Table of Contents

10.3.1 Common Performance Issues

Complete:

Introduction

The following are issues we have run into most commonly, in no particular order of significance.

- Over-reliance on strings or mis-using std::string. Large sections of CMS software are performance-limited by the reliance on strings, not by the numerical algorithms.
- Recomputing values unnecessarily. This most often occurs in the form of "x->y().z().w()", where "y" and "z" may not be as cheap as the developer assumed, or become excessively expensive when continuously recomputed for various reasons. The recomputation may be in a loop, but frequently when code has been split to dozens of small functions, it simply occurs because each small function gets the value again.
- Looking up values in maps several times in a row. One possibility is to use "map.find(key)" instead of "map[key]", and then use the iterator returned. Another possibility is to save the reference returned by "map[key]" and reuse it several times.
- Excessive sorting. Do not sort containers to find the smallest or largest element or the average value. Avoid sorting containers of large objects if a permutation vector can be used instead. Avoid sorting containers several times, especially on every insert. Be very careful with any sort involving floating point numbers and in particular where the sort key is computed, not taken directly from memory.
- Spending time to compute values that are not used at all. If there are significant asymmetries in the usage patterns, make sure there is a "fast path" through the code for the most common ones.
- Copying large objects, either passing too large objects by value, passing objects with significant memory allocations by value, or too frequently copying large object structures. Specifically using CLHEP dynamically sized vectors or matrices: HepMatrix, HepVector and related classes. Use ROOT::Math instead where possible.
- Random access reading from compressed ROOT files, for example to sample from a profile. One alternative is to use several data files and choose randomly among them, then read in each file linearly. Another alternative is not to compress the data files. A third alternative is to pre-read all the data in memory.
- Using compression inappropriately. Some compression algorithms differ significantly from the trade-off ZLIB makes on speed and compression ratio.
- Using magnetic field in unoptimal ways. The field is fast but not for free, avoid overusing it. It takes significant expertise to know when the lookups can be optimised, so consult experts for advise if necessary.

Common reasons for memory performance problems

We address here two major categories of memory performance: memory leaks and poor performance due to memory allocation or access patterns.

Memory leaks

The number one reason for memory leaks in CMS software is unclear object ownership and life time policy: who owns the object and is responsible for deletion, who may create references to the objects and when do references become invalid. Usually the code mixes smart pointers, reference counted objects, deep copies and passing by reference.

Above all, stick with maximally clear object life time policy that is manifestly clear to anyone reading the code *and in particular at interface hand-over points*. Developers reading code will in general make two assumptions: objects allocated on stack or passed by reference have "local" life time, and objects allocated on the heap with "new" have "extended" life time. Be particularly mindful about all points where developers will be passing objects to or from someone else's code.

Thus, never take and keep the address of an object passed to you by reference by some function or method. Nobody reading the code will know to expect that someone is now holding a reference to the object they think is temporary. Conversely, if the object *is* a temporary, in general allocate it on stack, not on the heap. If you allocate the object on the heap, anyone reading the code will most likely assume other objects will start taking references to that object.

Common idioms leading to leaks

- ESSource does not free the data it produced. The objects put into the object store are freed but need proper destructors, including everything contained within. Everything else must be destroyed by the source itself.
- Placing object reclamation on the "wrong" edge of the systeme state change. For example, clearing data structures on "new event" rather than "end of event". This can lead to confusing leaks at the end of the run, and potentially between runs. This is most likely lack of good examples and training, and possibly lack of automation at the framework level.
- Allocating an object on heap and forgetting to delete it. Allocate local objects on stack unless you absolutely have to allocate them on heap. If you have to allocate it on heap, use a smart pointer such as "std::auto_ptr" or one from boost so you don't need to remember to delete them.
- Having a vector of pointers and forgetting to delete the vector contents, not just the vector itself.
- Allocating an object and handing it over to someone else, who can't know whether they can or can not, or should or should not delete it, and how long they may maintain a pointer to that object. The only fix to this is to define clear object life time policy and to automate that policy to maximum possible extent.
- Being handed a cloned object and forgetting to delete it. Various algorithms seem to clone a deep structure to capture a state, and most callers seem not to have known they are in charge of freeing those objects. This seems to indicate the life time model is poorly enforced and smart pointers should be considered, or the interface needs to be remodeled not to create the copies, the method names need to be changed to give a better indication of what is happening, the client documentation is poor, or something else in the same vein. The main issue here is that for some reason it not obvious to the

programmers what they should do, or that it hasn't been automated for them.

Memory allocation and access patterns

By far the biggest factor affecting the performance of CMS software is allocating memory too frequently. On average there are nearly 800'000 memory allocations and deallocations each per second.

This comes up in numerous different contexts, but there are currently some very large offenders: CLHEP dynamic matrix and vector classes (HepMatrix etc.), and std::strings. Very large portions of the code creates short-lived, usually by-value matrix, vector and string objects, leading to massive memory churn.

The strings churn is frequently "accidental" in that C string literals are unexpectedly converted to a "std::string" then thrown away moments later. These are usually lookups where strings are used as a key, or passing arguments (for example to the message logger), and comparisons against "known" values. The vast majority of these can not only be removed entirely but prevented from happening again with better designs; please contact experts for guidance.

Another fairly frequent idiom is constructing large vectors one element at a time without calling "reserve()" or passing large vectors by value.

Some parts of the code would most likely benefit from specialised allocators, in particular pool allocators. This needs to be evaluated case by case.

Beyond the above there is another entirely different layer of memory performance concerns, especially due to severe access cost hierarchies of modern CPUs. This layer is currently entirely overshadowed by overwhelming issues at much higher levels and is therefore not relevant for optimisation at this stage. If you have code mature enough to proceed to this stage, please contact the experts.

Review status

Reviewer/Editor and Date	Comments
LassiTuura - 17 Apr 2007	created page
JennyWilliams - 29 Jun 2007	general tidying for inclusion in printable workbook

Responsible: PeterElmer
Last reviewed by: Reviewer

This topic: CMSPublic > WorkBookPerformanceCommonIssues
Topic revision: r11 - 2010-01-18 - KatiLassilaPerini