Optimizing ROOT's Performance Using C++ Modules

Dr. Vassil Vassilev
(presented by Philippe Canal)
10.10.2016

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

C++ Modules

Main goal is to enable scalable compilation of C++ code.

See Modules TS (n4592): http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4592.pdf

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.

C++ Modules in ROOT

Use the feature to reduce ROOT's memory usage and speed up interaction with ROOT's interpreter.

C++ Compilation Model

Translation Units

```cpp
// A.cpp

int pow2(int x) {
  return x*x;
}
```

```cpp
// B.cpp
extern int pow2 (int x);
int main() {
  return pow(42);
}
```

Each translation unit (TU) is independent. The communication problem is resolved via *name linkage*.

Linking And Compiler Copy-Paste

Translation Units

```
// A.h
// #include <string> expands to ~19290 LOC
int pow2(int x);
```

```
// A.cpp
#include “A.h”
//expanded textually to
//int pow2(int x);
int pow2(int x) {
  return x*x;
}
```

```
// B.cpp
#include “A.h”
//expanded textually to
//int pow2(int x);
int main() {
  return pow(42);
}
```

TU communication is done via external names. In order to minimize the errors header files are introduced.

Translation Units to Module Units

```
// A.h
// #include <string> expands to ~19290 LOC.
int pow2(int x);
```

```
// A.h module interface, aka modulemap.
module A {
  header A.h
  export * // The compiler exports pow2 as part of module A.
}
```

```
// A.cpp
import A.h // Uses the precompiled module file.
// import <string> doesn't recompile ~20K LOC over and over.
int pow2(int x) {
  return x*x;
}
```

C++ Modules Help Finding Issues in Your Code

The compilation units can reason about the code more accurately in some cases, issuing useful errors, warnings and hints.

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.
   For the cost of having to deal with (currently harmless) inconsistencies in your code, you will get 20% speed up

Compile ROOT with C++ Modules

- Grab latest clang (from trunk)
- `cmake -Dcxxmodules=On -DCMAKE_CXX_COMPILER=...`

  `configure --enable-cxxmodules --with-cxx=...`

Optimize ROOT's Runtime

User/Experiments' code has a lot of semantical equivalents to this.

```
// ROOT prompt (no C++ Modules):
gSystem->Load("MyLib");
// => dlopen("MyLib.so");
//    => cling->parse("1000s_of_fwd_decls.h");
MyLibClass<float> c; c.do();
// => cling->parse("#include <MyClass.h>");
```

Forces ROOT's interpreter to parse headers related to MyLib (even when we intend to use only tiny fraction of them).

This results in increased memory use and slowdown.

```
// ROOT prompt (no C++ Modules):
gSystem->Load("MyLib");
// => dlopen("MyLib.so");
//    => cling->mmap("MyLib.so.pcm");
MyLibClass<float> c; c.do();
```

C++ Modules-aware ROOT runtime will lazily allocate memory only for what you use and at the point of use!

Everything unused is mmaped.

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.
Using prebuilt modules reduces the compilation time by 21%.

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

Towards Stability

Ensuring implementation reliability:

- Bi-weekly meeting with Google engineers to collaborate on the development of the feature. HEP-specific requirements are represented by Vassil.
- LLVM modules self host buildbots
  - http://lab.llvm.org:8011/buildslaves/modules-slave-1
  - http://lab.llvm.org:8011/buildslaves/modules-slave-2 (requested by my LLVM GSoC student and I), thanks to Richard Smith it is up and running on Google Compute Engine. More to be added soon.
  - Based on both configure and cmake, running respectively roottest and ctest. Discovered already a few regressions in clang.
- Vassil was granted commit access everywhere in LLVM repositories.
- Vassil became a Bulgarian representative in the ISOCpp and he plans to participate actively in the standardization process of the C++ Modules feature.

In Summary

- Still a lot of work ahead in clang, cling, ROOT and CMSSW.
- The work on the C++ Modules helped ROOT improve its codebase:
  - Found a few issues some of them pretty severe (e.g. ill-formed dictionaries).
  - Removed a few bad practices (e.g. #define protected public).
  - Revisit long forgotten code, which is was not tested and not working (e.g. ROOT exception support, TException).
  - Cleanup redundant or cyclic header dependencies (e.g. TMVA).

In Summary

- Vassil would like to thank USCMS for funding this essential for ROOT piece of work.
- Vassil would like to thank Richard Smith from Google who helps a lot and provides a lot of wisdom.

Thank you!

Questions?

FAQ

Why does ROOT need modules?

- Modules are proven to reduce compilation and memory usage. It reduces ROOT's compile time on average by 21%.

Why we don't 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 do 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) has 2441 PCMs ~ 700M in total.

Are there performance numbers?

- Not yet, because they depend on completing stage 2 (slide 9). Preliminary results show 39% memory decrease in close-to-reality scenario.

FAQ

Does the C++ modules implementation have bugs:

Yes, and if you discover one please contact me.

Why we should build our software stack with modules?

- This will make the transition to ROOT's C++ Modules-aware environment easier and smoother.
- Gives different view of the code, exposing more bugs. (Analogously, why do we build with more than one compiler).
- Speeds up the builds (for ROOT by 20-30%)

Backup Slides

…

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();
}
// ...
```

...

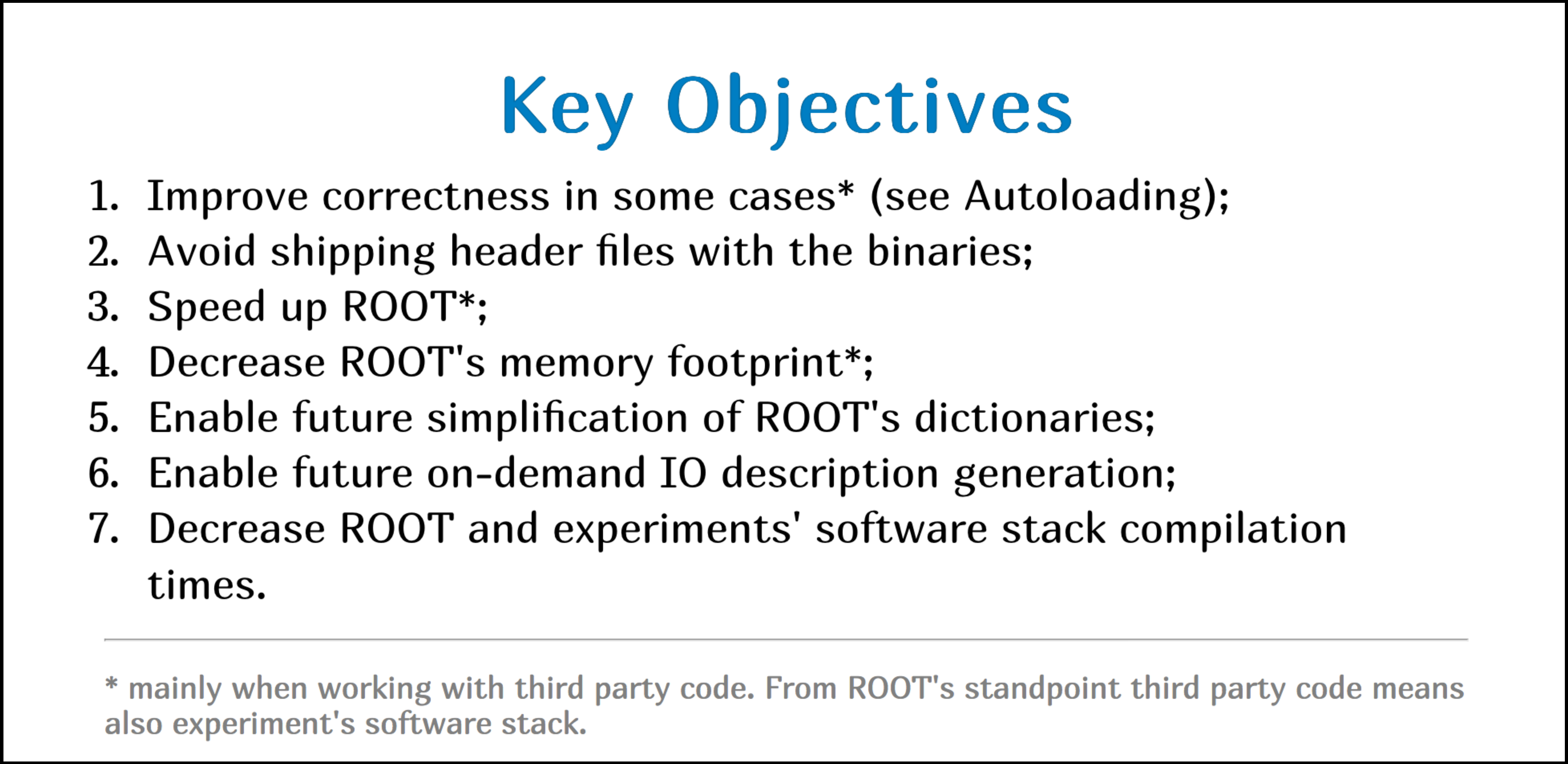
=> libExternal.so

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.

Initial Key Objectives



Key Objectives

1. Improve correctness in some cases* (see Autoloading);
2. Avoid shipping header files with the binaries;
3. Speed up ROOT*;
4. Decrease ROOT's memory footprint*;
5. Enable future simplification of ROOT's dictionaries;
6. Enable future on-demand IO description generation;
7. Decrease ROOT and experiments' software stack compilation times.

* mainly when working with third party code. From ROOT's standpoint third party code means also experiment's software stack.

Finding Bugs with Modules

C++ Modules compilation discovered problems:

- Discovered a few bugs in ROOT itself. E.g.

```
class TNDArrayT : public TNDArray { /*...*/ ClassDef(TNDArray); }; // must be
ClassDef(TNDArrayT)
```

- 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.