Updates on C++ Modules (PCMs) in ROOT

Vassil Vassilev

16.03.2016

Vassil's work is entirely sponsored by USCMS and FNAL.

ROOT 6

- “Drop-in” replacement for ROOT 5

- Excessive memory use wrt ROOT 5

  True C++ interpreter comes at certain cost - it needs to know everything about your library in order to use it.

ROOT's Library Loading

```
// MyExternalS.h
struct MyExternalS {
  // members
};
//…

MyExternalS* gS = new MyExternalS();
```

```
// Utils.h
#include "MyExternalS.h"
void do (MyExternalS* S = 0) {
  if (S = 0)
     S = gS; S->f();
  // ...
}
// ...
```

```
...
```

=> libExternal.so

CMSSW has a lots of semantical equivalents to this.

```
root[0] gSystem->Load("libExternal");
root[1] do(); // must work, w/o #include
```

Implicit #includes must be lazy, at the time of use. C++ Modules are exactly designed for this.

C++ Modules Adoption

Adoption plan in ROOT:

1. Use the feature the way it was designed
   Compile project's codebase with -fmodules and recent clang

2. Use the feature to optimize ROOT's runtime
   Provide rootcling support to build pcms and teach ROOT how to use them

3. Once the feature is implemented it could be picked up by CMS with relatively little (but > 0) amount of work

See my ROOT workshop talk: http://indico.cern.ch/event/349459/session/1/contribution/19
See my Weekly offline meeting report: https://indico.cern.ch/event/461005/session/2/contribution/13/attachments/1188722/1724740/CMSSW_Core_Software_Cxx_Modules.pdf

Progress on 1.

Compile ROOT with -fmodules switch:

- Implemented build system support
    - `cmake -Dcxxmodules=On` or `configure --enable-cxxmodules`
- ROOT builds and works on MacOS
- Fixed a crucial bug in clang: https://llvm.org/bugs/show_bug.cgi?id=24954
- Few other non-critical issues:
    - The first one is conceptual: we need to precompile glew.h, avoiding to turn on the implicit module maps for OpenGL on macos.
    - The second is a bug in clang's codegen in merging definitions from roofit, it is an assertion, which IMO is harmless, but still needs fixing (https://llvm.org/bugs/show_bug.cgi?id=25501).
    - The third is a 'too many open files' error, coming from MacOS. This might be a bug in clang: it seems to keep open the header files and the PCMs. Even worse the PCMs are opened twice. This hits the funny limitation on MacOS, because of the two many opened files per process. The issue is not a blocker, since we could build one top-level module.

Progress on 1. (contd)

Compile ROOT with -fmodules switch:

- ROOT **should** build and work on Unix
  - cmake -Dcxxmodules=On -Dlibcxx=On or configure --enable-cxxmodules --enable-libcxx
  - Doesn't work with libstdc++ yet (see next slide)
- Patches in clang under review, takes a lot of time, because there are only 1-2 people in the community who can review this code.
- Total of ~90 checkins in ROOT improving 360 files and **a lot of** time spent debugging and fixing clang.

Progress on 1.

Compile ROOT with -fmodules switch:

- Discovered a few bugs in ROOT itself
- class TNDArrayT : public TNDArray { … ClassDef(TNDArray); }; // must be ClassDef(TNDArray**T**)
- ClassDefs of templated classes lead to non-conforming dictionaries:
  - *A.h*: template<class T> struct TMyClassT { ...; ClassDef(MyClassT, 1); };
    B.h: struct S { TMyClassT<int> var; };
    G__: #include "A.h" // #1
    #include "B.h" // #2
    // Expanded contents of the ClassDef. E.g. template <> TClass *TMyClassT<int>::Class(){...}
  - B.h forces an implicit instantiation of TMyClassT **before** the compiler can see the definitions of the ClassDef. Namely, template <> TClass *TMyClassT<int>::Class(){} is seen after #2 when the compiler has instantiated it. C++ 14.7.3/6 explains it as: "If a template, a member template or a member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; **no diagnostic is required**." Saying that what we do makes the TU ill-formed but no diagnostic is required.

    Modules are more strict in that respect and issue diagnostics. We need to implement a ClassDef macro which inlines the contents, disallowing custom streamers covering 99.9% of the cases.

ToDo on 1.

Compile ROOT with -fmodules switch:

- ROOT on Unix
  - Extra efforts needed to support libstdc++
    libstdc++ doesn’t provide module map file, thus it stress tests the module merger even more.

- Patches in clang under review, takes a lot of time, because there is only 1-2 people in the community who can review.

Progress on 2.

Use the feature to optimize ROOT's runtime:

- `rootcling` has to be taught to generate the PCMs
- Requires a newer version of LLVM in ROOT
  - The ROOT team is planning to upgrade "soon"

Performance

Bjarne Stroustrup, creator of C++, at the ROOT 2015 workshop: “I’ve seen cases where C++ modules bring 100x compilation performance improvements.”

Other C++ and C++ modules experts confirm these facts in private exchanges.

Performance

Experts about C++ Modules:

- r261161: “[modules] … This results in a 35x speedup on one of our internal build steps (2m -> 3.5s),but is hard to unit test because it requires a very large number of modules.”
- r255384: “[modules] … This typically reduces PCM file size by about 1%.”
- r246680: “[modules] Don’t waste time reading in the names the module file writer gave to blocks…”
- r246497: “[modules] … This gives a huge performance improvement to builds with very large numbers of modules (in some cases, more than a 2x speedup was observed)”
- r245779: “[modules] … On one testcase with several thousand imported module files, this reduces the total runtime by 72%.”
- …

In many of those cases speedups translate directly into less memory allocations and less memory usage because of the intricacies of the underlying implementation.

Preliminary Performance Results

Due to the not-yet-complete status of (2), I've created a close-to-reality test case based on standalone clang.

```
// T.cpp
#include "THtml.h"
#include "TTree.h"
#include "TLorentzVector.h"

THtml h;
TTree t;
TLorentzVector l;
```

The -fmodules compilation yields 39% peak memory decrease.

This is backed up by compiler's reports of much less memory allocations.

Thank you!

Questions?

FAQ

Why ROOT needs modules?

- Modules are proven to reduce compilation and memory usage.

Why don’t we wait for somebody else to implement them?

- This was the case last couple of years and it didn’t work. We have specific use-cases, which hit a few corner cases. Extracting a minimalistic reproducer is often the most difficult part, not the fix itself. The fixes for our use cases are not a priority for clang’s community.

If we use C++ modules, are we bound to a certain compiler vendor/version?

- No, in a production ready environment PCMs will be provided by rootcling.

FAQ

Why we need to include every header, on which we depend?

- Indirect includes are a bad practice. The non-module builds work because of the specific include order (aka voodoo magic). The compiler is extremely helpful in telling what is missing.

What is the size of the PCMs on disk?

- Hard to tell, because they are very configurable. Worst case scenario for ROOT (a lot of duplications in PCM's content) configure-make has 2441 PCMs ~ 700M in total (cmake has a bug somewhere and generates 9477 PCMs ~ 2.8GB in total).

Are there performance numbers?

- Not yet, because they depend on completing stage 2 (slide 4). Preliminary results show 39% memory decrease in close-to-reality scenario.