CMS 64-bit transition & multi-core plans

Giulio Eulisse - FNAL hitman

taking credit for work done by

Peter Elmer, Vincenzo Innocente, Chris Jones, Lassi Tuura & many others

CMS 64-bit transition

x86-64

Pros:

- Better architecture

  additional / larger registers, better calling convention, reduced -fPIC cost, i.e. all in all from **15%** (G4) to **30%** (HLT, Reco) **faster**

Cons:

- Some "coding assumptions" are not valid anymore
- Memory hungry

  pointers take double the memory, by default linker aligns DSOs to MB page boundaries in 64bit mode

- CISC math no more

  x86-64 math unit lacks / has extremely different implementation of transcendental functions. libm falls back using more accurate (slower) software implementation to ensure IEEE compatibility

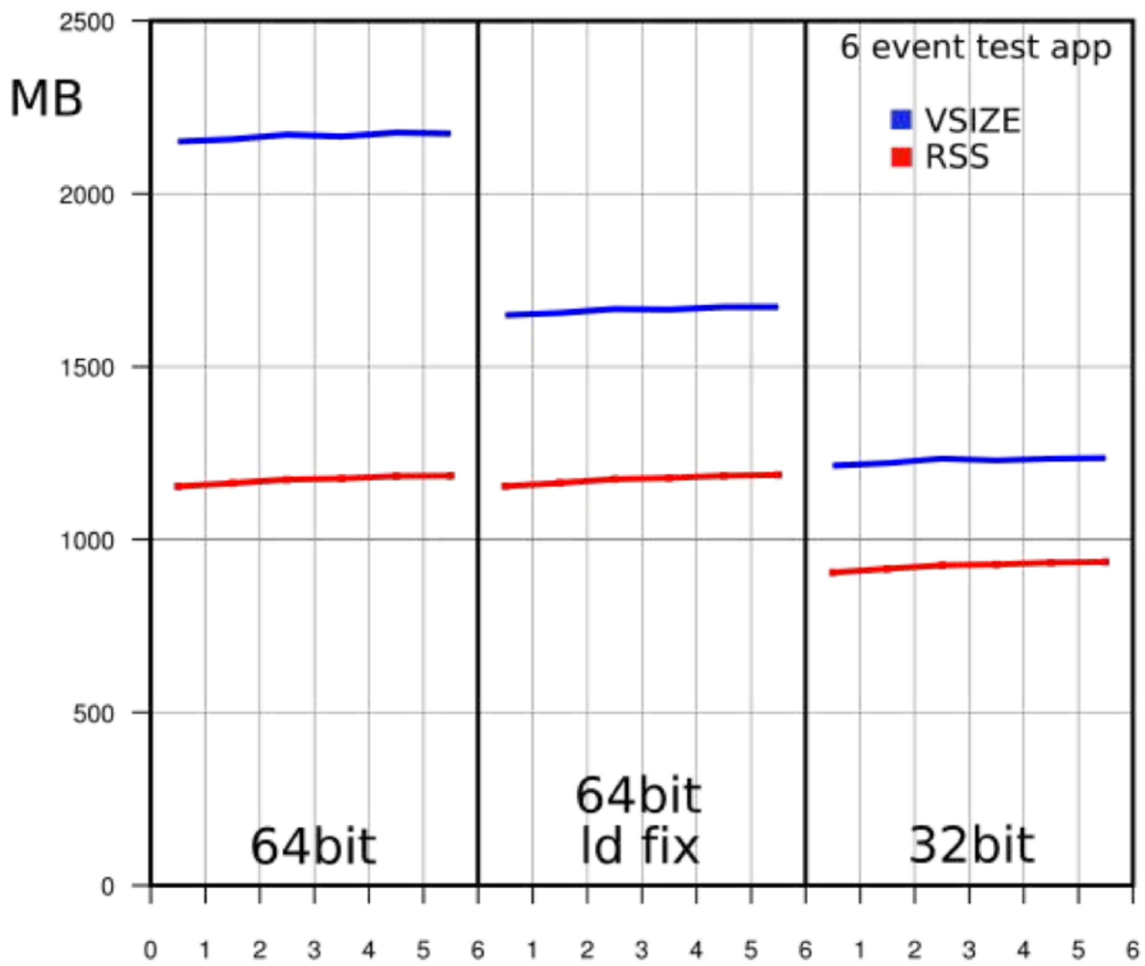
Memory footprint myth-busting

VSIZE* is in general a very poor metric for actual memory usage

Accounting should measure actual system memory use, not just address space allocation (VSIZE). Lots of modern programs written for 64-bit expect they can use address space liberally, and are smart about actual memory use

Most of the VSIZE increase comes from the fact the dynamic linker, by default, uses N-MB alignment for DSOs (libraries etc.). This is not actual memory usage - the gaps are unmapped. We are working around VSIZE-based accounting by using linker options to reduce gaps

VSIZE includes mmap-ed files which are actually read lazily from disk. It consumes memory only if paged in



* size of the process mapped address space

Memory footprint facts

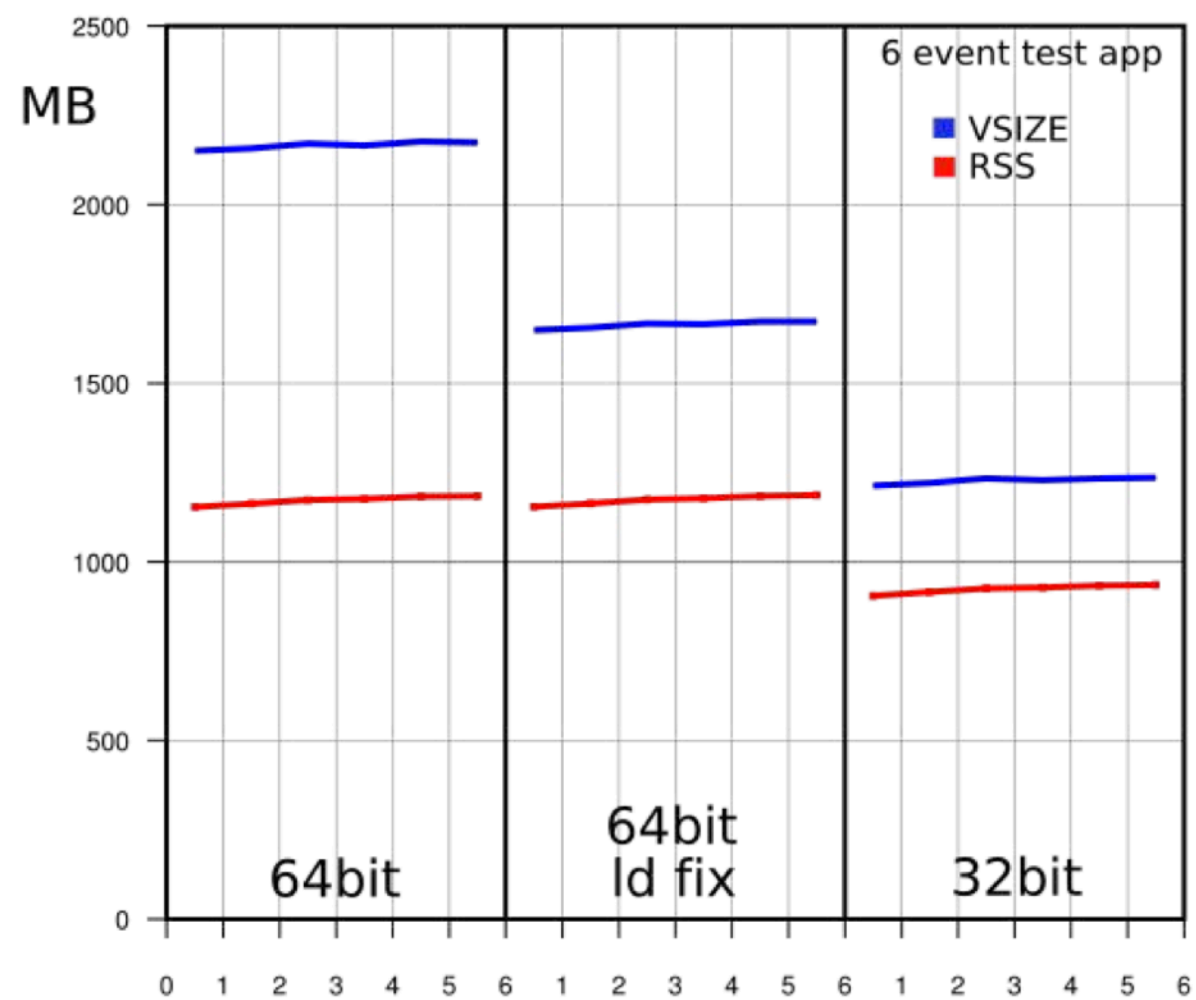
Nevertheless we see a 25-30% increase in RSS.

Padding and alignment overhead increases on 64-bit systems, especially with small field/object sizes

Pointers take double the amount of memory

People who don't know the difference between int and long (and use the latter) take double the amount of memory

Good news is that all the clean-ups we are already used to do for 32-bit now give a 2x gain



CMS and 64 bit

CMS ported its software stack to work **natively** on Linux / MacOSX x86-64

- Online high level trigger farm software
- Offline reconstruction and analysis
- Computing components and websites

Since 2011 we no longer build 32-bit software releases

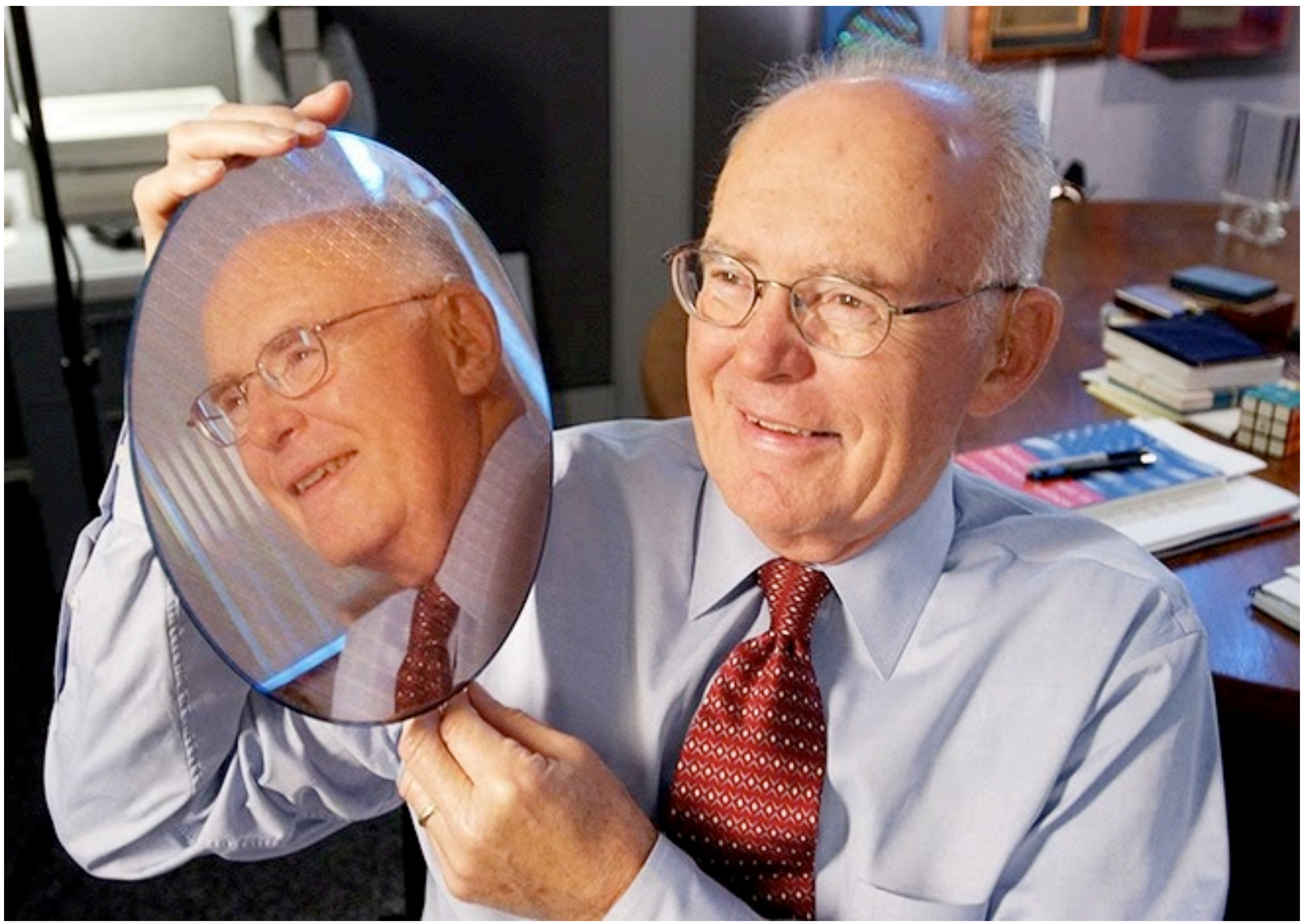
Mission accomplished

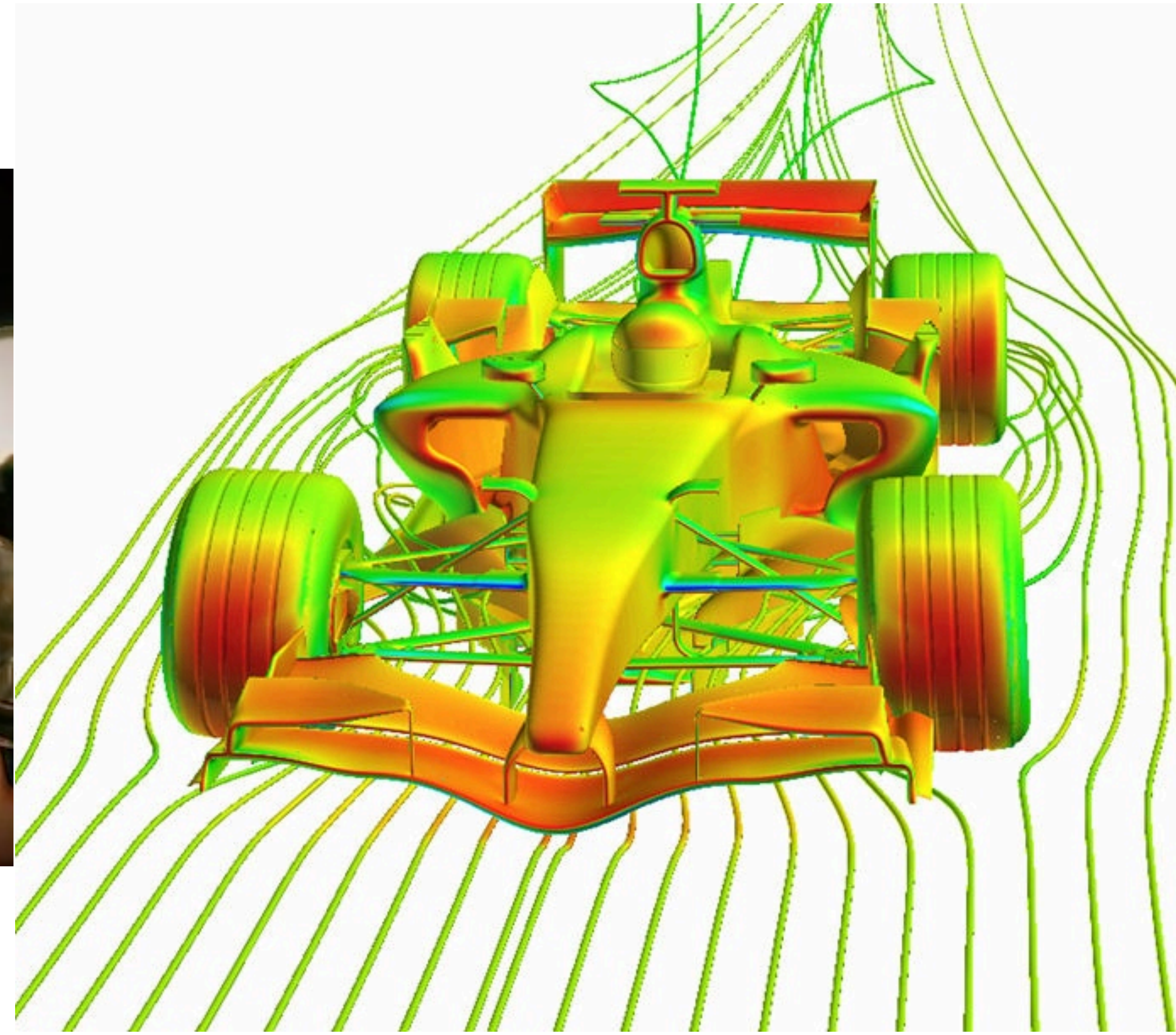
CMS multi-core plans

Multi-core

Mega-Hertz rush is over

Future is multi-core (until graphene will get in the loop)





future is already a few years old and apart from videogame developers and wind-tunnel guys everybody else still needs to figure it out

HEP present: single-core scheduling

HEP present: single-core scheduling

Bad idea:

The memory needs increase with each generation of CPU

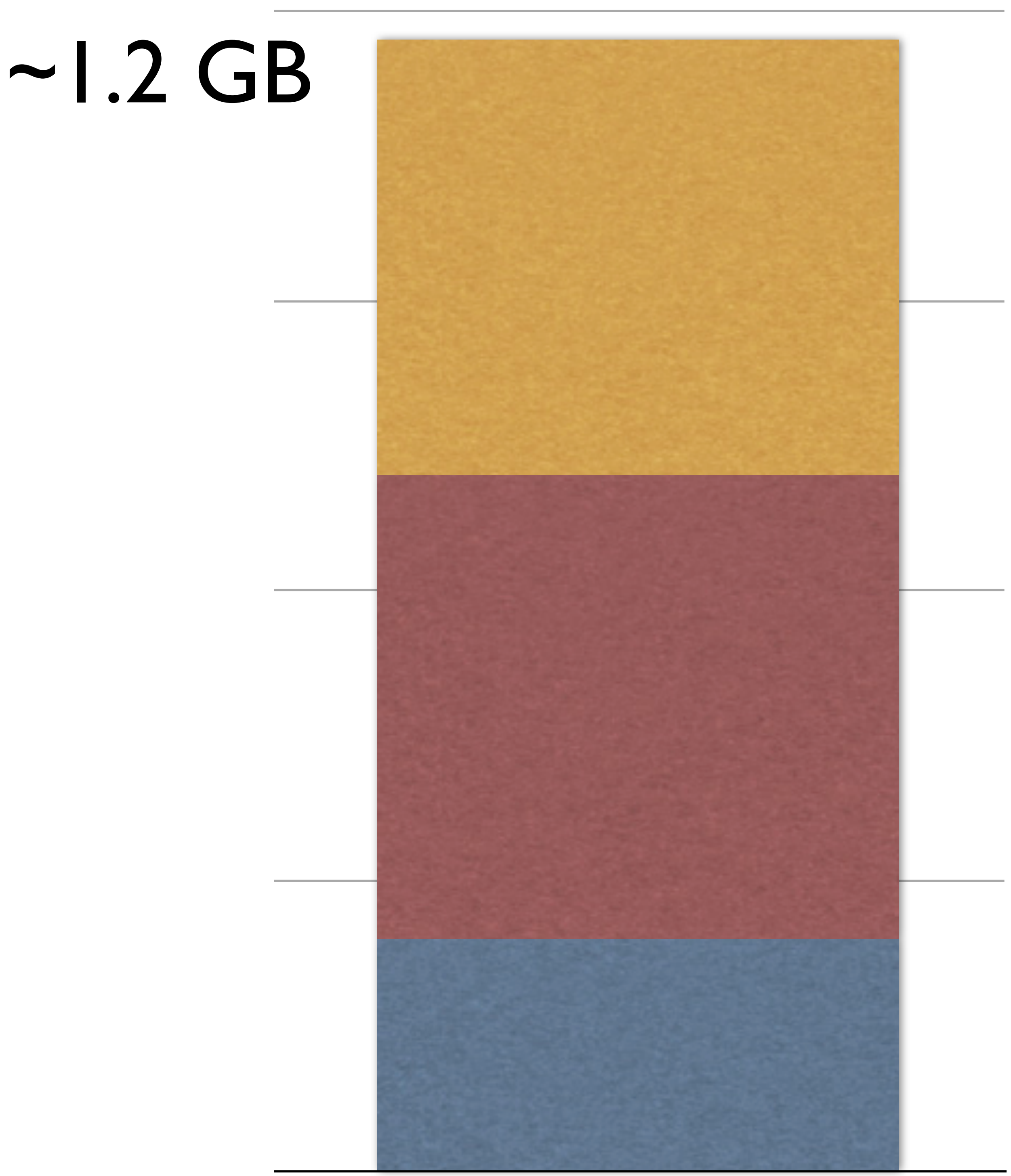
The number of independent readers and writers (to local disk, to remote storage) increases with each generation of CPU

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

At current rate we might end up not being able to afford 2GB per core

CMS offline software memory budget



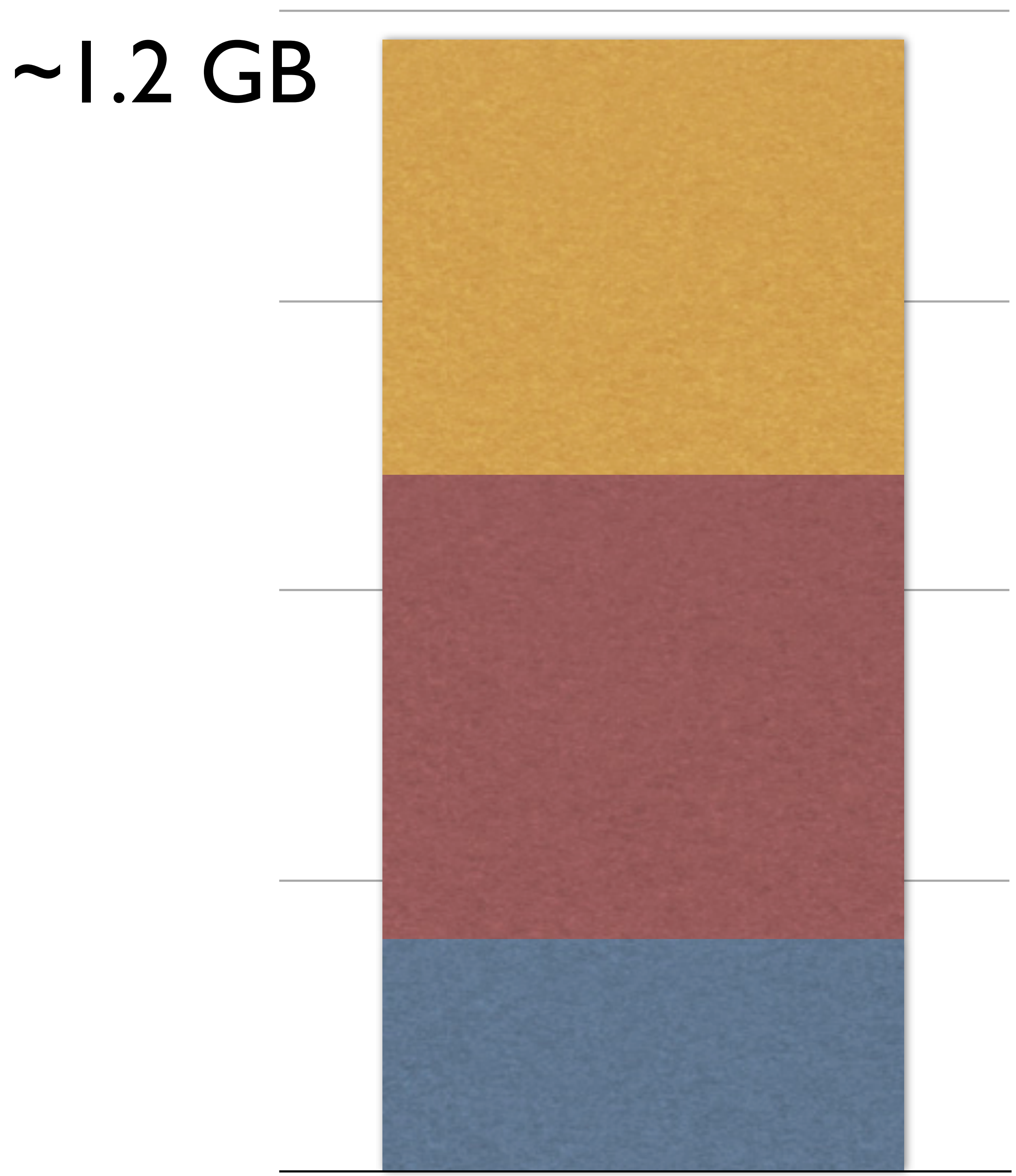
Event specific data

Read only data

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

Code

CMS offline software memory budget



Event specific data

Read only data

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

Code

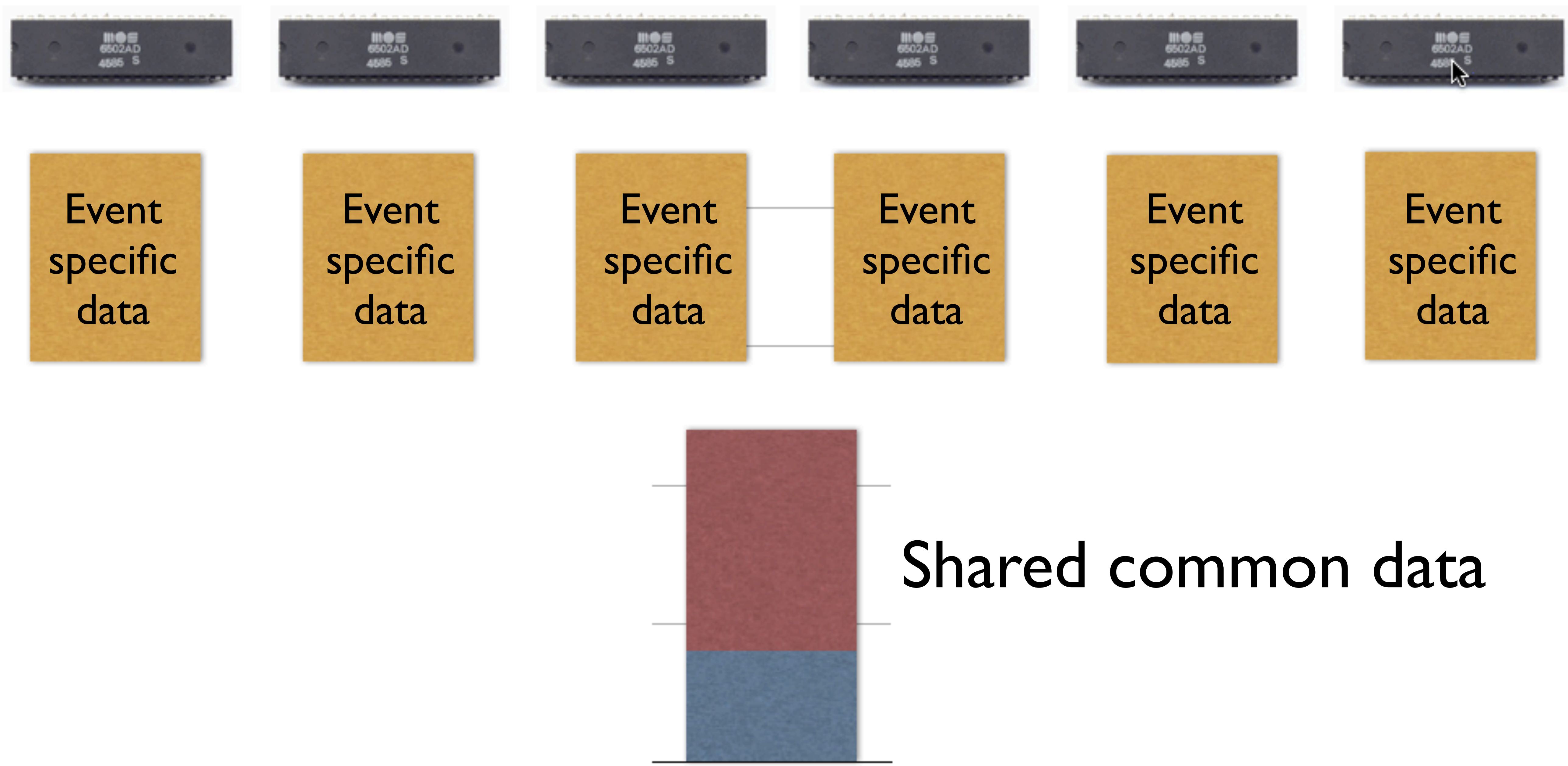
COMMON!

C-o-W*

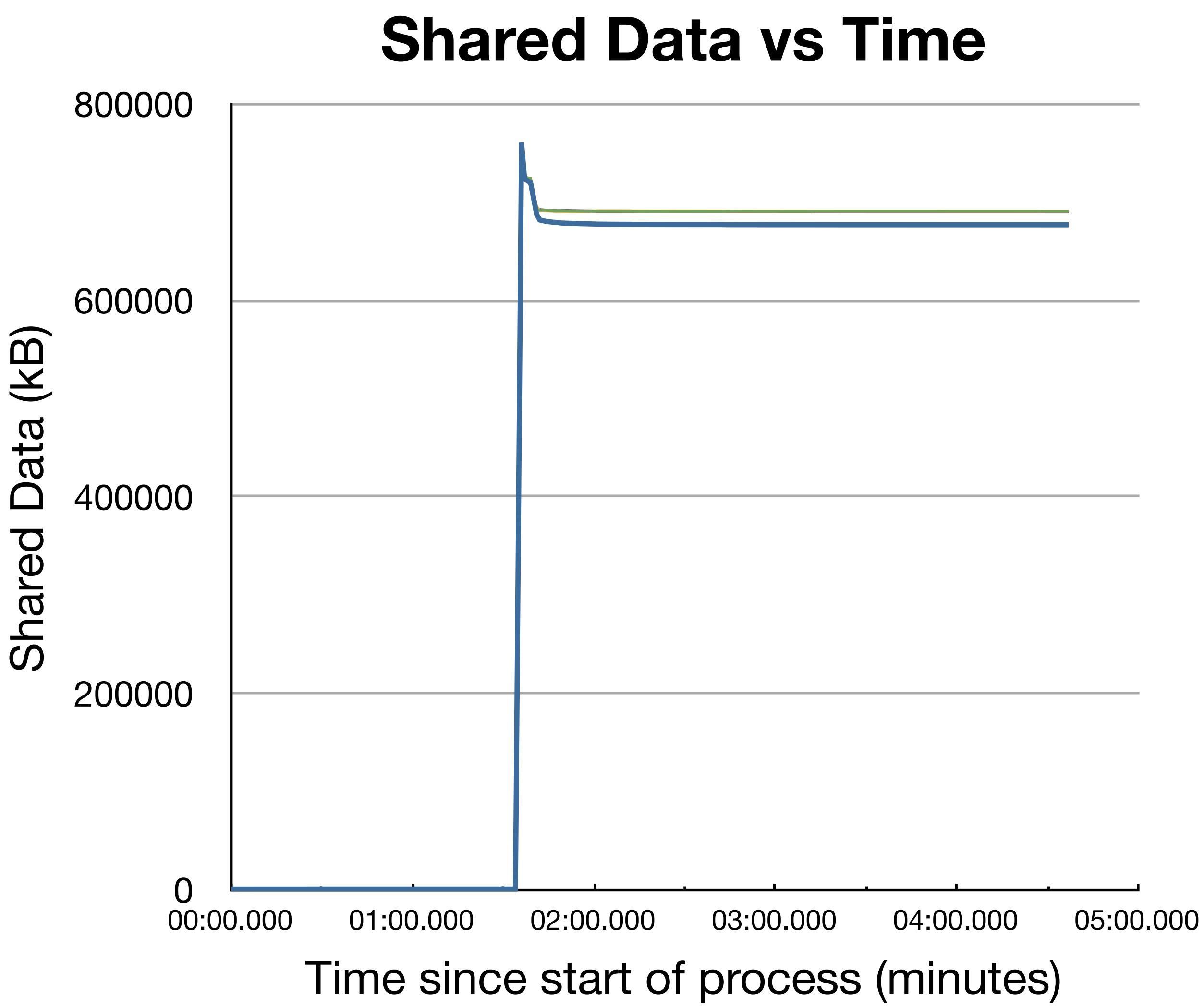
- Most (all?) of the common const data / code can actually be brought in the application very early
- If you fork at that point, the kernel is actually smart enough to share the common data memory pages between parent and the children
- The kernel "un-shares" the memory pages only when one of the processes writes to them
- New allocations (i.e. event data) end up in non shared pages

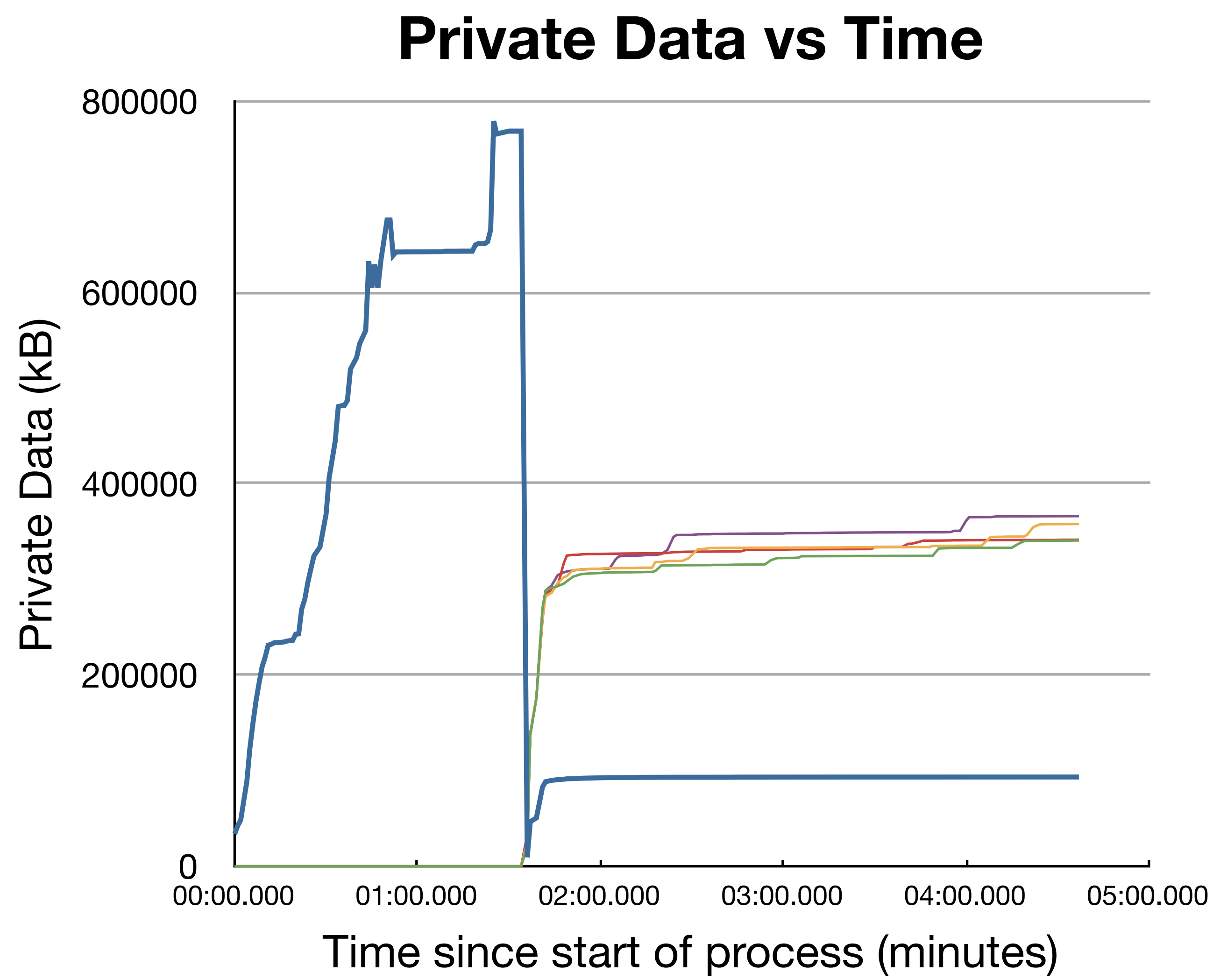
* Copy-on-Write

CMS near future multicore strategy: **forking**



Forking: memory sharing

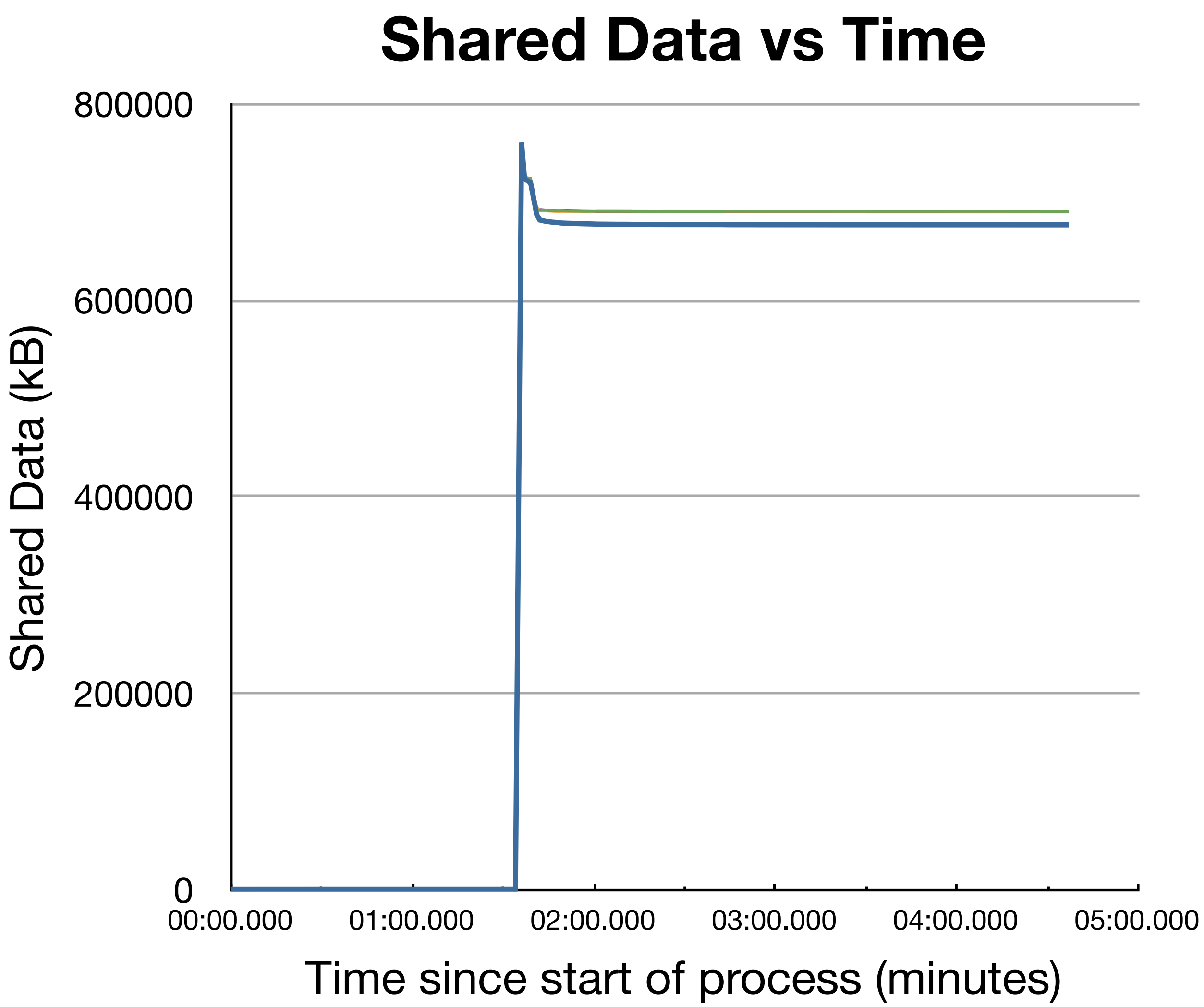


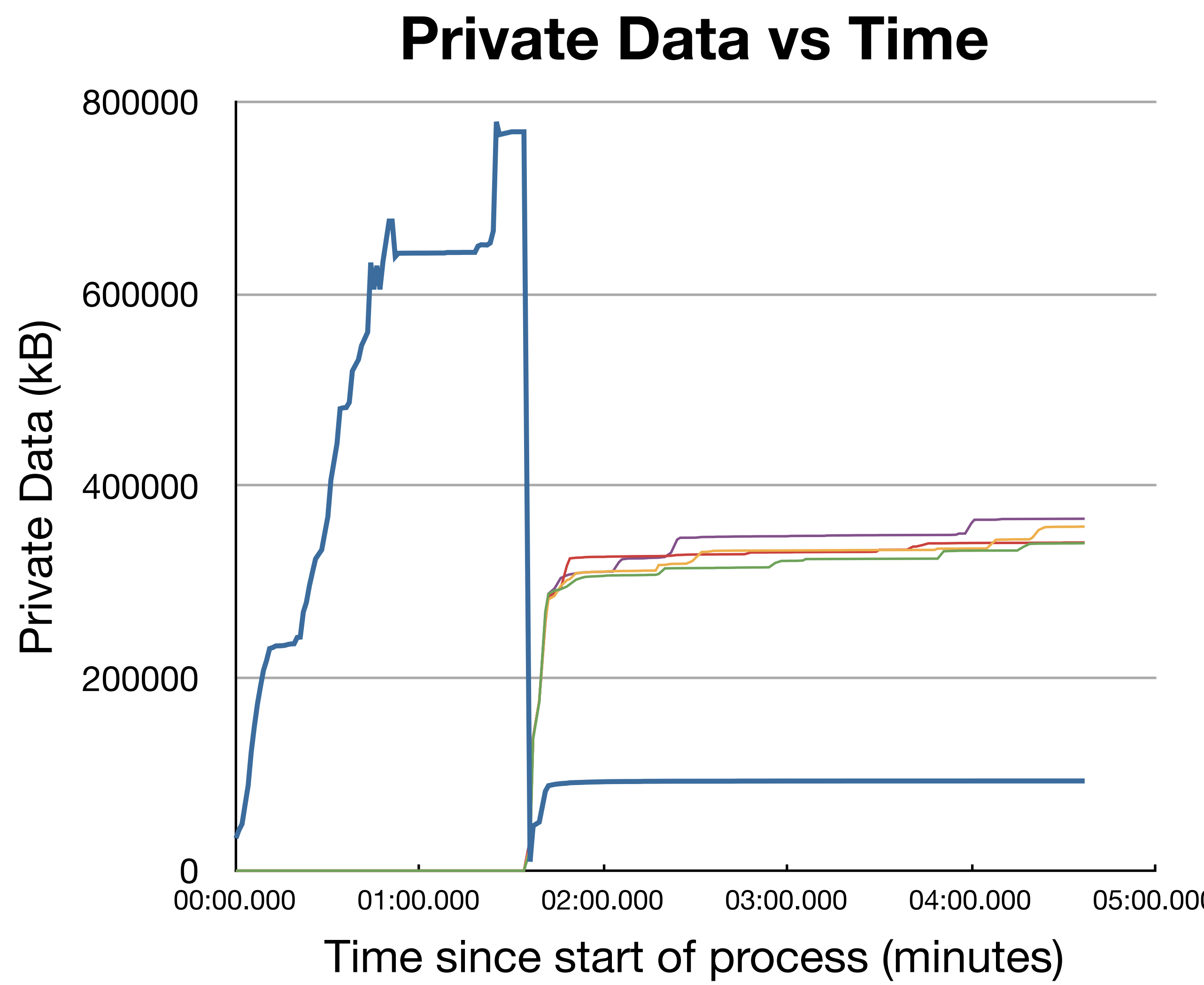


Measurements done using reconstruction with 64bit software on 4 CPU, 8 core/CPU 2GHz AMD Opteron(tm) Processor 6128

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



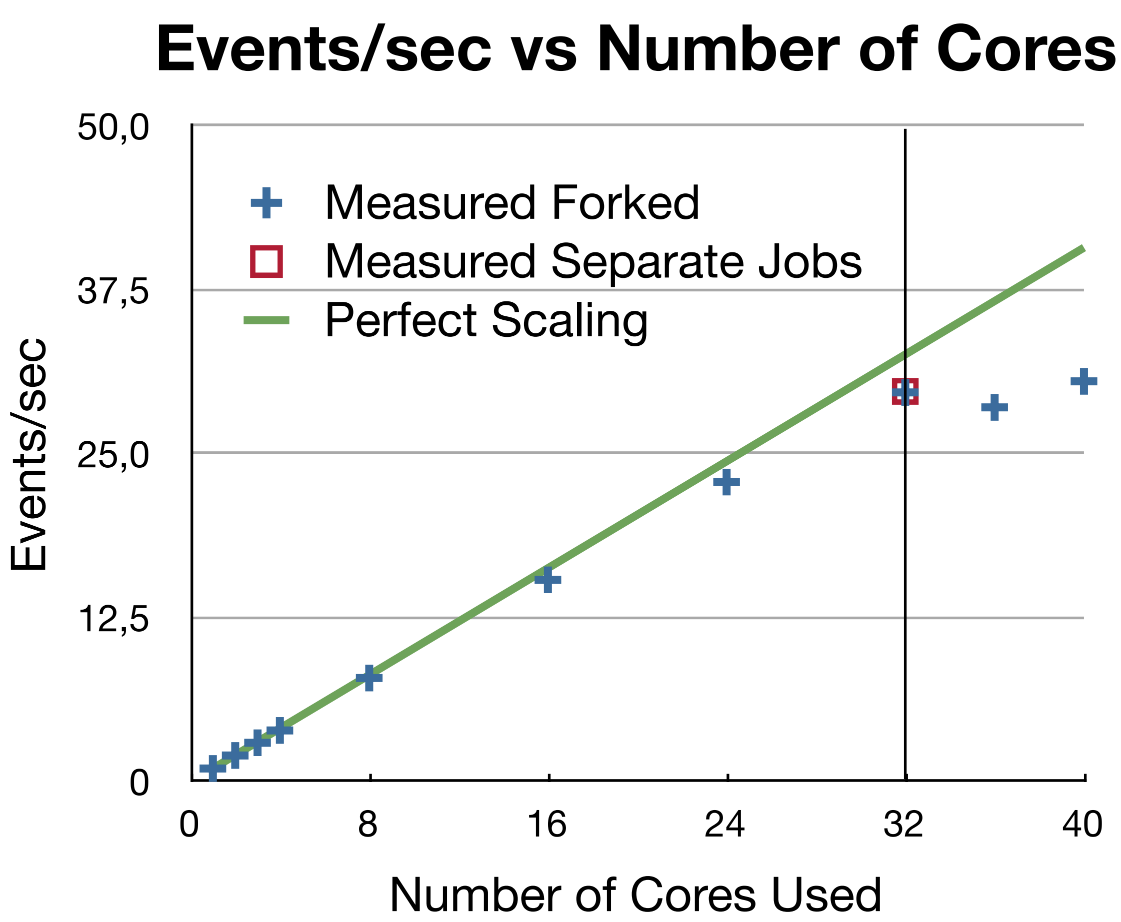


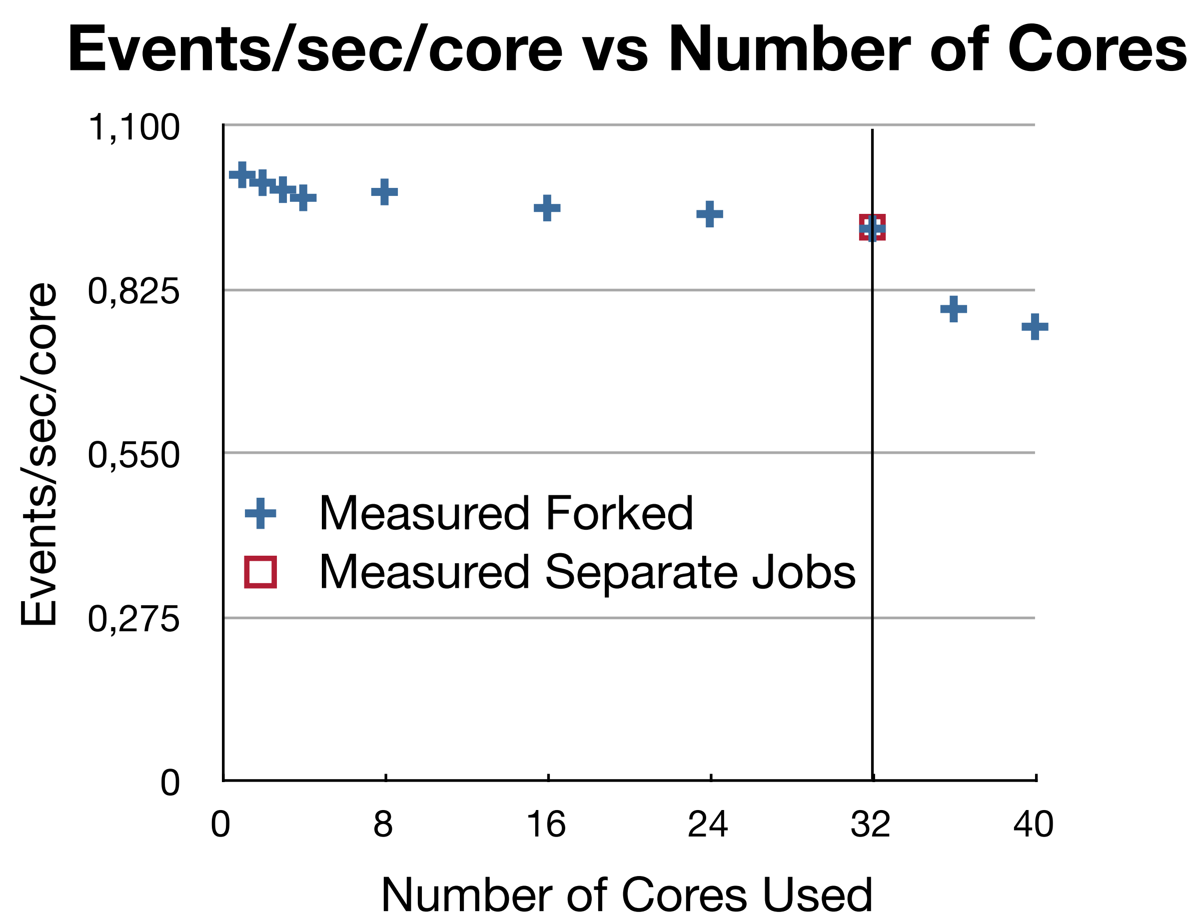
Measurements done using reconstruction with 64bit software on 4 CPU, 8 core/CPU 2GHz AMD Opteron(tm) Processor 6128

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

We suddenly have lots of free memory available

Forking: throughput





Resource accounting

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

Exploiting this new processing model requires a new model in computing resources allocation as well

Experiments need to have control over a larger quantum of resources

as multi-core aware jobs require scheduling of multiple cores at the same time

Correct resource accounting fundamental (and gets trickier)

“Whole-node” scheduling

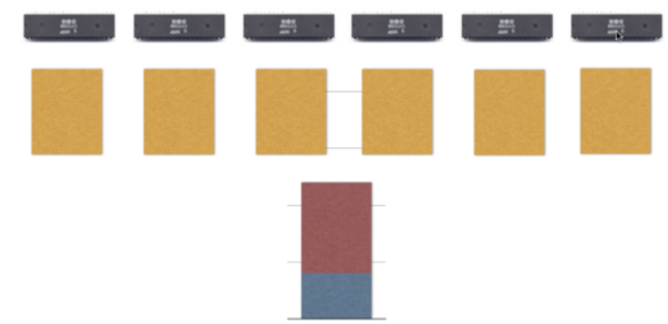
One natural unit in the system is the “whole node”: the physical thing running one (unvirtualized) copy of the OS and sharing a set of resources (CPU, disk, network, etc.)

The applications explicitly take over the management of the sharing of resources within the “whole node” quantum of resources

Compatible with current *modus-operandi*, will allow moving to **forking** / multi-threading, allowing for optimization of data/workflow management: I/O caching, local merging, etc

Sites only need to care about 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



Whole-Node Job Submission Task Force*

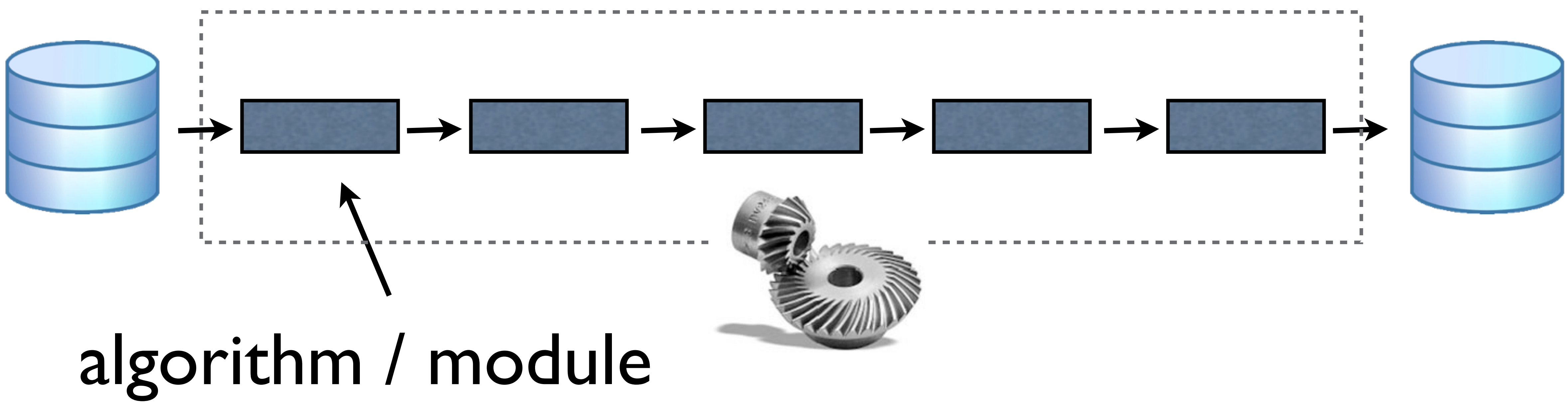
whole-node-task-force@cern.ch

(chaired by Peter.Elmer@cern.ch)

*LCG-MB mandated

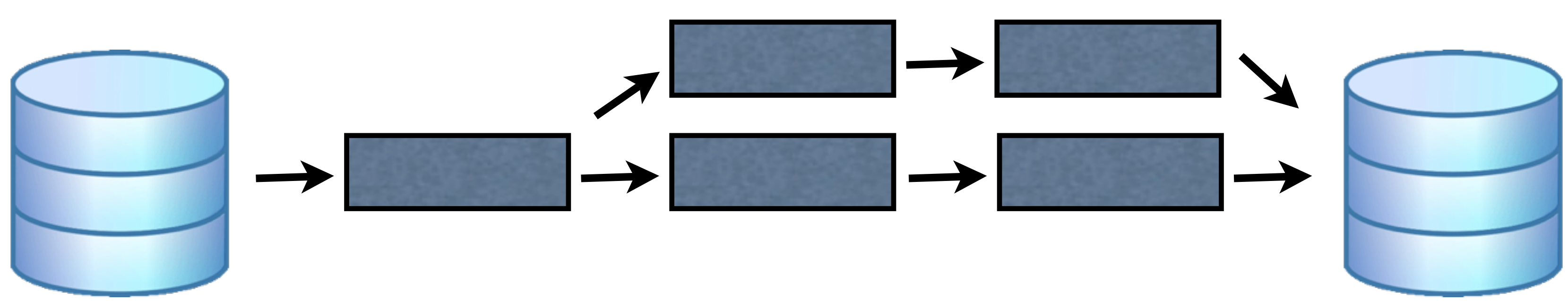
Far future(?): multi-threading

Current single threaded processing



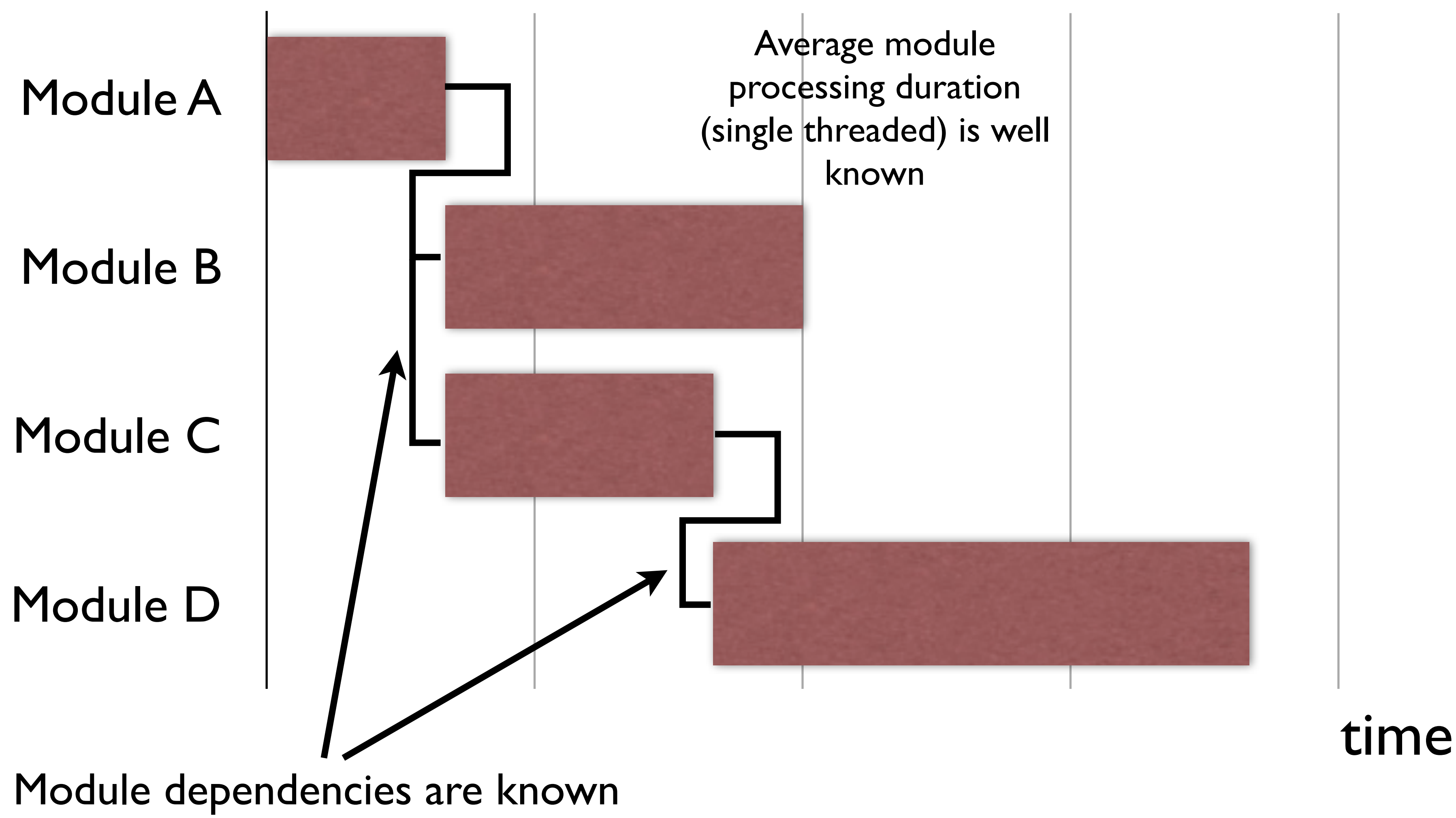
Far future(?): multi-threading

Unrelated parts could be elaborated by separate threads to increase throughput

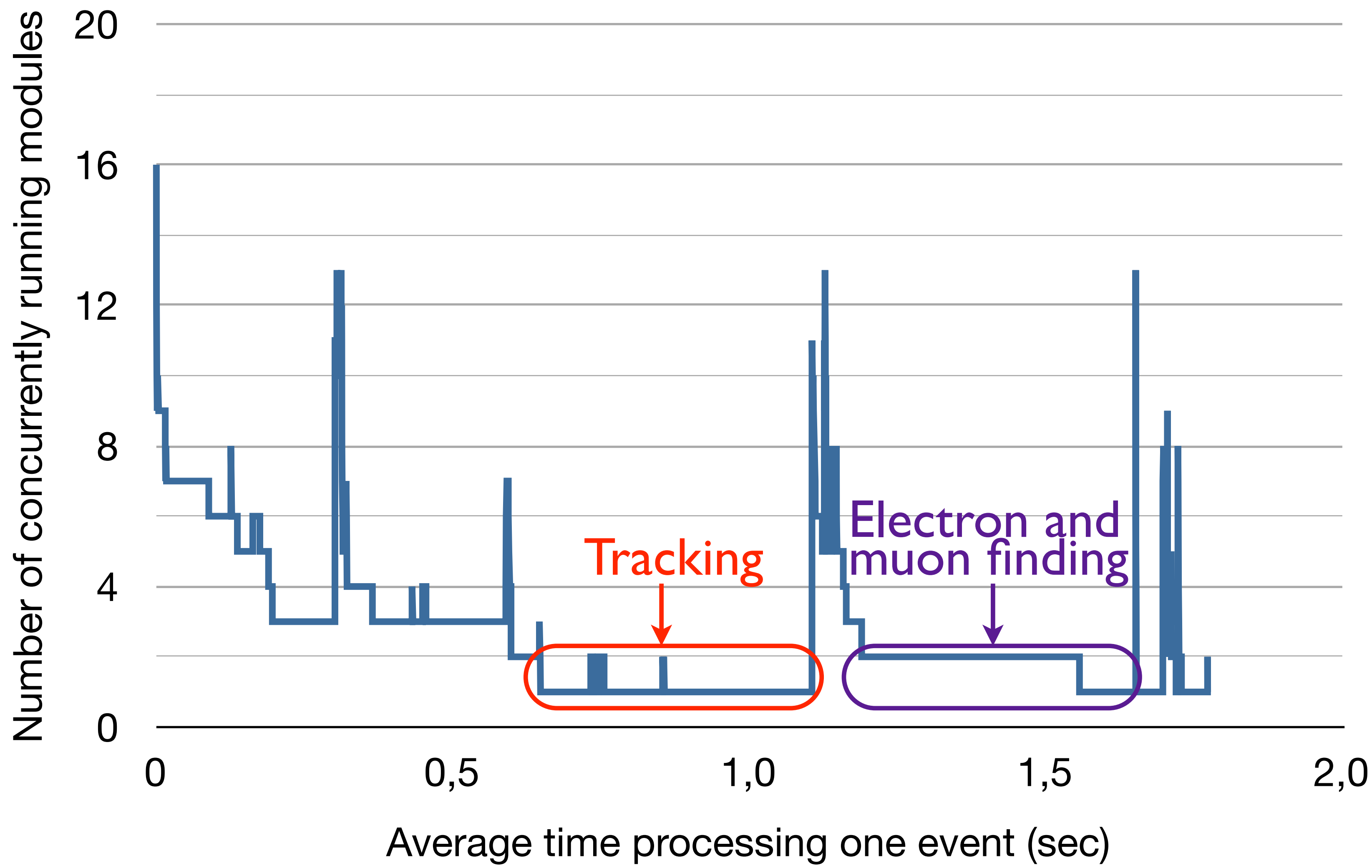


This is theoretically interesting but in practice **not worth the effort**!

Behavior / bottlenecks can be "estimated" even now



Number of Running Modules vs Time for TTBar RECO



Conclusions

- **64bit migration done**
- **Forking** proves to be effective and enough of a no-brainer for being considered a good strategy for the short - medium term
- The effort which would be required to have module level parallelism is not worth the actual gain given the current decomposition of algorithms
- **Deployment of whole-node scheduling and associated system level accounting key to exploiting multi-core**