CMS Software Performance Strategies

Peter Elmer – Princeton University
Giulio Eulisse, Lassi Tuura – Northeastern University
Vincenzo Innocente - CERN
(and many people developing software in CMS)

CHEP 09, Prague
23 Mar, 2009

Introduction

The advantages of improvements to the performance and efficiency of our software are clear. For a fixed set of available resources, such improvements imply we can:

- use fewer resources for task X (use the rest for additional tasks Y, Z, ...)
- use the same resources, but get the task done more quickly

The fewer computing resources a given task needs, the less likely it is to get entangled in computing "workflow" problems:

- grid infrastructure failures, queue time limits, memory limits, storage system I/O limits and failures, etc.

Alternatively, the entire cost of the project can be reduced.

What do we mean by performance?

There are many things one can label as “software performance”:

- CPU time per processed event
- Memory footprint of application
- CPU/wall clock time ratios
- I/O rates and patterns
- Event data sizes, transaction rates on databases, application startup time, software compilation times, and many other things, etc. etc.

There are also many use cases from bulk production to analysis.
To keep things simple, I'll focus in this presentation on the first two.

Overview

In pursuit of improved software performance CMS has transitioned from a dedicated "Performance Task Force" to a regime in which such work happens more routinely and systematically as part of the release planning, integration and validation.

In this talk I will describe the various strategies and tools we have been developing and using to pursue improved software performance and give some specific examples and benchmarks to give an idea of the gains achieved!

CPU utilization

Three broad classes of changes which affect the CPU performance:

1. Physics algorithmic – the program output changes
   - Extra cuts, simulating extra or more detailed effects, "Do we run the extra trackfinder?"
2. Algorithmic, but technical – the program output does not change
   - Caching, lazy evaluation, removal of redundant calculations, data structures, etc.
3. Purely technical – the program output does not change
   - Changes related to specific issues in C++, the memory management, the operating system, the compilers and the hardware where are applications run

By far the largest gains/losses come from #1, of course. Improvements from #2 and #3 are "free beer" in that there are no trade-offs with physics (though there may be trade-offs between CPU use and memory, etc.)

General code improvements

Strategy 1: As a consequence of our Performance Task Force, several groups now routinely include work on improving performance into their development plans in addition to "new feature" development/changes (some of which increase the CPU requirements)

See for example the talk on CMS simulation tuning by F. Cossutti

Strategy 2: Monitor in detail the performance over time, watching for increases and also looking for new possibilities for code performance improvements, improving things continuously.

No "crisis, boom and bust" cycle for performance!

I don't have time in this presentation to cover all of the specific things we've looked at, but there is one very pervasive one in the which I will discuss: **the (ab)use of dynamic memory.**

(In the free beer category!)

Dynamic memory allocation

From igprof (sampling) performance measurements, we have seen in the past that our applications take 20% or more of the time explicitly in memory related operations (malloc/free/new/delete).

In addition these memory operations are often associated with other activity such as temporaries, copy ctors, dtors and (potentially) indirect performance loss from stalls.

Observation: memory operations are often accidental, unnecessary, can be reduced in number or eliminated altogether. The way they originate often results in CPU time and/or instructions spread across multiple functions.

Strategy: use the number of memory allocations itself as a first-level benchmark for optimization (smaller is better), in addition to optimizing using CPU time and/or instructions as a benchmark.

igprof - memory allocations

```
Counter: MEM_TOTAL

-------------------------------------------------------------------------------
Flat profile (cumulative >= 1%)

% total              Total              Calls  Function
  100.0  322'568'440'598    2'810'467'694  <spontaneous> [1]
  100.0  322'568'440'598    2'810'467'694  __libc_start_main [2]
  <...>
-------------------------------------------------------------------------------
Flat profile (self >= 0.01%)

% total               Self              Calls  Function
   16.43   53'001'649'472        427'576'864  G4NavigationHistory::G4NavigationHistory(G4NavigationHistory const&) [40]
   15.37   49'591'809'600        427'515'600  G4Transportation::PostStepDoIt(G4Track const&, G4Step const&) [29]
    8.92   28'757'984'245          2'187'563  TBasket::ReadBasketBuffers(long long, int, TFile*) [38]
    6.48   20'901'969'200          2'195'585  inflateInit2_ [46]
    6.23   20'099'701'776        179'461'623  TrackingAction::PreUserTrackingAction(G4Track const*) [49]
    4.93   15'891'135'434          2'192'939  TBuffer::TBuffer(TBuffer::EMode, int) [53]
    <...>
    0.41    1'320'651'768         22'232'517  std::vector<CLHEP::Hep3Vector, std::allocator<CLHEP::Hep3Vector> >::_M_insert_aux(__gnu_cxx::__n
    <...>
    0.30      969'483'000         15'910'239  std::vector<G4CascadParticle, std::allocator<G4CascadParticle> >::operator=(std::vector<G4Cascad
    <...>

-------------------------------------------------------------------------------
Call tree profile (cumulative)

            <...>
            95.8  ..............  309'122'379'267 / 309'122'379'267    2'711'999'373 / 2'711'999'373      edm::WorkerT<edm::EDProducer>::im
[18]        95.8  309'122'379'267                0 / 309'122'379'267    2'711'999'373                    edm::EDProducer::doEvent(edm::Event
            80.7  ..............  260'281'563'356 / 260'281'563'356    2'366'681'410 / 2'366'681'410      OscarProducer::produce(edm::Event
             6.2  ..............   20'048'772'685 / 20'048'772'685       43'669'316 / 43'669'316         SiStripDigitizer::produce(edm::Ev
            <...>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Rank    % total               Self             Self / Children                 Calls / Total            Function
            80.7  ..............  260'281'563'356 / 309'122'379'267    2'366'681'410 / 2'711'999'373      edm::EDProducer::doEvent(edm::Eve
[21]        80.7  260'281'563'356          27'600 / 260'281'535'756    2'366'681'410                    OscarProducer::produce(edm::Event&,
            80.6  ..............  260'004'420'536 / 260'004'420'536    2'366'670'088 / 2'366'670'088      RunManager::produce(edm::Event&,
             0.1  ..............      174'215'776 / 174'215'776              400 / 400                    CaloSD::fillHits(std::vector<PCal
            <...>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Bytes

Number of Allocations

An example (and abbreviated) igprof memory allocation report. It contains two flat profiles (allocations within a function and its callees, allocations within a function) and also a call graph.

igprof - memory allocations

```
Counter: MEM_TOTAL

-------------------------------------------------------------------------------
Flat profile (cumulative >= 1%)

% total            Total            Calls  Function
  100.0  322'568'440'598    2'810'467'694  <spontaneous> [1]
  100.0  322'568'440'598    2'810'467'694  __libc_start_main [2]
  <...>
-------------------------------------------------------------------------------
Flat profile (self >= 0.01%)

% total             Self            Calls  Function
   16.43   53'001'649'472      427'576'864  G4NavigationHistory::G4NavigationHistory(G4NavigationHistory const&) [40]
   15.37   49'591'809'600      427'515'600  G4Transportation::PostStepDoIt(G4Track const&, G4Step const&) [29]
    8.92   28'757'984'245        2'187'563  TBasket::ReadBasketBuffers(long long, int, TFile*) [36]
    6.48   20'901'969'200        2'195'585  inflateInit2_ [46]
    6.23   20'099'701'776      179'461'623  TrackingAction::PreUserTrackingAction(G4Track const*) [49]
    4.93   15'891'135'434        2'192'939  TBuffer::TBuffer(TBuffer::EMode, int) [53]
   <...>
    0.41    1'320'651'768       22'232'517  std::vector<CLHEP::Hep3Vector, std::allocator<CLHEP::Hep3Vector> >::_M_insert_aux(__gnu_cxx::__n
   <...>
    0.30      969'483'000       15'910'239  std::vector<G4CascadParticle, std::allocator<G4CascadParticle> >::operator=(std::vector<G4Cascad
   <...>

-------------------------------------------------------------------------------
Call tree profile (cumulative)

           <...>
           95.8  ..............  309'122'379'267 / 309'122'379'267    2'711'999'373 / 2'711'999'373      edm::WorkerT<edm::EDProducer>::im
[18]       95.8  309'122'379'267               0 / 309'122'379'267    2'711'999'373                    edm::EDProducer::doEvent(edm::Event
           80.7  ..............  260'281'563'356 / 260'281'563'356    2'366'681'410 / 2'366'681'410      OscarProducer::produce(edm::Event
            6.2  ..............   20'048'772'685 / 20'048'772'685       43'669'316 / 43'669'316         SiStripDigitizer::produce(edm::Ev
           <...>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Rank    % total             Self            Self / Children              Calls / Total            Function
           80.7  ..............  260'281'563'356 / 309'122'379'267    2'366'681'410 / 2'711'999'373      edm::EDProducer::doEvent(edm::Eve
[21]       80.7  260'281'563'356           27'600 / 260'281'535'756    2'366'681'410                    OscarProducer::produce(edm::Event&,
           80.6  ..............  260'004'420'536 / 260'004'420'536    2'366'670'088 / 2'366'670'088      RunManager::produce(edm::Event&,
            0.1  ..............      174'215'776 / 174'215'776              400 / 400                  CaloSD::fillHits(std::vector<PCal
           <...>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

1 - large number of allocs, copy ctor!
2 - CMS code (recently removed)
3 - vector capacity resizing (M_insert_aux)
4 - more copying (assignment)

Many opportunities for performance improvements become clear when looking at an application with the "dynamic memory allocations" view, ranging from the use of dynamic memory within tight loops (1,2) to the usual C++ craziness (3,4)

CPU time/event by release

Our reconstruction, heavily dominated by CMS code, is an important application to optimize. Here we have made significant progress, a factor of ~3 over the past year (see table).

RECO				
Release	Time/event	Perf ticks	Alloc Rate	Alloc Rate
CMSSW_1_8_4	17.0 s	18.92%	1.17 MHz	99.4 MB/s
CMSSW_2_1_7	5.5 s	12.63%	786 kHz	88.3 MB/s
CMSSW_3_0_X	5.0 s	12.33%	775 kHz	81.0 MB/s
CMSSW_3_1_X	6.0 s	11.80%	762 kHz	82.8 MB/s

Pythia TTbar events

The changes made spanned all three of the categories mentioned earlier, and also cover an additional year's worth of reco development, of course.

The figure-of-merit numbers do not correlate strictly with the total time as there were many other things going on here, but smaller is better and reduce the total time even if something else is pushing it up!

CPU time/event by release

The other application we often use for benchmarks is GEN-SIM-L1-DIGI-HLT

GEN-SIM-L1-DIGI-HLT				
Release	Time/event	Perf ticks	Alloc Rate	Alloc Rate
CMSSW_1_8_4	99.0 s	9.54%	356 kHz	54.4 MB/s
CMSSW_2_1_7	132.0 s	7.98%	401 kHz	28.7 MB/s
CMSSW_3_0_X	127.4 s	8.38%	426 kHz	29.5 MB/s
CMSSW_3_1_X	112.6 s	5.64%	250 kHz	27.3 MB/s

Pythia TTbar events

The initial increase is due to changes made as part of the (physics) tuning of the simulation, see talk by F. Cossutti. Subsequent improvements come primarily from our performance work. Here we are dominated by Geant4, but we have made improvements both to our own code and to Geant4.

GEN-SIM-L1-DIGI-HLT = an application which runs the event generation, simulation, digitization simulation, L1 trigger emulation and High Level Trigger (HLT) code

Memory footprint

CMS officially asks our computing centers to provide 1GB/job-slot in order to run our applications. Keeping our applications within a 1GB memory footprint has been an ongoing battle.

The Apollo Guidance Computer had 4kB RAM, 32kB ROM and they went to the Moon.
We have 1GB and usually have no clue how we've filled it up.

Strategy: understand *precisely* how we use the memory and justify it all systematically: heap, code, BSS and how it all fits together
A tall order: we are somewhat tools-limited, and definitely "understanding" limited, so we've focused on developing these things.

Tools – Massif heap profiler

From valgrind 3.3.0 the massif heap profiler has improved significantly and is much more useful for large applications such as ours.

Snapshot 62

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
62	8,921,920,724	35,320,408	26,487,090	8,833,318	0

74.99% (26,487,090B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

Search (regexp): go clear

	bytes	%	function	library
->(-)	9,993,231	28.29%	std::string::_Rep::_S_create(unsigned, unsigned, std::allocator<char> const&)	new_allocator.h:81
| ->(-)	9,048,256	25.62%	char* std::string::_S_construct<char const*>(char const*, char const*, std::allocator<char> const&, std::forward_iterator_tag)	basic_string.tcc:150
| | ->(-)	9,048,204	25.62%	std::string::string(char const*, std::allocator<char> const&)	basic_string.h:1386
| | | ->(-)	6,726,161	19.04%	Reflex::TypeName::TypeName(char const*, Reflex::TypeBase*, std::type_info const*)	libReflex.so
| | | ->(+)	1,630,868	04.62%	Reflex::MemberBase::MemberBase(char const*, Reflex::Type const&, Reflex::TYPE, unsigned)	libReflex.so
| | | ->(+)	306,682	00.87%	Reflex::ScopeName::ScopeName(char const*, Reflex::ScopeBase*)	libReflex.so
| | | ->(+)	306,329	00.87%	Reflex::PluginFactoryMap::FillMap(std::string const&)	libReflex.so
| | | ->(-)	78,164	00.22%	*below ms_print's threshold*	in 1905 places
| | ->(+)	52	00.00%	*below ms_print's threshold*	in 1+ places
| ->(+)	943,029	02.67%	char* std::string::_S_construct<char*>(char*, char*, std::allocator<char> const&, std::forward_iterator_tag)	basic_string.tcc:150
| ->(+)	1,946	00.01%	*below ms_print's threshold*	in 1+ places
->(+)	4,926,672	13.95%	Reflex::ClassBuilderImpl::AddFunctionMember(char const*, Reflex::Type const&, void	libReflex.so
->(+)	824,064	02.33%	Reflex::FunctionTypeBuilder(Reflex::Type const&, Reflex::Type const&)	libReflex.so
->(+)	786,088	02.23%	std::vector<Reflex::Type, std::allocator<Reflex::Type> >::_M_insert_aux(__gnu_cxx::__normal_iterator<Reflex::Type*, std::vector<Reflex::Type, std::allocator<Reflex::Type> > >, Reflex::Type const&)	liblcg_RelationalStorageService.so
->(+)	763,440	02.16%	Reflex::ClassBuilderImpl::AddTypedef(Reflex::Type const&, char const*)	libReflex.so
->(+)	716,715	02.03%	*below ms_print's threshold*	in 1685 places
->(+)	631,824	01.79%	__gnu_cxx::hashtable<std::pair<std::string const* const, Reflex::TypeName*>, std::string const*, __gnu_cxx::hash<std::string const*>, std::_Select1st<std::pair<std::string const* const, Reflex::TypeName*> >, std::equal_to<std::string const*>, std::allocator<Reflex::TypeName*> >::find_or_insert(std::pair<std::string const* const, Reflex::TypeName*> const&)	libReflex.so

Massif example from a standalone main() that just loads all libs and plugins used by full reco: 25MB and 1.1M allocs just from global ctors (mostly reflex)!

(html version of massif report courtesy of G. Petrucciani)

Main problems: slow speed and the large memory overhead.

igprof – heap snapshot

New

```
Counter: MEM_LIVE

-------------------------------------------------------------------------------
Flat profile (cumulative >= 1%)

% total       Total       Calls  Function
  100.0 643'544'794   9'039'778  <spontaneous> [1]
  100.0 643'544'794   9'039'778  __libc_start_main [2]
  100.0 643'544'794   9'039'778  main [3]
  <...>
-------------------------------------------------------------------------------
Flat profile (self >= 0.01%)

% total        Self       Calls  Function
    7.88  50'720'256      81'668  std::vector<double, std::allocator<double> >::reserve(unsigned int) [64]
    4.10  26'370'848      23'557  std::vector<DDExpandedNode, std::allocator<DDExpandedNode> >::operator=(std::vector<DDExpandedNode, std::allo
    3.86  24'846'408     507'463  std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_Rep::_S_create(unsigned int, unsigne
    3.72  23'968'408      38'247  G4PhysicsVector::FillSecondDerivatives() [152]
    2.98  19'151'520      37'552  G4AllocatorPool::Grow() [166]
    2.76  17'764'404   1'268'886  G4SmartVoxelHeader::BuildNodes(G4LogicalVolume*, G4VoxelLimits, std::vector<int, std::allocator<int> > const*
    <...>
    1.94  12'480'460     601'927  std::vector<int, std::allocator<int> >::_M_insert_aux(__gnu_cxx::__normal_iterator<int*, std::vector<int, std
    <...>
    1.64  10'556'448         644  TBuffer::TBuffer(TBuffer::EMode, int) [269]
    <...>
    0.95   6'086'424      76'493  std::vector<G4SmartVoxelProxy*, std::allocator<G4SmartVoxelProxy*> >::operator=(std::vector<G4SmartVoxelProxy
    <...>

-------------------------------------------------------------------------------
Call tree profile (cumulative)

          <...>
            0.1  ..........      370'568 / 40'192'560       31'433 / 2'097'958      G4SmartVoxelHeader::BuildVoxelsWithinLimits(G4LogicalVolum
            0.5  ..........    2'914'120 / 38'959'788      227'681 / 2'039'097      G4SmartVoxelHeader::BuildVoxelsWithinLimits(G4LogicalVolum
            4.2  ..........   26'897'476 / 31'646'688    1'611'429 / 1'673'692      G4SmartVoxelHeader::BuildVoxelsWithinLimits(G4LogicalVolum
[127]       4.7   30'182'164  17'764'404 / 12'417'760    1'870'543                G4SmartVoxelHeader::BuildNodes(G4LogicalVolume*, G4VoxelLimi
            1.9  ..........   12'417'760 / 12'480'460      601'657 / 601'927        std::vector<int, std::allocator<int> >::_M_insert_aux(__gn
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Total bytes in heap and number of Allocations

An example (and abbreviated) igprof heap snapshot report. Again it contains two flatprofiles (memory allocated from each given function and all of its callees, allocated memory from each given function itself) and also a call graph.

In this particular example there are 643'544'794 bytes in the heap from 9M(!) allocations, averaging only 71 bytes/allocation.

igprof – heap snapshot

```
Counter: MEM_LIVE

-------------------------------------------------------------------------------
Flat profile (cumulative >= 1%)

% total       Total       Calls  Function
  100.0 643'544'794   9'039'778  <spontaneous> [1]
  100.0 643'544'794   9'039'778  __libc_start_main [2]
  100.0 643'544'794   9'039'778  main [3]
  <...>
-------------------------------------------------------------------------------
Flat profile (self >= 0.01%)

% total        Self       Calls  Function
   7.88  50'720'256      81'668  std::vector<double, std::allocator<double> >::reserve(unsigned int) [64]
   4.10  26'370'848      23'557  std::vector<DDExpandedNode, std::allocator<DDExpandedNode> >::operator=(std::vector<DDExpandedNode, std::allo
   3.86  24'846'408     507'463  std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_Rep::_S_create(unsigned int, unsigne
   3.72  23'968'408      38'247  G4PhysicsVector::FillSecondDerivatives() [152]
   2.98  19'151'520      37'552  G4AllocatorPool::Grow() [166]
   2.76  17'764'404   1'268'886  G4SmartVoxelHeader::BuildNodes(G4LogicalVolume*, G4VoxelLimits, std::vector<int, std::allocator<int> > const*
   <...>
   1.94  12'480'460     601'927  std::vector<int, std::allocator<int> >::_M_insert_aux(__gnu_cxx::__normal_iterator<int*, std::vector<int, std
   <...>
   1.64  10'556'448         644  TBuffer::TBuffer(TBuffer::EMode, int) [269]
   <...>
   0.95   6'086'424      76'453  std::vector<G4SmartVoxelProxy*, std::allocator<G4SmartVoxelProxy*> >::operator=(std::vector<G4SmartVoxelProxy
   <...>

-------------------------------------------------------------------------------
Call tree profile (cumulative)

          <...>
           0.1  ..........     370'568 / 40'192'560      31'433 / 2'097'958      G4SmartVoxelHeader::BuildVoxelsWithinLimits(G4LogicalVolum
           0.5  ..........   2'914'120 / 38'959'788     227'681 / 2'039'097      G4SmartVoxelHeader::BuildVoxelsWithinLimits(G4LogicalVolum
           4.2  ..........  26'897'476 / 31'646'688   1'611'429 / 1'673'692      G4SmartVoxelHeader::BuildVoxelsWithinLimits(G4LogicalVolum
[127]      4.7   30'182'164  17'764'404 / 12'417'760   1'870'543                G4SmartVoxelHeader::BuildNodes(G4LogicalVolume*, G4VoxelLimi
           1.9  ..........  12'417'760 / 12'480'460     601'657 / 601'927        std::vector<int, std::allocator<int> >::_M_insert_aux(__gn
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Heap allocations come with admin overhead and alignment loss, depending on the allocator. For the default linux (ptmalloc2) allocator, 8 bytes/alloc is a reasonable estimate. 9M alloc => 72MB!

1 - 24MB of strings, mostly from reflex

2 - A large number of (on average) small allocations means significant overhead, many of these should perhaps be in containers (e.g. they are things created at the beginning of the job, not event products)

3 - M_insert_aux for vectors implies that resizing of the vector capacity has probably happened => between 0% and 50% of the allocation isn't used

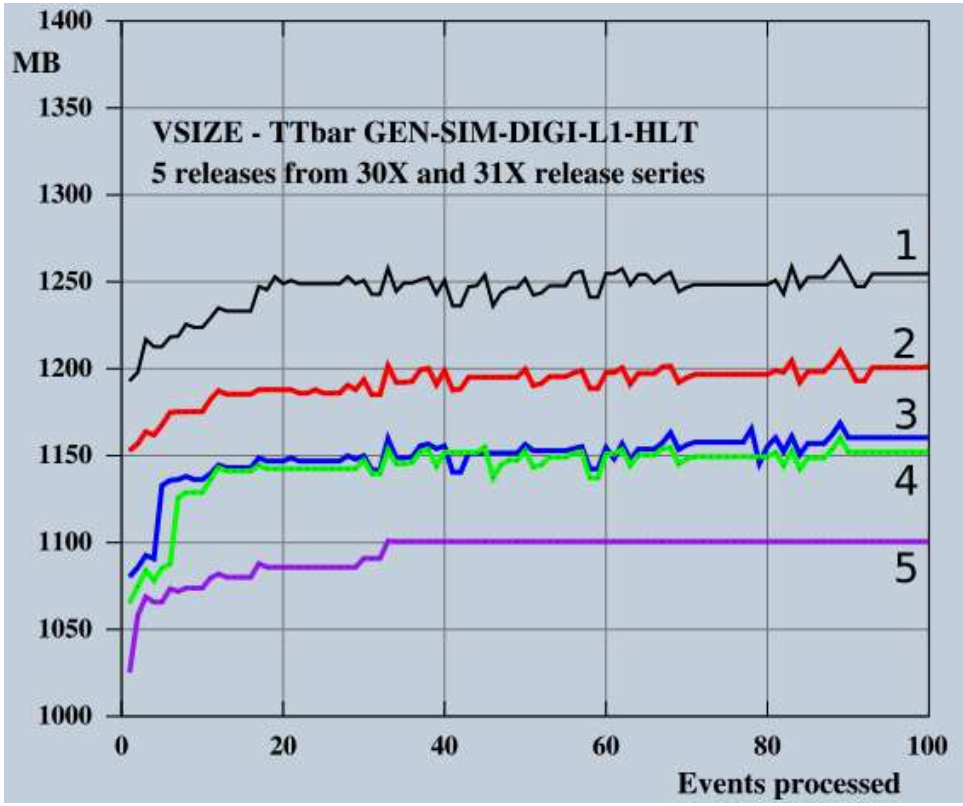
4 - ROOT I/O of course has some opportunities for optimization

Memory footprint

Through a combination of "physics algorithmic", "algorithmic, but technical" and purely technical changes we've made steady progress on reducing the memory footprint.

An example:

Our GEN-SIM-L1-DIGI-HLT application is 150MB slimmer than we were a few months ago.



GEN-SIM-DIGI-L1-HLT example (recent releases)

An additional ~100MB+ of cruft is identified since some time, but not yet removed: the default LHAPDF shared library groups together all possible PDF's *and* their common blocks, etc. Most jobs don't need most of that and the result is an unnecessarily large BSS segment.

Common memory problems

Common (technical) issues with memory management, affecting either dynamic allocations and/or the memory footprint, are:

- Confusion as to how std::vectors work, copying of large data structures
- Memory allocation in tight loops, in particular the creation, destruction and use of vectors, maps, deques, etc
- Dynamic allocation of numerous tiny objects
- Multiple in-memory copies, strings used in inappropriate places
- The use of new-d objects simply to facilitate “no object” (null pointer) as a possible answer

... and many others! The important thing is to use actual, detailed profiles and go through them very critically, asking questions!

Code Size

CMS applications use shared libraries. The libraries themselves are a large fraction of the memory footprint, e.g. 200MB in reco.

	RECO		
	Total Size	Resident Size	Non-resident
CMSSW non-DataFormats	88.5 MB	60.8 MB	27.7 MB
CMSSW DataFormats	50.7 MB	33.8 MB	16.9 MB
externals	61.1 MB	27.6 MB	33.5 MB
Total	200.4 MB	122.3 MB	78.1 MB

Observation 1: A lot of code (78.1MB) is non-resident and not used. The spuriously linked libraries have been removed, so this is just an unfortunate, but expected, consequence of how we make and use shared libraries.

Goal 1: We would like to make static binaries for several large, standard bulk-production applications. (Some preliminary work done, but not yet accomplished.)

Code Size - Dictionaries

	RECO		
	Total Size	Resident Size	Non-resident
CMSSW non-DataFormats	88.5 MB	60.8 MB	27.7 MB
CMSSW DataFormats	50.7 MB	33.8 MB	16.9 MB
externals	61.1 MB	27.6 MB	33.5 MB
Total	200.4 MB	122.3 MB	78.1 MB

Observation 2: The data dictionaries (DataFormats) account for a surprisingly large fraction of the loaded code (50.7 MB ~ 25%). In addition, global ctors from these dictionaries account for 25-30MB in the heap (from strings).

Goal 2: We have been building the full data dictionaries. Changing to building –dataonly dictionaries with genreflex gains 20MB in library size and ~15 MB in heap allocations. We still need to figure out some details of deploying both the minimal and full dictionaries to support all use cases, though.

Code Size – Bloat

	RECO		
	Total Size	Resident Size	Non-resident
CMSSW non-DataFormats	88.5 MB	60.8 MB	27.7 MB
CMSSW DataFormats	50.7 MB	33.8 MB	16.9 MB
externals	61.1 MB	27.6 MB	33.5 MB
Total	200.4 MB	122.3 MB	78.1 MB

Observation 3: Even if some of the bulk cruft described on the previous slides were to be removed, it still doesn't explain the code size we have. In addition we have (from perfmon studies) seen that the code bloat may also be playing poorly with the CPU memory hierarchy and affecting CPU performance.

Strategy: Pursue a better understanding of code bloat in general, how to identify it and in particular how it affects CPU performance. See next talk by Giulio Eulisse for more details of our code bloat investigations

64bit builds

A native 64bit build should show improved performance to additional/larger registers, reduced function call overhead, reduced -fPIC cost, etc.

We have in fact seen 5-20% improvements in some of our applications

However the transition to production use of 64bit is non-obvious because:

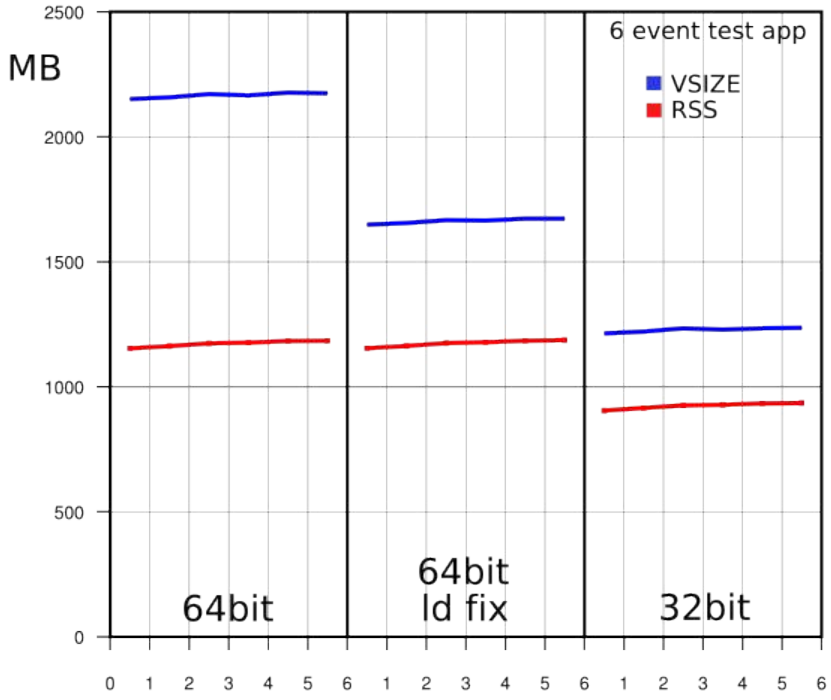
- Many sites still had 32bit CPU's and (more often) 32bit SLC4 OS deployments - may be solved this year by SLC5/64bit updates and by simple retirement of older machines
- We didn't want the complication of validating/deploying both 32bit and 64bit builds
- Applications run from the 64bit builds appear to require twice the virtual memory of the 32bit applications

Observation: the 64bit builds will be more interesting for us later this year

Strategy: understand the memory use of the 64bit integration build!

64bit vs 32bit memory footprint

- Although the VSIZE doubles for the 64bit applications, the RSS increase is much more modest (25-30%) But where does any of this increase come from?
- Most of the VSIZE/RSS difference between 32bit and 64bit comes from a default 1MB alignment of data/text pages, imposed by ld for 64bit, visible with pmap as memory segments with no permissions
- With a custom linker script, ~500MB can be removed by relinking the numerous and very small CMSSW libs (see plot). Another ~100MB would be possible from linking external libs in the same way.
- The actual code (text/data) size itself only increases by a small amount (~5%) from 32bit to 64bit (at least with gcc3.4.5)



- The heap allocation overhead/alignment cost for 64bit is twice that of 32bit, this itself implies an *extra* 40-80MB for our applications, plus the 64bit pointers themselves imply an *extra* 20-40MB (e.g. for 5-10M allocations in the heap)
- The good news is that the same things we are doing for our 32bit applications will help (twice) for 64bit

Multi-core CPU's

I've focused exclusively on the traditional, single application performance, but of course in the era of increasingly multi-core CPU's new issues arise:

- Our simple approach of running one process/core requires ever increasing amounts of (main) memory as the number of cores increases
- We risk ever increasing problems from poor use of the memory hierarchy within the CPU's.

See plenary talk by Vincenzo Innocente

Strategy 1: continued focus on understanding and improving the single application performance and (in particular) the memory utilization
Strategy 2: active involvement in the LCG R&D program on HEP software on multi-core CPU's

Misc things

There are a number of other related things that are part of the overall CMS strategy for performance improvement, but which I don't have time to cover:

- Release by release automated monitoring of performance
  - See the Release Validation talk by Oliver Gutsche and the poster on Benchmarking by Gabriele Benelli
- Specific optimizations with respect to ROOT I/O
  - See the talk by Benedikt Hegner
- Compilers (gcc 4.1, 4.3)

Summary

- CMS has been actively pursuing improvements to the performance and efficiency of our software
- We have made great progress:
  - A factor of three improvement in the reconstruction time over the past year as well as continued progress on the simulation, despite ongoing development.
  - We have kept the memory footprint close to our 1GB goal
- In addition we are steadily improving our understanding how and why we obtain the performance we see and how to improve it and have developed tools, such as igprof, to monitor it in detail

Tools used - standard

- Valgrind/callgrind/kcachegrind
  - well known tools, plus a CMS framework service that can be used to limit valgrind instrumentation to interesting sections of the coders
  - precise/reproducible results for before/after comparison
  - reasonably fast when profiling a limited piece of code, but slow and memory intensive when profiling entire applications
- Valgrind memcheck/massif - see later slide
- perfmon - profiling using hardware performance counters
  - Learning how to use this (see talk by G.Eulisse)
  - For the future: libpfm and selective instrumentation
- Misc tools: nm, size, process pmap/smap, malloc_stats/mallinfo

Tools developed by CMS

- framework "services" - report per-module time, memory, etc.
- igprof
  - fast, low overhead, no instrumentation of code required
  - sampling performance profiling
    - overhead: negligible for speed, tens of MB in memory footprint
  - accurate memory profiling
    - provides information on dynamic memory use, leaks and heap profile
    - overhead: 50% for speed, o(200MB) for memory footprint
  - recent developments include a hook for triggering heap profiles on demand (e.g. from a framework service)

Software Development Model

Our software development model shapes in part how we pursue better performance:

- relatively large number of developers, the codebase grows relatively quickly
- most people developing software don't necessarily consider themselves "software developers" as such
- Software releases support both sandbox style development *and* production application building via python configs
- Releases are divided into >1100 packages, each with its own shared libraries/plugins