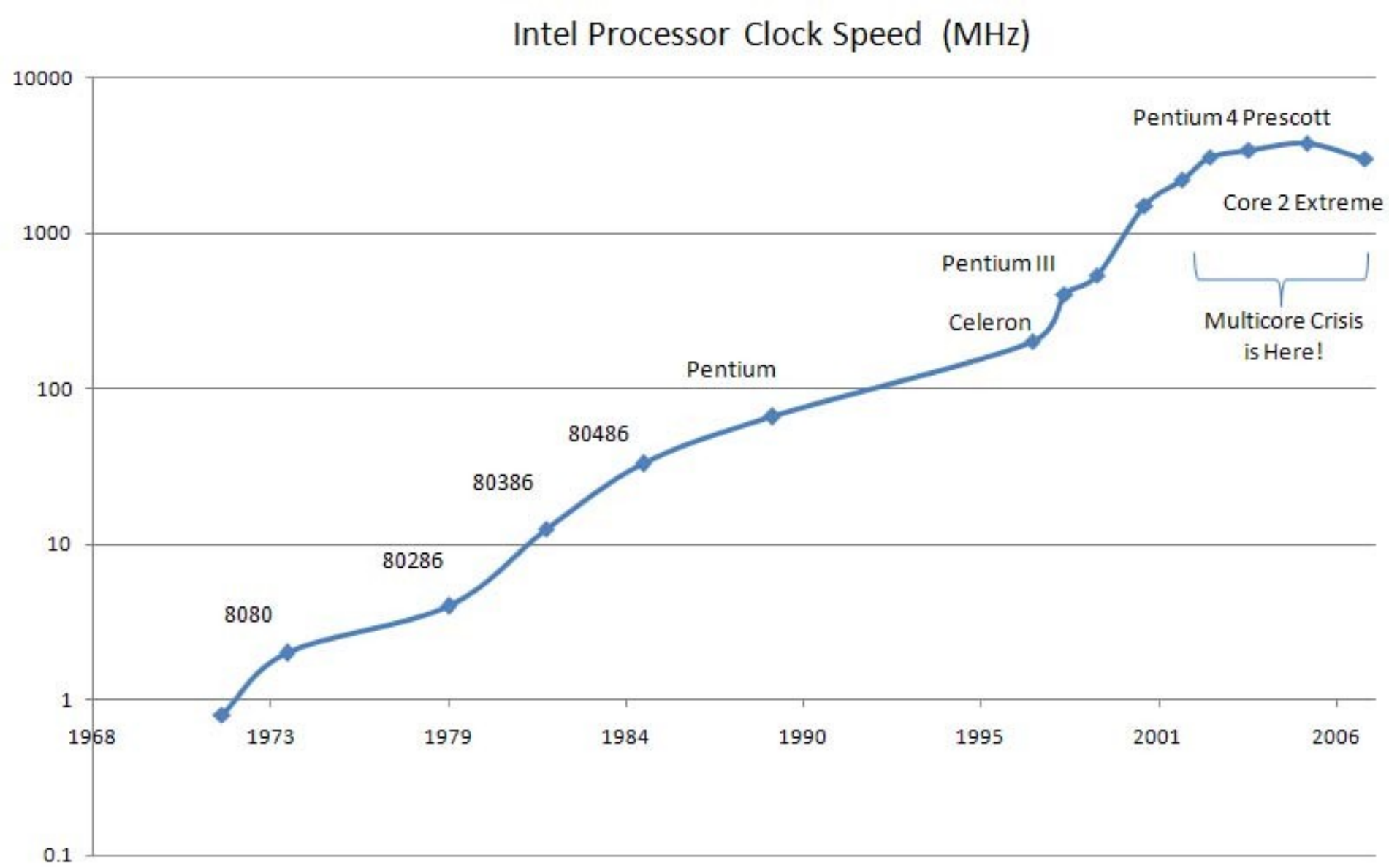


VSIZE is a really poor estimator for the physical memory needs or usage of a process, especially for 64bit processes.

CMS and 64bits

- As of Jan2011 the CMS has transitioned to building only native 64bit software in the current releases and this is deployed:
  - Online (High Level Trigger) at CERN
  - Offline reconstruction, simulation and analysis (everywhere)
  - Computing components and websites (mostly at CERN)
- The rapid increase in data during 2011 is also helping to pull people doing analysis forward onto the newer (64bit-only) software releases
- (Victory! What next?)

CPU clock speeds/Multicore



Multicore CPU's – "Phase 0"

- The current model for HEP applications to exploit multicore CPU's is very simplistic: we exploit the "job" (and event) level parallelism and (typically) launch one application process per core
- The local schedulers matched this by configurations to schedule independently (and incoherently) an individual "job" per core
- Hyperthreading and pilot jobs are only minor caveats to the above. The "quantum" of work has become something of order "one batch slot/core".
- This strategy "works" in that we are able to exploit multicore CPU's reasonably efficiently on the kinds of multicore machines deployed today, but there are many consequences (next slide)

Multicore CPU's – "Phase 0" Consequences

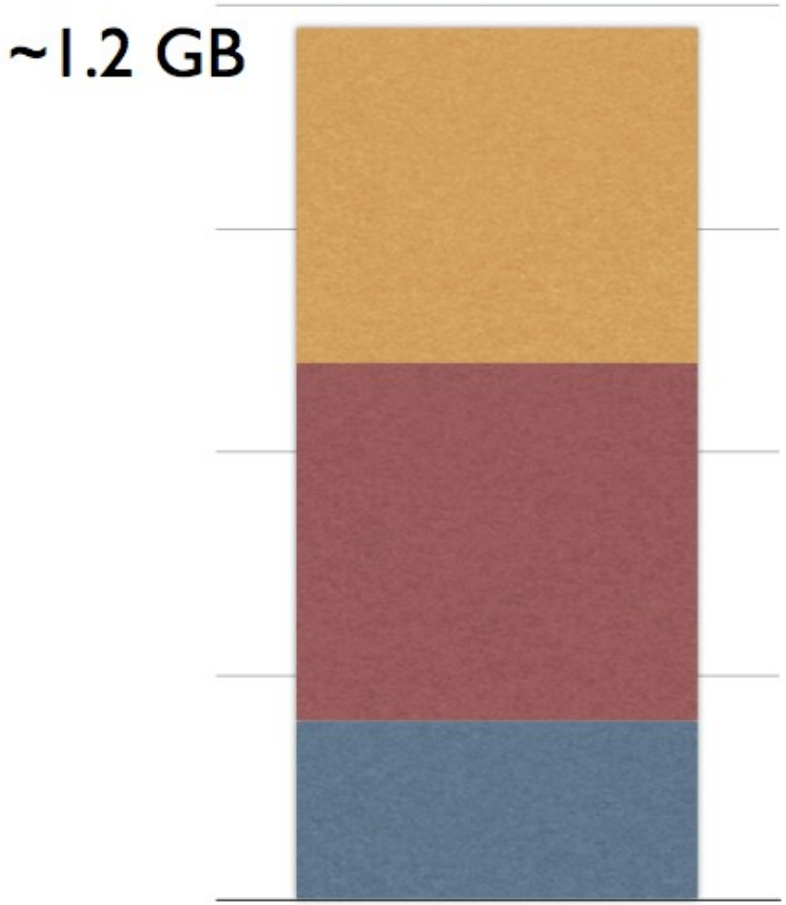
- The simple "Phase 0" job-level parallelism has consequences:
  - The memory needs increase with each generation of CPU
  - The number of independent readers and writers (to local disk, to remote storage) increases with each generation of CPU
  - An ever increasing numbers of independent and possibly incoherent jobs running on any given piece of physical hardware.
  - Each of these running "jobs" commands an ever tinier slice of resources and do not explicitly share resources they could share
- And on top of that, there are a number of reasons to believe that this trivial parallelism won't scale efficiently on the CPU's themselves.
- Note that we have been going down the "~one job/core" route simply because we couldn't do anything else, it isn't a "natural" way to use these resources for high throughput.

Can we do better?

- For sure.
- We cannot, however, easily jump to a properly and perfectly parallelized application that directly supports exploitation of multiple cores.
- We have been doing investigations into how to do that, but it would be a large undertaking requiring significant code changes.
- There is however an intermediate solution ("Phase 1") in which we can do better than the current situation.

Typical HEP memory footprint

CMS offline software memory budget



Event specific data

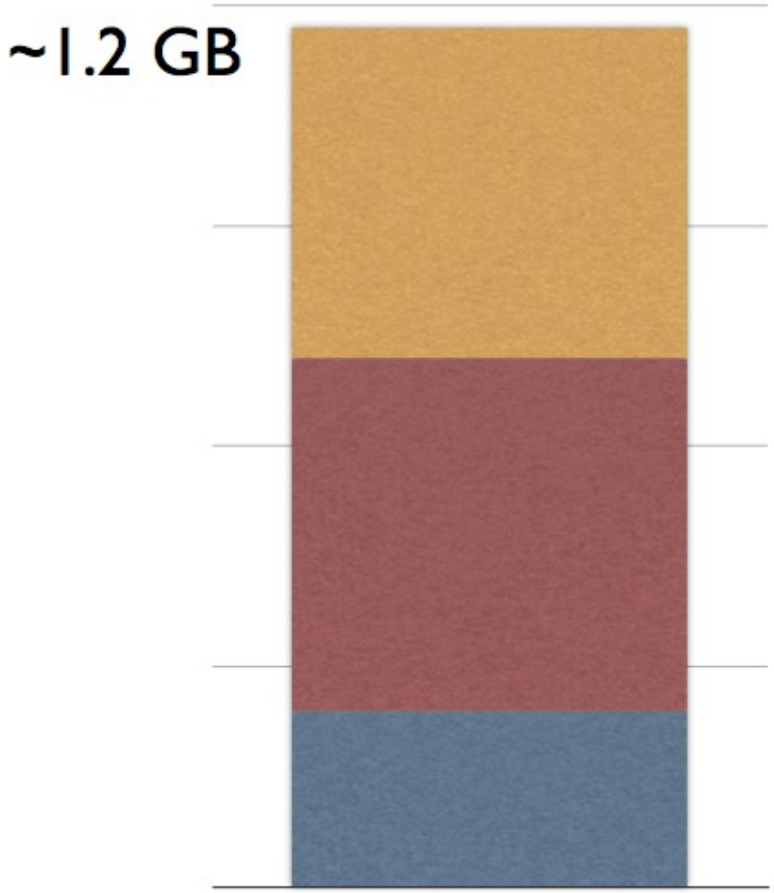
Read only data

geometry,
magnetic field,
conditions and alignment,
physics processes, etc

Code

Typical HEP memory footprint

CMS offline software memory budget



Event specific data

Read only data

geometry,
magnetic field,
conditions and alignment,
physics processes, etc

Code

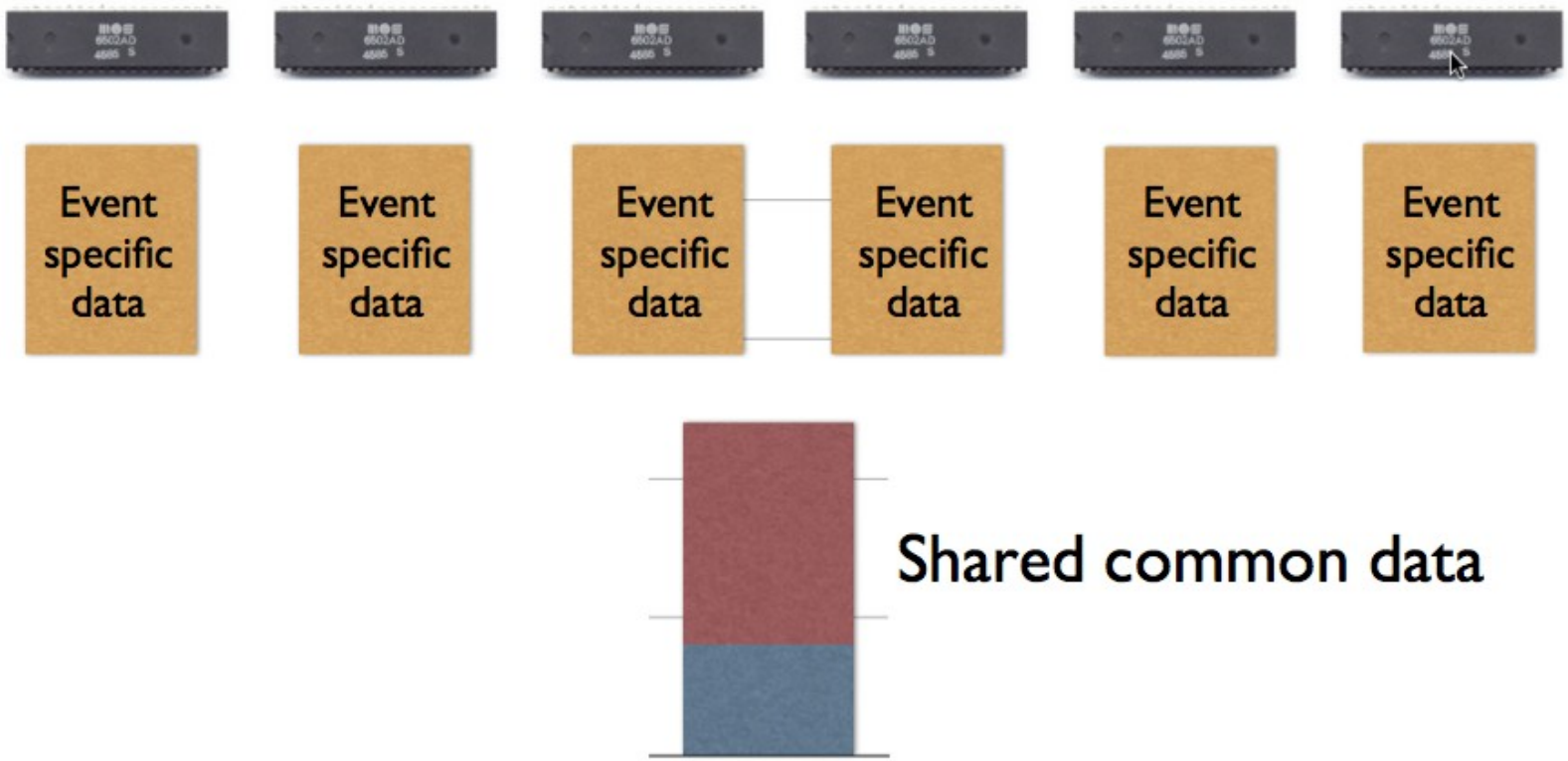
COMMON!

Exploiting Copy on Write

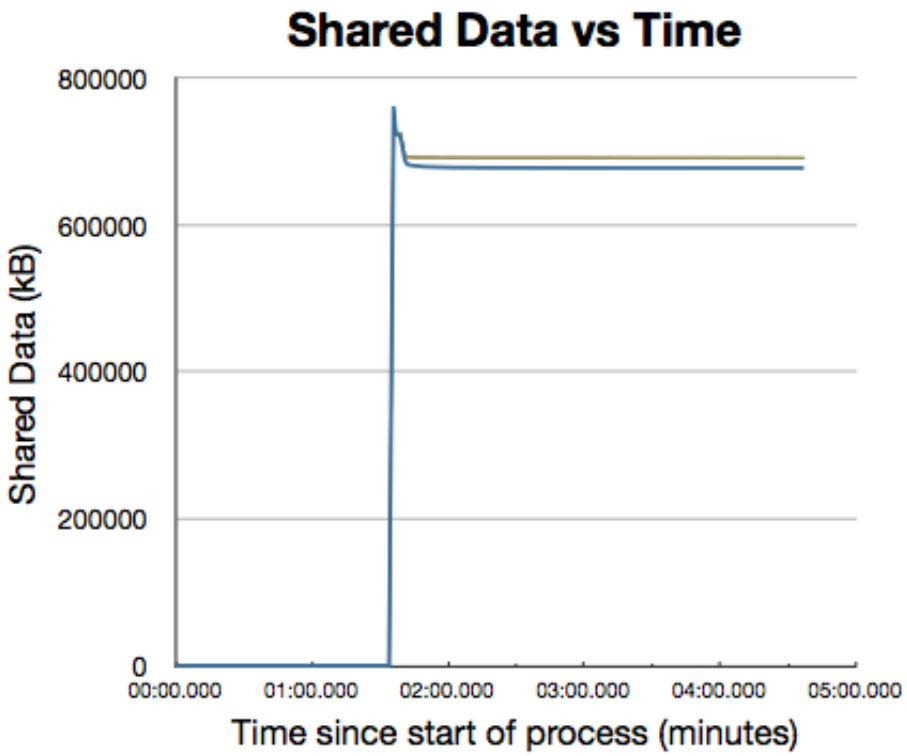
- Much of the common constants and data can actually be loaded into our applications very early
- If you fork a new process at that point the kernel is actually smart enough to share common data memory pages between the parent and children
- The kernel "un-shares" the memory pages only when one of the processes writes to them
- Subsequent memory allocations (e.g. "event" data) in either process wind up on non-shared pages

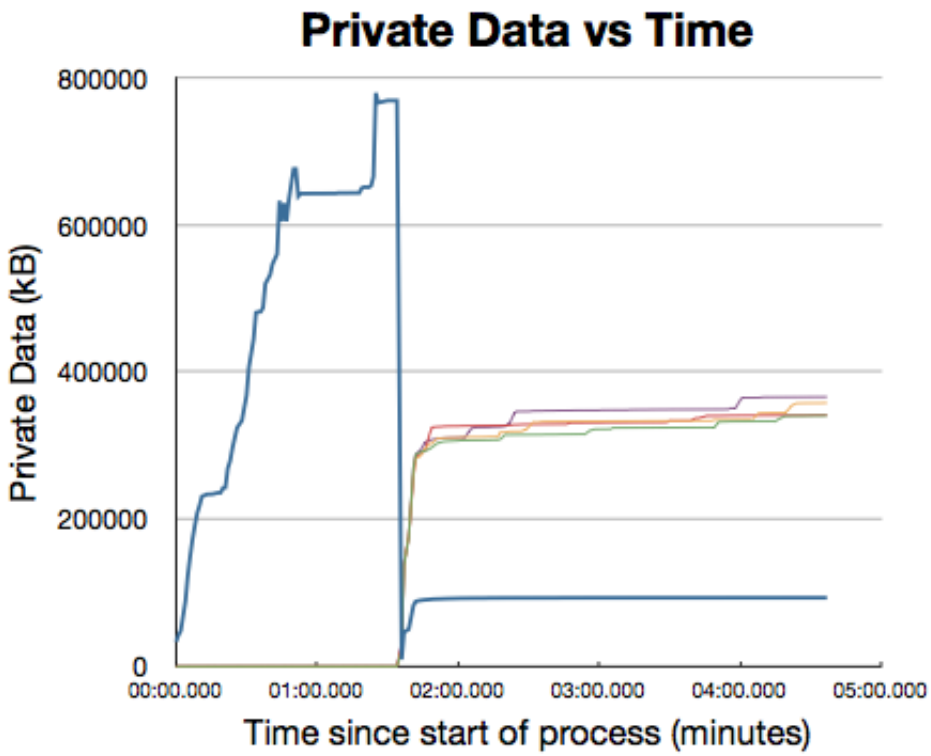
“Multi-processing” applications

CMS near future multicore strategy:
forking



Forking: memory sharing

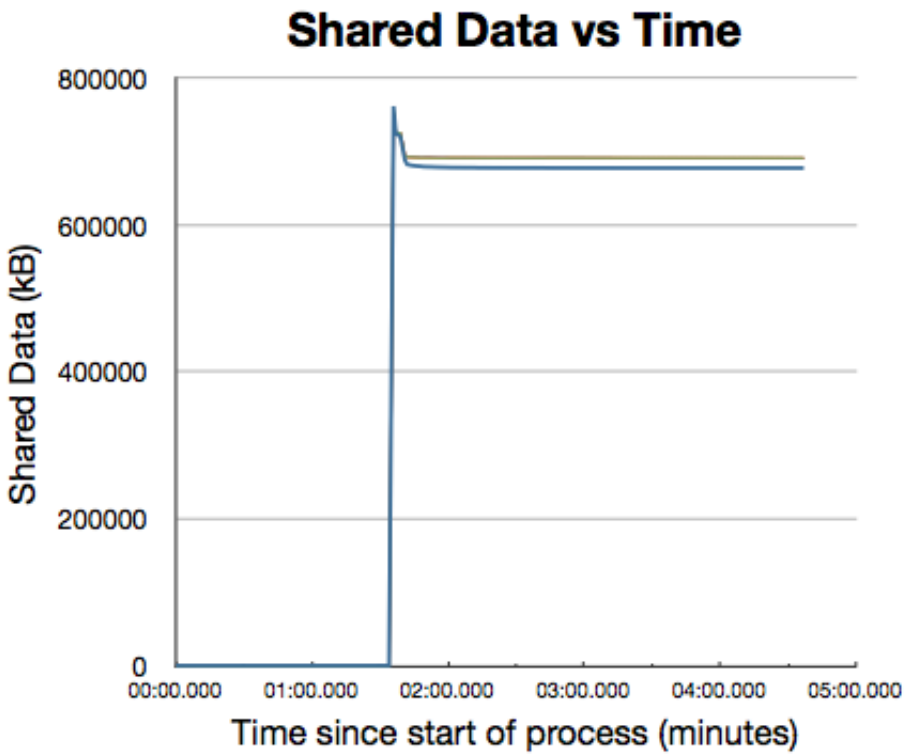


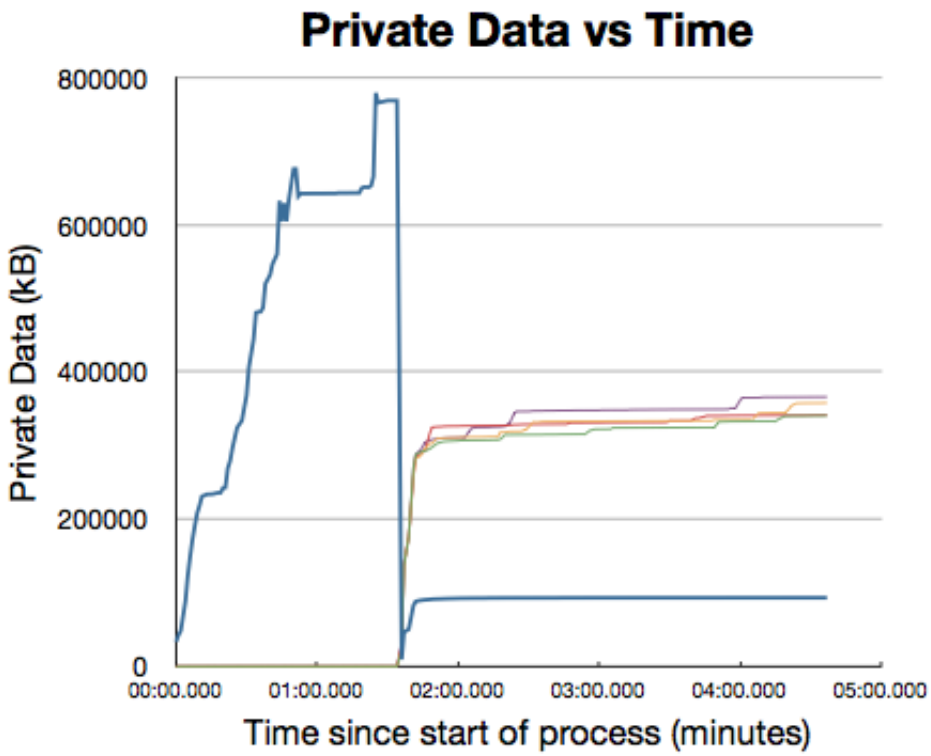


Measurements done using reconstruction with 64bit software on 4 CPU, 8 core/CPU 2GHz AMD Opteron(tm) Processor 6128

Shared memory per child: ~700MB
Private memory per child: ~375MB
Total memory used by 32 children: 13GB
Total memory used by 32 separate jobs: 34 GB

Forking: memory sharing



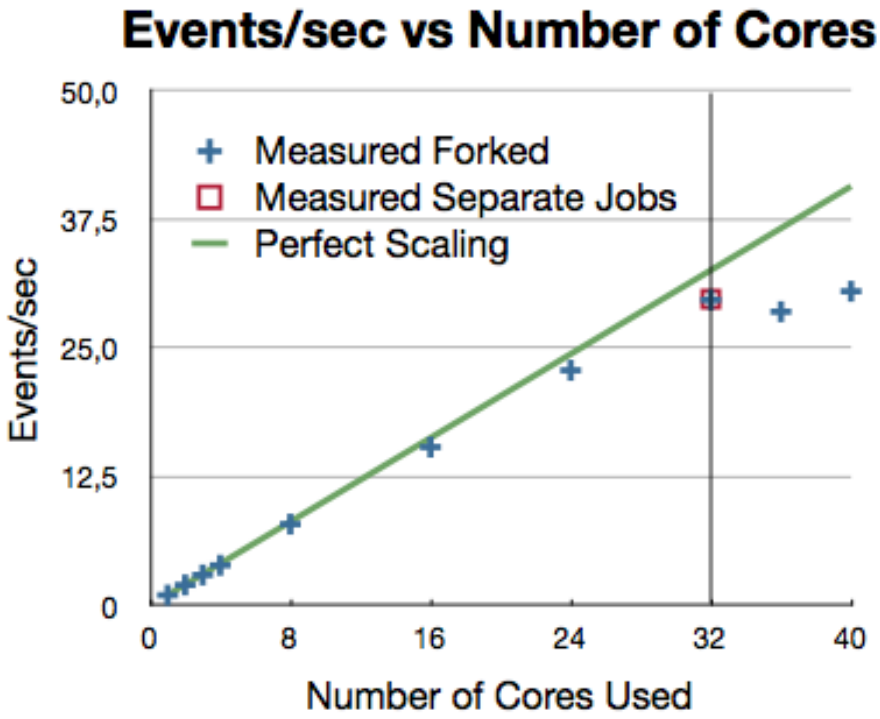


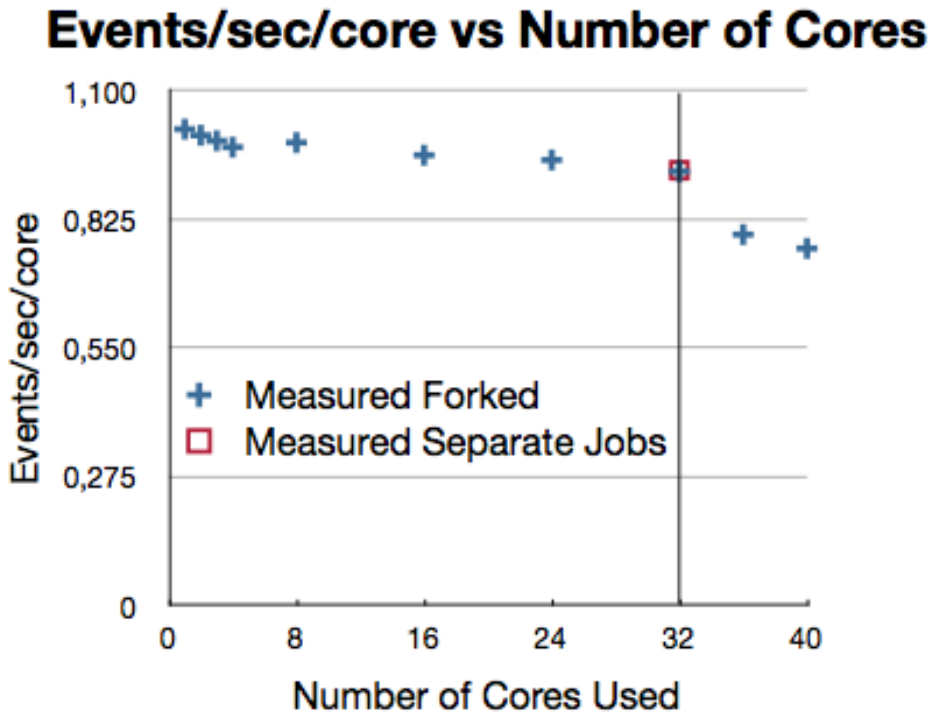
Measurements done using reconstruction with 64bit software on 4 CPU, 8 core/CPU 2GHz AMD Opteron(tm) Processor 6128

Shared memory per child: ~700MB
Private memory per child: ~375MB
Total memory used by 32 children: 13GB
Total memory used by 32 separate jobs: 34 GB

Greatly reduced physical memory needs

Forking: throughput





Another memory accounting issue

VSIZE is NEVER a good way of accounting *for actual memory usage. In particular on 64bit*

RSS is only slightly better*. It works in the case of a single process, but still* ***does not actually account for sharing of resources*** *in forking jobs*

Multi-core(-aware) applications require a global understanding of the physical to logical memory mapping. ***CMS requested using PSS to account memory use***

More resources:

http://www.selenic.com/smem/

"ELC: How much memory are applications really using?" (http://lwn.net/Articles/230975/)

"Whole Node" scheduling

- (Obviously) the one natural unit in the system is the "whole node", where that is defined as: the physical thing running one (unvirtualized) copy of the OS and sharing a set of resources (CPU, disk, network, etc.)
  Using "Whole node" scheduling as the resource "quantum" has obvious benefits:
  - The applications *explicitly* take over the management of the sharing of resources within the "whole node" quantum of resources.
  - It is compatible with the current "phase 0" applications as well as more explicitly multicore-aware applications (via forking, threading, etc.) optimized over time to maximize throughput
  - It also makes a number of data/workflow management optimizations possible: I/O caching, local merging, etc.
  - Sites only need to defend the whole node, not individual processes.
  - A move to a proper "whole node" accounting for CPU/memory use, etc. recognizes the role of the OS in optimizing access to resources

CMS Multicore Strategy

- Phase 0 – independent jobs on each core (+ hyperthreading?)
- Phase 1 – development and deployment of CMS framework and WM system to fork sub-processes after loading bulk conditions

This year

  - Advantages: reduce memory needs and more flexibility for data/workflow mgmt (with only limited changes to software)
  - First steps with sites, grid providers, etc. to multicore

- Phase 2 – deployment of more fine grained parallelism

Next Year And beyond?

  - More difficult, requires greater changes to our software
  - Impacts software development model, may require more sophisticated software development in some cases

How does "whole node" affect Data Mgmt?

- Changes the system working point by an "order of magnitude": the number of schedulable running "entities" within the system will drop by a factor of 4-8 (today), more later, and will become approximately constant in each site going forward
- The resources (local disk, memory, etc.) managed by a (pilot) job start to become significant, providing many opportunities for optimization:
  - Stage-in to and mgmt of local disk, suddenly we are "memory rich"
  - coordinated/coherent I/O access across activity "node"
  - reduced external connections&streams
  - local output merging (or direct write of combined file)
  - "backfill" CPU intensive activities if necessary

Files sizes and transfer

- Hand in hand with a change to “whole node” scheduling it is important to make sure we can scale file sizes at the same time
- We removed the 2GB limit some years ago, but in practice the reliability of the system is such that we've not gone very much beyond that.
- The limitations come from the fact that any error during file transfers requires the transfer to restart from scratch. This is seen as the main bottleneck preventing us from increasing significantly the file size.
- CMS Request: we would like the appropriate changes to be made to FTS and/or the storage implementations to avoid restarting transfers from scratch on errors (when possible)

"Whole Node" Job Submission Task Force

- Atlas, LHCb and Alice are also interested in "whole node" job submission, not just CMS
- Atlas and LHCb (which share their application framework) also have the "forking" functionality to share memory
- Alice is interested in using PROOF-Lite
- Since all of the experiments expressed interest in this, the LCG-MB set up a "Whole Node Job Submission" task force (I am also the chair of that task force, however it is multi-experiment).
- *Mailing list: whole-node-task-force@cern.ch*

(or send me a mail at Peter.Elmer@cern.ch)

Whole-Node Task Force Mandate

- Follow the "commissioning" of multi-core jobs being carried by the experiments. Facilitate discussion between the experiments and the computing resource providers to understand the specific problems encountered, realistic requirements and ingredients for an eventual "whole-node" deployment.
- Understand what, when and by whom changes need to be made in the full job submission chain. The idea is to consider the complete chain starting from the job submission user interface tools, and finalizing to the concrete configuration of the batch systems at the Grid sites.
- Prepare a roadmap for the deployment of end-to-end whole-node job submission taking into account the real status of the experiment applications.

- Propose new resource accounting and monitoring for multi-core jobs. The traditional accounting schemas are probably no longer adequate when the experiment allocates the complete node and takes responsibility of make proper use of all the resources. New methods will need to be discussed and agreed.

(Reports to LCG-MB)

Other investigations

- (Returning from multicore discussion to compiler discussions)
- In CMS we expect that we will be able to make another compiler version transition at the end of the year (again, this is limited for us by our desire to keep our online/trigger in synch with the offline)
- At the moment we are investigating a number of things: gcc46, c++0x, large (and/or reordered) shared libraries, vectorization, link-time-optimization, symbol visibility, etc.
- Ask again at the end of the year to see what, if anything, we were able to deploy for these things and what we learned.

Summary

- CMS has recently completed its transition to fully 64bit software, which brought a number of improvements.
- We (and the other LHC experiments) are interested in better exploitation of multicore machines. We have a first implementation (using forking) of an application which does this.
- The accounting of process memory needs (in particular using VSIZE and/or RSS) needs to be carefully rethought, due to both 64bit and multicore.
- Deployment of "whole node" job scheduling, and associated system accounting improvements, is a key first ingredient to deploying multicore-aware applications.

Pictures I have to show....

… because I swiped some slides for this talk from Giulio Eulisse (who in any case swiped things from my earlier talks, and so on).

CMS 64bit transition!

CMS multicore plans!

(Thanks to Giulio...)