Optimizing ROOT’s Performance Using C++ Modules

Vassil Vassilev
CERN, EP-SFT, Geneva, Switzerland
FermiLab, P.O. Box 500 Batavia, IL, USA
E-mail: vvasilev@cern.ch

Abstract.
ROOT comes with a C++ compliant interpreter cling. Cling needs to understand the content of the libraries in order to interact with them. Exposing the full shared library descriptors to the interpreter at runtime translates into increased memory footprint. ROOT’s exploratory programming concepts allow implicit and explicit runtime shared library loading. It requires the interpreter to load the library descriptor. Re-parsing of descriptors’ content has a noticeable effect on the runtime performance. Present state-of-art lazy parsing technique brings the runtime performance to reasonable levels but proves to be fragile and can introduce correctness issues. An elegant solution is to load information from the descriptor lazily and in a non-recursive way.

The LLVM community advances its C++ Modules technology providing an io-efficient, on-disk representation capable to reduce build times and peak memory usage. The feature is standardized as a C++ technical specification. C++ Modules are a flexible concept, which can be employed to match CMS and other experiments’ requirement for ROOT: to optimize both runtime memory usage and performance. Cling technically “inherits” the feature, however tweaking it to ROOT scale and beyond is a complex endeavor. The paper discusses the status of the C++ Modules in the context of ROOT, supported by few preliminary performance results. It shows a step-by-step migration plan and describes potential challenges which could appear.

1. Introduction
ROOT’s 6th version introduces a set of modern tools in its core. The LLVM-based C++ interpreter cling is one of them. It increases the interoperability between ROOT’s two different worlds: interpretative and compiled. Cling enables smoother transition between interpreted and compiled code because it uses LLVM’s compiler, clang to blur the boundaries between both worlds. A widely-used extension mechanism are the shared libraries. A shared library contains reusable object code in form of (name mangled) symbols. Library’s object code is lowered from high-level programming language such as C/C++. The object code cannot interoperate with third-party code without a shared library descriptor. The descriptor is used by compilers to generate compatible object code and mend both sides together. Precise description format is C/C++ header files which define the shared library layout. The third-party code imports the description (by including a set of header files) which makes the content from the library available.

The explicit loading of a shared library in ROOT must read the library’s descriptor and open the library. The descriptors are parsed at runtime causing significant performance degradation when they contain more than a few hundred header files. The well-known issue of build scalability for compiled programs grows into a performance issue for ROOT’s runtime.
On a use of a unknown entity, ROOT loads implicitly the shared library defining the entity. ROOT must read the shared library’s descriptor and open the shared library transparently to the user. This element requires each library to maintain a catalog of the entities it provides. A catalog contains library’s lightweight descriptor and extra library mapping information. The catalogs are created from the full descriptor and are registered at ROOT’s startup.

Often, the user needs a small fraction of the descriptor. ROOT avoids loading the full descriptor if a lightweight descriptor of the same library was read. The lightweight descriptor is produced by an indexer (usually at build time) and stored into a file with a rootmap extension. The file contains a set of forward declarations of the entities in the descriptor. Loading a shared library parses only the lightweight descriptor. If more information about an entity is required the lightweight descriptor reads chunks of the full descriptor. More information about an entity can be requested at any time and point in user code. In order to implement this, ROOT’s interpreter stores the current parsing state before the request and restarts parsing the chunk of the full descriptor. The recursive parsing mode is fragile and error-prone because it is not a well-defined language feature. The lightweight descriptor reduces the memory footprint when the content of the library is sparsely used. In cases of heavy use of the library the read information is the sum of the size of the lightweight and the full descriptors, i.e. the reduction turns into a penalty. Moreover, parsing the full descriptor causes a noticeable slowdown.

Relying on C++ Modules technology as a library descriptor addresses the above-mentioned shortcomings. The technology deserializes in memory only a minimal set of pre-parsed entities which eliminates the parsing and works well not only for sparse library uses. There is no need of extra forward declarations and recursive parsing. Implicit and explicit loading of shared library happens with a minimal overhead of used resources.

This paper is structured into five sections: section 2, Background, makes a brief overview of the C++ Modules technology and its implementation in the Clang compiler; section 3, Improving ROOT’s Runtime Performance Step by Step, proposes an incremental adoption strategy; section 4, Preliminary Performance Results, presents some direct and indirect benefits of using the feature; section 5, Conclusion & Future Work, concludes and outlines a future work plan.

2. Background

Initial design goal of C++ Modules feature is to enable scalable compilation for C/C++ code. As noted in [1], the feature targets improving the compilation model inherited from the C language. C compilation model introduces a notion of independent compilation. A program can consist of multiple independent compilations, translation units. Each translation unit is translated independently on the rest, with no knowledge of any details of the way it is used in the program. The communication between translation units is done via name linkage. A translation unit can reference, by name, entities defined elsewhere by qualifying them as external (Listing 1).

```cpp
// A.cpp
int pow2(int x) {
    return x * x;
}

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

Listing 1: A.cpp defines pow2 and B.cpp resolves pow2 via name linkage.

linker resolves the communication problem between translation units. This brittle low level technology is the backbone of the C/C++ compilation model.

A common organization practice of C/C++ codebases is to declare names in header files. This can minimize the errors and give an illusion for uniform view of the declared entities in
a program. From compilers’ point of view, however, those files have to be textually expanded for each translation unit which includes them. This simple concept, served well for C/C++ for decades, has a few practical drawbacks. The textual expansion of invariant header files in the including translation unit causes significant increase of compile times and memory usage. The tendency to move more and more content in the header files underlines the scalability weakness of the concept. Another well-known problem is that header files are vulnerable to macro definitions. Depending on the actual macro definition the content of the header file can vary drastically introducing unintended inconsistencies between the including translation units.

2.1. From Translation Units to Module Units

The technical specification (TS), [1], proposes modification of the C++ standard which lays out the behavior of a scalable compilation system. It defines concepts such as module units, modules, module export declaration and module import declaration. A module unit is a translation unit that contains a module declaration. A module is a collection of module units. A module export declaration splits a module unit in an interface and implementation. The module interface is a combination of one or more entities annotated with an export keyword. An import declaration makes visible the module interface or parts of it to the current translation unit. The modules

```cpp
import std.io; // make names from std.io available
module M; // declare module M
export module std.random; // import and export names from std.random
export struct Point { int x; int y; }; // define and export Point
```

Listing 2: Export and import of entities.

TS allows the implementation to differentiate which parts of a translation unit are intended for public use and which are private. The proposed syntax unties the translation unit from the preprocessor-based textual expansions which copy and paste invariant code repeatedly. It gives the compiler vendors a way to minimize the recompilations of invariant code.

A common implementation choice is creating an on-disk version of a collection of invariant code (a module file). The module file is in a pre-digested form and contains internal compiler representation of already parsed code. This is very similar to the pre-compiled header (PCH) files. The PCH is usually a single file attached before parsing. Tiered PCHs are produced very carefully to avoid duplication of code. The distinguishable difference is that module files allow potentially duplicate content to exist. For instance, line 3 from Listing 2 imports a module std.random and exports it making its contents available in module M. Re-exporting modules or importing two modules with (partially) overlapping content requires the compiler to support merging of the (semantically) same entities present in several simultaneously available modules. This, in turn, requires rethinking of the cornerstone of C++, the one definition rule (ODR).

2.2. Modules Implementation in Clang

An implementation of the modules concepts exists in the LLVM frontend clang [2], [3]. Clang supports the Modules TS and hosts modules research and development work. The implementation encourages incremental, bottom-up [4] adoption of the modules feature. Modules in Clang are designed to work for C, C++, ObjectiveC, ObjectiveC++ and Swift [5], [6]. Users can enable the modules feature without modifications in header files. The LLVM compiler allows users to specify module interfaces in dedicated file, called module maps files. A module map file expresses the mapping between a module file and a collection of header files. If the compiler finds such file in the include paths it automatically generates, imports and uses
module files. The module map files can be mounted using the compiler’s virtual file system overlay mechanism to non-writable production library installations.

In practice, a non-invasive modularization of the example in Listing 1 can be done easily by introducing a module map file (Listing 3). In a number of cases the module map files can be automatically generated if the build system knows about the list of header files in every package.

### 3. Improving ROOT’s Runtime Performance Step by Step

Usually, for convenience, users preload a set of shared libraries at startup time. The implementation must ensure interoperability with the libraries and should read their descriptors. The lightweight library descriptors are invented to reduce the significant increase of the startup time and the memory bloats. As mentioned in section 1, this has a few limitations. For instance, requesting single small entity per large header file or requesting all entities of the shared library.

```c++
// A.h
#include <string>
#include <vector>
template <class T, class U = int> struct AStruct {
    inline void doIt() { /*...*/ }
    std::string Name;
    std::vector<U> Collection;
    // ...
};
template<class T, class U = AStruct<T>>
void freeFunction() { /* ... */ }
void do(unsigned N = 1) { /* ... */ }

// Main.cpp
#include "A.h"
int main() {
    do();
    return 0;
}
```

Listing 4: A.h, part of the descriptor of libA, expands to more than 26000 lines of code.

The full descriptor can contain thousands of files expanding to millions of lines of code (Listing 4). The compilation of the translation unit in Listing 5 requires processing of more than 26000 lines of code. In this pathological but not uncommon example the variant code is approximately about 0.0001%.
ROOT’s exploratory programming concept adds an extra requirement with respect to loading of shared libraries. It lifts the dependency resolution and linking burden from the users and encourages them to focus on the core functionality of their code. At runtime ROOT supports explicit and implicit loading of shared libraries (Listing 6). Explicit loading (line 5, Listing 6)

```c++
// ROOT
1 root [] AStruct<float> S0; // implicit loading of libA. Full descriptor required.
2 root []* AStruct<float> * S1; // implicit loading of libA. No full descriptor required.
3 root [] if (gFile) S1->doIt(); // implicit loading of libA. Full descriptor required.
4 root [] gSystem->Load("libA"); // explicit loading of libA. No full descriptor required.
5 root [] do(); // error: implicit loading of libA is currently unsupported.
```

Listing 6: ROOT supports implicit and explicit loading of libraries.

requests libA and its contents to be made available. In practice, this means loading of the shared library in memory and exposing its descriptor to the interpreter. Implicit loading (line 2, 3 and 4, Listing 6) requests libA indirectly by using its contents. The user should be able to request an entity which is defined in a library descriptor of a not-yet-loaded library.

Technically, a straightforward implementation needs parsing of the shared library descriptors. This doesn’t scale very well. There are a few more scalable approaches to trim the amount of unnecessary parsing. Minimizing the size of the library descriptors by providing a smaller descriptor of the library full descriptor. A lightweight descriptor is a catalog containing all forward-declarable entities of a descriptor. If the user code requires a complete definition, the lightweight descriptor loads parts of the descriptor. For example, line 2, Listing 6 requires a complete definition of AStruct, loads libA and reads A.h. Conversely, line 3 does not require a definition, loads libA and uses only the forward declaration from the lightweight descriptor.

The lightweight descriptors are powerful but limited. For instance, default arguments cannot be redeclared (section 8.3.6.4 of [7]). Making line 2 and line 4 of Listing 6 work require modifying the interpreter in a non C++ conforming way. Implicit loading on unknown free functions is impossible (line 6). A severe source of issues, is the recursiveness of the descriptor parsing. The descriptor of libA can be requested at any position during parsing (line 4). This expects storing interpreter’s parse state, escaping to the global scope, parsing the descriptor and restoring.

A scalable implementation of implicit and explicit shared library loading leans on minimal parsing of only the essential parts of the descriptor. The C++ Modules offer scalable compilation by truncating the re-parsing of invariant code. They are lazily loaded and serve as an on-demand external source of information. Clang parses header files once and stores them in an io-efficient file format, called precompiled module. It fulfills the requirements to become a robust replacement for both lightweight and full library descriptors. Considering the fact that the technology is under development, we find appropriate the gradual adoption approach. It implies to use the feature to improve build performance first. Later, after tuning, to use the feature as a replacement for the shared library descriptors and at last to replace the ROOT dictionaries.

3.1. Building ROOT with Clang C++ Modules
There are a few sources, describing in details the technical work to enable clang C++ modules in [2], [4], [3] and [8]. A few of them (notably [4]) report improvements in the build performance and better scalability. Clang C++ Modules can be configured from containing only single headers to containing all headers of a library. The offered flexibility help tweaking the module files content to match topology of the particular project. The closer the match is, the better performance is observed. For example, some libraries have disjoint header files and it is better to represent
them as one header per module or submodule. Others, have header files designed to be included always together and it is better to create a single module file for them. The C++ Modules builds of ROOT explored a single module per header and single module per library setup. During the process we encountered a number of recurring issues with:

- Non-standalone header files – module files are produced by parsing the relevant headers in isolation. The process fails if the set of headers in the module map do not contain all necessary includes or forward declarations. Fixing the issue required approximately 1000 lines of mechanical changes in ROOT.
- Include directives in extern "C" contexts – some C libraries are not designed to be included in C++ translation units. An easy solution is to enclose the include directive in such extern context. It doesn’t work with C++ Modules because the system does not know if it builds a C or a C++ module. Usually, the build systems have information if headers are C style. This issue was resolved by approximately 10 lines of semi-mechanical changes in ROOT.
- Configuration macros – some headers are designed to be configured “outside” by macro definitions. These headers require specific handling because the system disallows by default the module files to be mutated by external macro definitions. This required approximately 100 lines of non-trivial changes in ROOT.

The experimental ROOT builds with Clang C++ Modules are stabilizing. Several nightly builds were introduced in order to ensure that no regressions appear. Currently, the builds are used as a testbed for some of the results presented in section 4.

### 3.2. Optimizing Shared Library Descriptors

The build setup of ROOT aids producing modules on demand even at runtime. The module map file describes the relationship between ROOT’s header files and shared libraries. ROOT’s dictionary generator and indexer, `rootcling`, can produce module files in compiler- and platform-independent way. Building modules for third-party libraries is done by an experimental implementation in rootcling. It uses the compiler as a library and queries its API to create module files in place of third-library’s full and lightweight descriptors.

When ROOT starts it looks for the module files and registers them as a source of external information. The system makes only the minimal set of content available in memory on implicit or explicit library loading. The experimental implementation is open to incremental replacement of the shared library descriptors. Namely, if there is a module file ROOT supersedes the default library descriptors and uses the information in the module files. This ensures gradual transition and in case of problems the implementation can fall back to the working case.

### 4. Preliminary Performance Results

The performance measurements are done on a virtual machine with 7 concurrent jobs and 6 GB of RAM. The machine has unix-based environment, libstdc++ version 6.3.1 and latest clang compiler, built from source code. All measurements are done with pre-built module files. The reported results are illustrative and classified as preliminary because not enough time has been spent in optimizations. For instance, during compilation clang generates duplicate module files. The duplicates can be reduced by removing redundant macro definitions from the source code.

Figure 1, shows details about compilation times (1a) and average peak memory (1b) usage per ROOT component of a full ROOT build. On average the compilation times decrease by 40% in comparison to non-modules builds. The average peak memory usage increases by 25%. The increase is due to unimplemented optimizations in the C++ Modules system. The time of writing of this paper the C++ Modules system deserializes template specialization declarations in a non-lazy manner. This happens only when a template definition is used in a multi-module setup. The non-lazy template specialization declaration deserialization consist of around 37% of
the total deserialization for some translation units. This triggers a “domino-effect” because the bodies of some declarations which are definitions is deserialized together with all used types.

![Figure 1: Timing and Peak Memory Usage Comparison](image)

The benchmarks in Figure 1c and 1d provide information about the potential use of the C++ Modules as a replacement for shared library’s descriptors. The comparison is done between modules and non-modules usage. The third bar denotes the performance of our internal implementation of lazily loading of template specialization declarations. The four cases simulate single- and multi-module environment of a shared library with module file as a descriptor:

- **STL (unused)** – has dozens of included header files from STL without a use of their content. It aims to simulate explicit library loading where the library is only made available;

- **STL (partly used)** – has dozens of included header files from STL with a sparse use of their content. It aims to simulate implicit loading of library by interaction at ROOT’s prompt in a single-module setup;

- **ModDeser** – has several included header files from different ROOT libraries and creates objects of these types. It aims to simulate implicit library loading in a multi-module setup;

- **ROOT Eve** – has a lightweight descriptor (when compiling without modules), a few included files and creates objects of these types. It aims to simulate the removal of the lightweight descriptor of ROOT’s libEve in multi-module setup.

The results show noticeable performance improvement expectancy if C++ Modules are adopted in ROOT’s runtime. The ROOT Eve example measures the performance of C++ modules instead of lightweight shared library descriptors. The observed speedup is around 1000% and the memory footprint is reduced by 25% (using our local optimization for late deserializations).
5. Conclusion & Future Work

The C++ Modules system is promising technology. It combines years of development efforts of industry to solve a very practical problem: build time scalability. ROOT uses C++ Modules at compile time and has a plan to use them at runtime. C++ Modules reduce ROOT’s compile time by approximately 40%. The paper complements the compile-time results with preliminary results of the potential usage of C++ Modules to optimize ROOT performance at runtime. At runtime, the replacement of the full and lightweight shared library descriptors lead to observed parsing speedups of about 1000%. The memory footprint results are inconclusive but show a possible reduction by 30%-40%.

ROOT continues to evolve towards C++ Modules-friendliness. In future, the redundant macro definitions in the codebase should be removed. ROOT should trim the duplicate information module files which come from libraries without module files. Improvements can be done to minimize the recompilations and the memory footprint. More studies about replacement of ROOT’s dictionaries with modules files should be made.

Despite the mature C++ Modules implementation in clang, it can be improved in a few places. One of them is implementing the aforementioned lazy deserialization of template specializations optimization. Another relatively easy step is fixing a deficiency in the module file macro re-export logic. Module files relocatability should be improved by replacing the full file paths with relative. In addition, the implementation should generate new module files only when the compilation options cause incompatible module units.

The design and the implementation of the system both in ROOT in clang is flexible and allows further tweaks. The preliminary results encourage deeper runtime integration. The full adoption of the feature will not only improve performance but it will also eliminate the error-prone and limited recursive parsing. It will extend the capabilities of the implicit library loading, allowing to load entities such as function declarations.

Acknowledgments

The work is supported by USCMS and FNAL. The author is also grateful to Raphael Isemann for the assistance with the performance measurements.

References

[1] Reis G D, Hall M and Nishanov G 2015 N4465: A Modules System for C++ (Revision 3) Technical specification International Organization for Standardization Geneva, CH

[2] Smith R 2016 There and Back Again: An Incremental C++ Modules Design cppCon URL https://cppcon2016.sched.com/event/7mM6/there-and-back-again-an-incremental-c-modules-design

[3] Clang Modules Documentation http://clang.llvm.org/docs/Modules.html accessed: 2017-02-11

[4] Klimek M 2016 Deploying C++ Modules to 100s of Millions of Lines of Code cppCon URL https://cppcon2016.sched.com/event/7mM2/deploying-c-modules-to-100s-of-millions-of-lines-of-code

[5] Gregor D 2016 Modules cppCon URL http://llvm.org/devmtg/2012-11/#talk6

[6] Swift Modules Documentation https://github.com/apple/swift/blob/master/docs/Modules.rst accessed: 2017-02-11

[7] ISO 2012 ISO/IEC 14882:2011 Information technology — Programming languages — C++ (Geneva, Switzerland: International Organization for Standardization) URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[8] Modularize Documentation http://clang.llvm.org/extra/modularize.html accessed: 2017-02-11