# Technical performance status

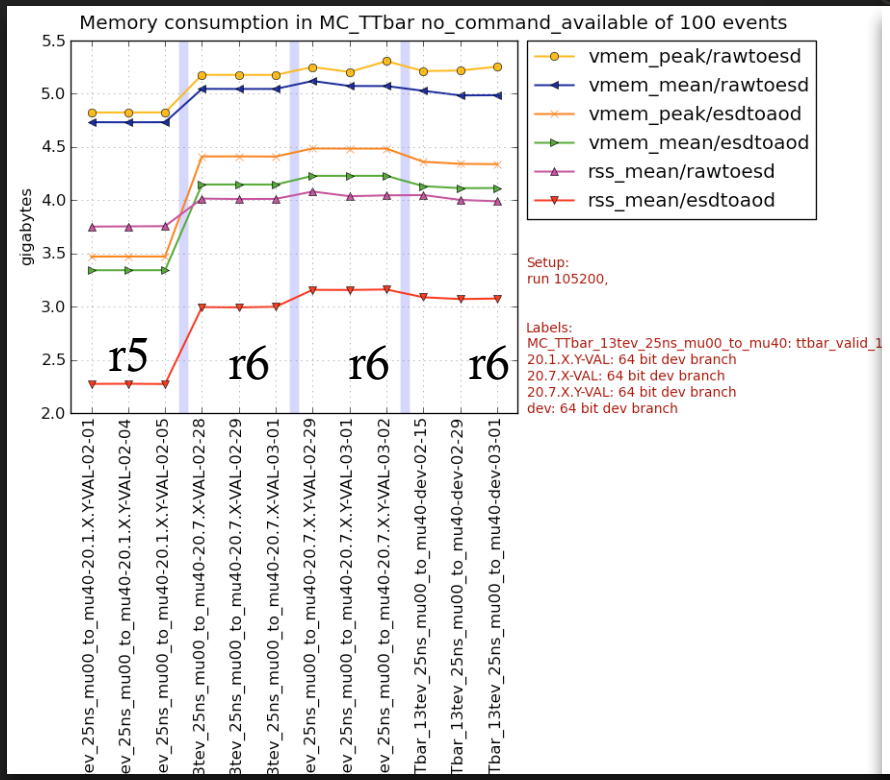
Antonio Limosani

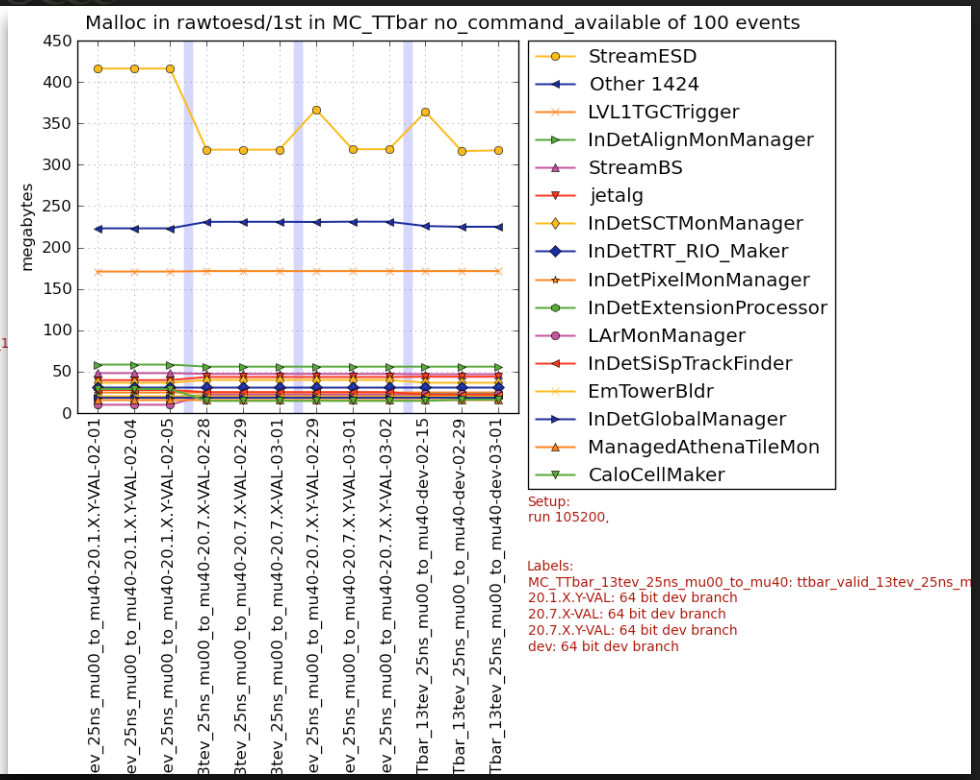
University of Sydney

@ROOT Planning Meeting,

2nd February 2016 .

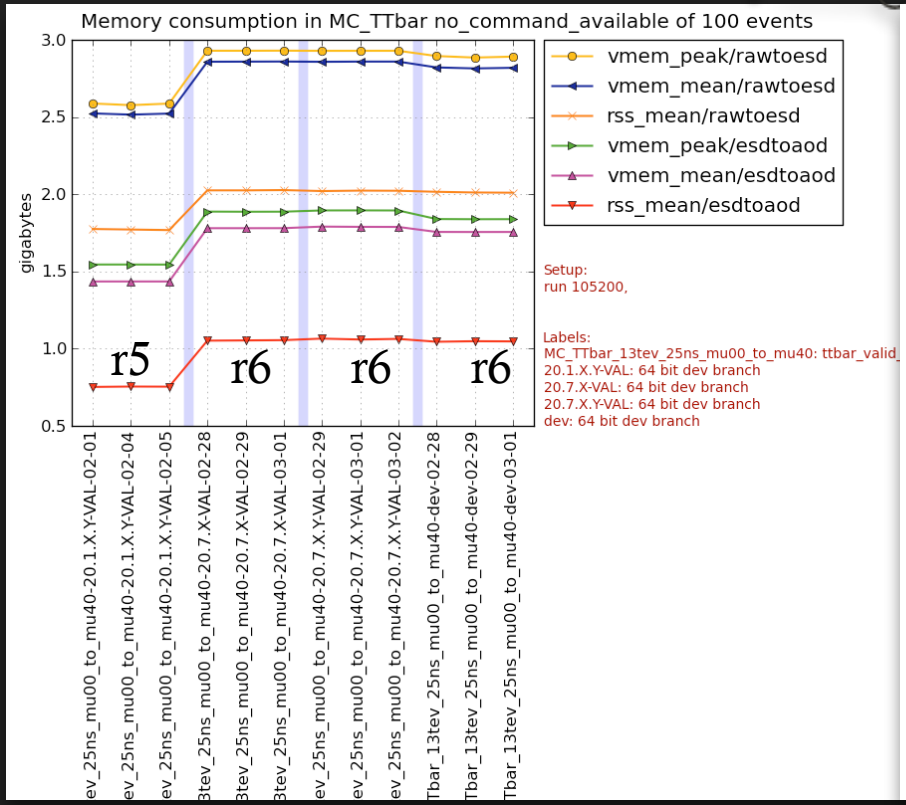
# ATLAS full workflow

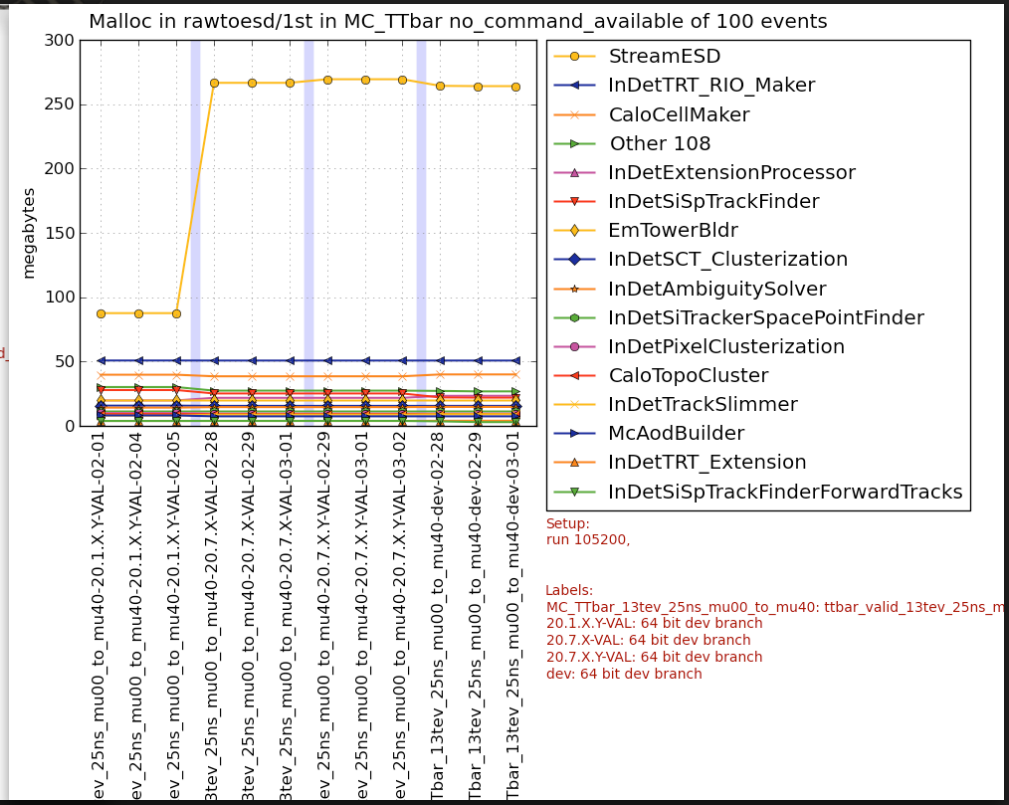



- MC : RDO to ESD/AOD (monitoring & trigger)
- rss_mean 3.8 GB to 4.0 GB
- At the start of this exercise to reduce the memory the gap was 0.6 GB in RDO to ESD

# ATLAS system workflow




- MC : RDO to ESD/AOD (System) Much reduced workload
- rss_mean 1.8 GB to 2.0 GB
- "PerfMon" is limited to finding where in ATLAS software the memory increase occurs, removing an include in one area where it's not needed, may mean it gets pulled in somewhere else where it's needed.

# ATLAS@Tier0 workflow

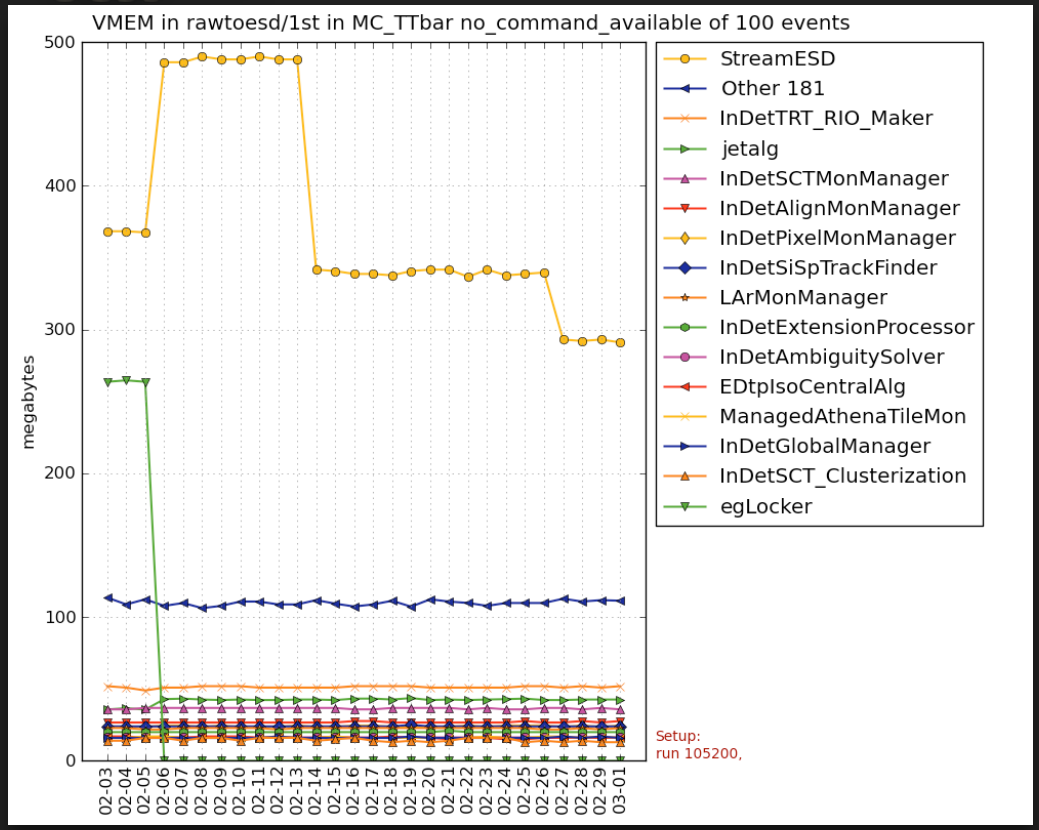
- This is the workflow that really counts and where the alarm was raised. Multi-processor jobs, sharing of memory - "PerfMon" tests are single-core
- Key metric : avg PSS, real resident memory taking into account sharing of memory between cores.
- Preliminary test in nightly (20.7.X-VAL,rel_2) shows for avgPSS we have recovered performance in 20.1 and in fact exceeded it in this measure.
- maxPSS may be a concern, may reflect AutoFlush setting change from 5 to 10? To improve I/O utilisation

ATLAS@Tier0 Memory Usage (Armin Nairz)

| n_cores=4 | 20.1.8.4 | 20.7.4.1 | 20.7.5.1 | 20.7.X-VAL,rel2 |
|---|---|---|---|---|
| avgPSS | 7.5 | 8.9 | 8.5 | 7.2 |
| avgRSS | 13.9 | 14.5 | 14.3 | 10.2 |
| avgSwap | 0.0 | 0.1 | 0.0 | 0.0 |
| avgVMEM | 20.2 | 21.5 | 20.6 | 14.3 |
| | | | | |
| maxPSS | 9.2 | 10.9 | 10.0 | 12.3 |
| maxRSS | 16.4 | 17.2 | 17.0 | 17.3 |
| maxSwap | 0.0 | 0.1 | 0.0 | 0.0 |
| maxVMEM | 23.2 | 25.8 | 24.7 | 24.5 |

# ROOT I/O

- Peter Van Gemmeren & Marcin Nowak removed instances of transient class definitions being included in header files that provide persistent class definitions. Our transient event data model classes should not be included in dictionaries created by ROOT6. Real gains demonstrated here
- Some gains still to come
  - ROOT-7972 - Philippe Canal "updated the v6.04 and v6.06 patch branches as well as the master branch with a series of commits that should solve the spurious autoparsing associated with the STL collections that are base classes, like _Vector_base<CaloEnergy_p2,allocator<CaloEnergy_p2> >"
  - Also Peter found our containers contain highly compressible data, which fill buffers/memory before being flushed - onus on ATLAS to get developers to optimise memory footprint of containers




```
Br1102 :xAOD::CaloClusterAuxContainer_v2_CaloCalTopoClustersAuxDy
n.CellLink.m_persKey : Total Size= 38064 bytes File Size = 1063
Baskets : 5 : Basket Size= 43008 bytes Compression= 34.90
Br1124 :xAOD::CaloClusterAuxContainer_v2_CaloCalTopoClustersAuxDy
n.ENG_BAD_CELLS : Basket Size= 43520 bytes Compression= 16.67
Br
1125 :xAOD::CaloClusterAuxContainer_v2_CaloCalTopoClustersAuxDyn.
N_BAD_CELLS : Basket Size= 43520 bytes Compression= 20.34
Br1126 :xAOD::CaloClusterAuxContainer_v2_CaloCalTopoClustersAuxDy
n.BAD_CELLS_CORR_E : Basket Size= 43520 bytes Compression= 16.57
```

# TTreeCache::Read()

- Using FOMTool (N. Rauschmayr & S. Kama) we identified memory hotspot in real data job. Many instances of

```
From 0xa55a8000 to 0xa55b2fff Size 43 kB

0        malloc+0x6d in [ /afs/cern.ch/sw/lcg/contrib/gcc/4.9.1/x86_64-slc6/include/c++/4.9.1/bits/atomic_base.h:308,/afs/
cern.ch/user/n/nrauschm/publicworkspace/FOMTools/UnusedMemory/src/mallocinterpose.cxx:417, ]
275      _Znwm+0x18 in [ ../../../../gcc-4.9.1/libstdc++-v3/libsupc++/new_op.cc:50, ]
276      _Znam+0x9 in [ ../../../../gcc-4.9.1/libstdc++-v3/libsupc++/new_opv.cc:33, ]
8088     _ZN14TFileCacheReadC1EP5TFileiP7TObject+0x287 in [ ??:0, ]
8073     _ZN10TTreeCacheC1EP5TTreei+0x24 in [ ??:0, ]
8074     _ZN5TTree15SetCacheSizeAuxEbx+0x33b in [ ??:0, ]
11675    _ZN5TTree4FillEv+0x484 in [ ??:0, ]
11676    _ZN4pool17RootTreeContainer11writeObjectERNS_14DbContainerImp12_TransactionE+0xe6d in [ ??:0, ]
11677    _ZN4pool14DbContainerImp4saveERKNS_14DbObjectHandleIvEE+0x10f in [ ??:0, ]
11678    _ZN4pool14DbContainerObj4saveERKNS_14DbObjectHandleIvEEPKNS_10DbTypeInfoE+0x52 in [ ??:0, ]
11679    _ZN4pool11DbContainer5_saveERKNS_14DbObjectHandleIvEEPKNS_10DbTypeInfoE+0x66 in [ ??:0, ]
11715    _ZN4pool13DbDatabaseObj8addShapeEPKNS_10DbTypeInfoE+0x5df in [ ??:0, ]
```

- Why should TTree read cache be initialised during TTree::Fill()?

- For RAW to ESD, or anything, there should not be the need for ROOT TTreeCache. But it's used inadvertently by TTree::Fill().

- https://sft.its.cern.ch/jira/browse/ROOT-8031

- export ROOT_TTREECACHE_SIZE=0 releases : RSS memory 13(10) MB reduction in MC(Data), big saves in VMEM

```
q431 : patched "export ROOT_TTREECACHE_SIZE=0"

16:45:49 PMonSD ======================================= step 1st ====
16:45:49 PMonSD          n    cpu    vmem  malloc component
16:45:49 PMonSD [1st]    1   1380  150816  135992 StreamESD

q431 : clean

16:46:51 PMonSD ======================================= step 1st ====
16:46:51 PMonSD          n    cpu    vmem  malloc component
16:46:51 PMonSD [1st]    1   1390  274496  256731 StreamESD
```

# Simplest test : Just athena

- I've just now completed an even simpler test. I don't run any reconstruction.
- All I run is a dummy event loop for 100000 events in athena and run PerfMon to measure the memory footprint. The memory footprint is 200MB larger in 20.7 than in 20.1.
- /afs/cern.ch/user/l/limosani/public/log.dummyjob.20.{1,7}
- jo.py : theApp.EvtMax = 100000

```
20.1 (ROOT 5)
…
Info in <TCint::AutoLoadCallback>: loaded dependent library libGaudiPythonDict for class
basic_ostream<char,char_traits<char> >
Info in <TCint::AutoLoadCallback>: loaded library libGaudiPythonDict for class
basic_ostream<char,char_traits<char> >
Info in <TCint::AutoLoad>: loaded library xAODEventInfoDict.so for class pair<string,string>
…
PMonSD [---] vmem_peak=336684 vmem_mean=336680 rss_mean=123916


20.7 (ROOT 6)
…
Info in <TInterpreter::Autoparse>: >>> RSS key DataObject - before 139.840 MB - after 241.624 MB - delta 101.784
MB
Info in <TInterpreter::Autoparse>: >>> RSS key Chrono - before 245.392 MB - after 282.752 MB - delta 37.360 MB
…
PMonSD [---] vmem_peak=584700 vmem_mean=584680 rss_mean=326188
```

# Compare maps between 20.1 and 20.7

- Some more analysis on the "do nothing" Athena job. When I investigate the /proc/<pid>/maps file I find the following top 5 libraries (.so) for reserving memory. 20.7 adds about 100 MB of memory due to allDict.cxx.pch and libCling.so

- https://sft.its.cern.ch/jira/browse/ROOT-7972

```
20.1
2 MB libresolv-2.12.so
3 MB libGaudiKernel.so
4 MB libGaudiPythonDict.so
5 MB libHist.so
7 MB libCore.so

20.7
  3 MB libGaudiKernel.so
  3 MB libCore.so
  3 MB libHist.so
 30 MB libCling.so
105 MB allDict.cxx.pch
```

- Mapped memory, so is it real? /proc/${PID}/smaps

```
allDict.cxx.pch
Size:             107792 kB
Rss:               59860 kB
Pss:               59860 kB

libCling.so
Size:              31200 kB
Rss:               12520 kB
Pss:               12520 kB
```

- In the end this memory should be shared across cores

# Other ATLAS workflows : High Level Trigger

20.7 (ROOT 6)

```
HLT_physicsV6_menu_rerundb/atn_test.log:Info in <TInterpreter::Autoparse>:
>>> RSS key coral::Attribute - before 188.392 MB - after 307.860 MB - delta 119.468 MB

HLT_physicsV6_menu_rerundb/atn_test.log:Info in <TInterpreter::Autoparse>:
>>> RSS key DataObject - before 416.252 MB - after 428.752 MB - delta 12.500 MB


HLT_physicsV6_menu_rerundb/atn_test.log:Info in <TInterpreter::Autoparse>:
>>> RSS key Chrono - before 431.140 MB - after 458.960 MB - delta 27.820 MB
```

Stewart Martin-Haugh

"I had a look at the coral::Attribute file:

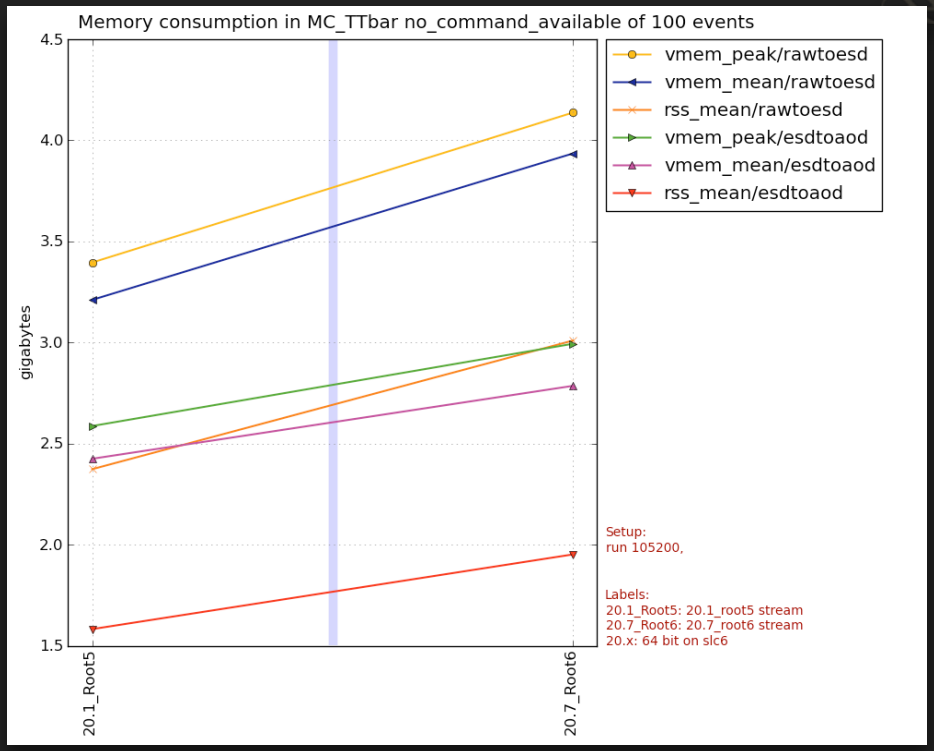
/afs/cern.ch/sw/lcg/releases/LCG_81c/CORAL/3_1_0/x86_64-slc6-gcc49-opt/include/CoralBase/Attribute.h

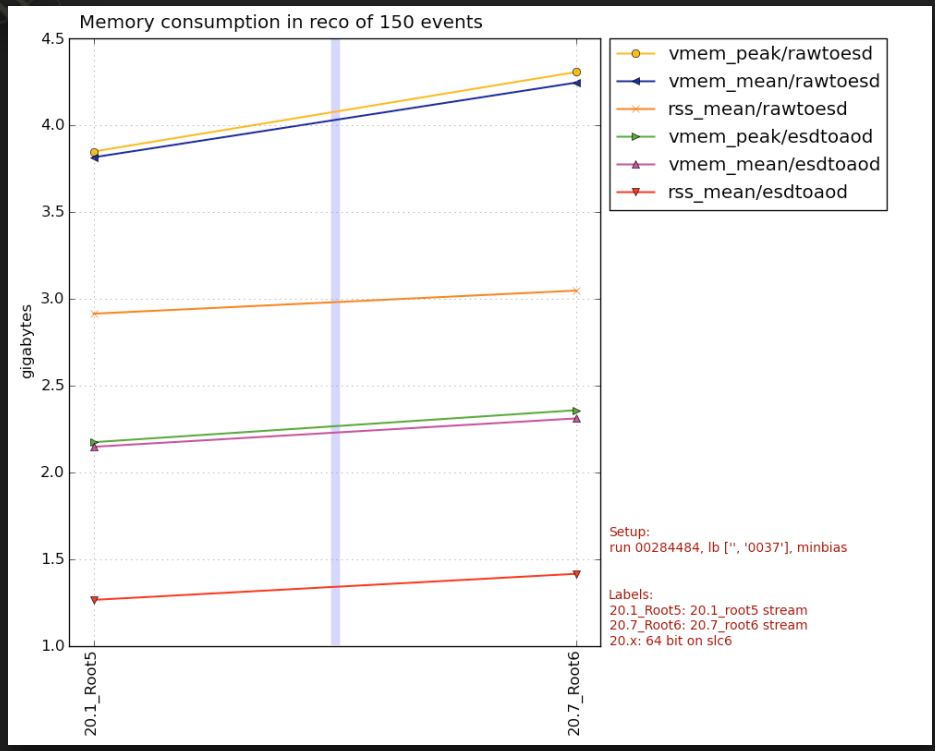
and I can't see much room for improvement at first glance. I would guess most of the overhead comes from the templated methods."

# Summary

- Easy gains have probably already been made, namely egLocker and StreamESD, and auto parsing
- Remaining gains will be harder because they require algorithm re-writes
- Crisis has been brought under control. Must pursue improvements because of a new potential workflow which skips ESD writing to improve throughput at Tier0 by avoiding unnecessary I/O
- Big thanks to improvements and feedback from the ROOT team

# ATEAM-259




- "Tests from Armin Nairz using latest 20.7 show that the PSS has jumped by 400MB per core in Tier-0 reconstruction tests. This would be +1.6GB per Tier-0 MP job slot, which is way more that we could possibly absorb and would completely jeopardise the Tier-0 for data taking." ATEAM-259 **"Reduce memory consumption in 20.7 RAWtoESD c.f. 20.1"**

- Increase is evident between 20.1 and 20.7, a major change has been the migration from Root5 to Root6 (PMB jobs show 600 MB increase in MC+monitoring and and increase of 100 MB Data jobs (q431).

- http://atlas-pmb.web.cern.ch/atlas-pmb/slc6/arch-mon-monitoring-reco-data15-main/

- http://atlas-pmb.web.cern.ch/atlas-pmb/slc6/arch-mon-monitoring-reco-mc15-ttbar-valid-13tev-25ns-mu00-to-mu40/

# Auto parsing in standard trigger job

- trigtest.pl --test AthenaTrigRDO_MC_pp_v6 --run AthenaTrigRDO_MC_pp_v6 --conf TriggerTest.conf
- The big are:
- Info in <TInterpreter::Autoparse>: >>> RSS key coral::Attribute - before 199.736 MB - after 319.756 MB - delta 120.020 MB
- Info in <TInterpreter::Autoparse>: >>> RSS key DataObject - before 366.172 MB - after 393.896 MB - delta 27.724 MB
- Info in <TInterpreter::Autoparse>: >>> RSS key Chrono - before 396.760 MB - after 426.736 MB - delta 29.976 MB
- Info in <TInterpreter::Autoparse>: >>> RSS key egammaPID::ROOT6_NamespaceAutoloadHook - before 506.536 MB - after 550.052 MB - delta 43.516 MB

# /proc/$PID/maps

- Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process or thread. Each row has the following fields:

```
address           perms offset  dev   inode   pathname
08048000-08056000 r-xp 00000000 03:0c 64593   /usr/sbin/gpm
```

- **address** - This is the starting and ending address of the region in the process's address space
- permissions - This describes how pages in the region can be accessed. There are four different permissions: read, write, execute, and shared. If read/write/execute are disabled, a '-' will appear instead of the 'r'/'w'/'x'. If a region is not shared, it is private, so a 'p' will appear instead of an 's'. If the process attempts to access memory in a way that is not permitted, a segmentation fault is generated. Permissions can be changed using the mprotect system call.
- offset - If the region was mapped from a file (using mmap), this is the offset in the file where the mapping begins. If the memory was not mapped from a file, it's just 0.
- device - If the region was mapped from a file, this is the major and minor device number (in hex) where the file lives.
- inode - If the region was mapped from a file, this is the file number.
- pathname - If the region was mapped from a file, this is the name of the file. This field is blank for anonymous mapped regions. There are also special regions with names like [heap], [stack], or [vdso]. [vdso] stands for virtual dynamic shared object.