Evolution of ROOT package management

O Shadura[1], B Bockelman[2] and V Vassilev[3]

[1]University Nebraska − Lincoln, Lincoln, NE 68588, USA
[2]Morgridge Institute for Research, Madison, WI 53715, USA
[3]Princeton University, Princeton, NJ 08544, USA

E-mail: oksana.shadura@cern.ch

Abstract. ROOT is a large code base with a complex set of build-time dependencies; there is a significant difference in compilation time between the "core" of ROOT and the full-fledged deployment. We present results on a "delayed build" for internal ROOT packages and external packages. This gives the ability to offer a "lightweight" core of ROOT, later extended by building additional modules to extend the functionality of ROOT. As a part of this work, we have improved the separation of ROOT code into distinct modules and packages with minimal dependencies. This approach gives users better flexibility and the possibility to combine various build features without rebuilding from scratch.

Dependency hell is a common problem found in software and particularly in HEP software ecosystem. We would like to discuss an improvement of artifact management ("lazy-install") system as a solution to the "dependency hell" problem.

HEP software stack usually consists of multiple sub-projects with dependencies. The development model is often distributed, independent and non-coherent among the sub-projects. We believe that software should be designed to take advantage of other software components that are already available, or have already been designed and implemented for use elsewhere rather than "reinventing the wheel".

The main idea is to build the ROOT project and all of its dependencies recursively and incrementally, making it fundamentally different than just adding one external project and rebuilding from scratch. In addition, this allows to keep a list of dependencies to be able to resolve possible incompatibility of transitive dependencies caused by the versions conflict.

In our contribution, we will present our approach to artifact management system of ROOT together with a set of examples and use cases.

1. Introduction

During the different stages of development, through the years ROOT [1] was using various build tools to manage its code, starting from custom build tools going through recently-removed Makefiles and currently using CMake [2] – a cross-platform build-generator tool.

Build system types are usually a very opinionated topic to discuss. Following the talks from notorious developer's gatherings and conferences, often build systems and CMake in particular is ridiculed [3]. A set of *Modern CMake* tutorials are developed to encourage good practices among the developers communities. They suggest to start treating CMake as code and think in targets, use properly CMake interface targets either for exports or imports. ROOT is no different, even though the CMake build system is rather new development it can be improved. Using community good practices addresses variety of problems of ROOT build and installation

procedures. It is evident that there were a lot of efforts done by the ROOT team to modernize ROOT CMake code and provide a stable and reliable support for ROOT build system [4].

This paper gives some insights of used good practices and what they allow users to do beyond building and shipping ROOT in the standard way.

2. Background

Evolving ROOT CMake to improve ROOT installation also allows decreasing build complexity and enables further decoupling of semantically independent components. Component decoupling is essential for embedding ROOT in other ecosystems as it enables more precise configuration of what the ecosystem actually uses. Ideally, ROOT should allow building ROOT component or package on top of already pre-configured or pre-build ROOT. In order to make the ROOT packaging more flexible and less monolithic was introduced an idea of a ROOT-aware dependency manager [5], that could provide missing functionalities for ROOT.

Current organisation of key components for ROOT build system is very stable. They use the following, fundamental to CMake, concepts:

(i) ROOT libraries (library targets) together with its specially organised code and tests;
(ii) ROOT build options to manipulate with available to user, set of libraries as build deliverables or artefacts;
(iii) a set of standalone projects, integrated in ROOT, such as LLVM/Clang, Clad and etc;
(iv) ROOT dependencies divided in two groups: dependencies hosted by ROOT - *ROOT builtins* and external OS package dependencies.

ROOT CMake build options are divided into the two groups: few build features, such as *cxxmodules, runtime_cxxmodules* and multiple build options such as *gsl_shared, xml*. Both groups are interdependent and sometimes looks outdated. The ROOT options are named after directory hosting a library's sources and CMakeLists.txt. That's why sometimes it looks like ROOT CMake options has an ambiguous naming convention and user doesn't know what will be enabled with the particular option.

Imagine situation when user is requesting to build ROOT "Core" libraries (libCore, libCling, and libRIO) and to enable machine learning libraries on of them via TMVA option. Outcome will be that instead of enabling only TMVA option, the user will enable other not requested N libraries. This example helps to explain the missing build functionality in ROOT, the possibility of the delayed builds.

As a solution, we propose to introduce the concept of a ROOT sub-package.

3. Implementation ideas

We can unite libraries with dependencies into the groups or "sub-packages", dealing with ROOT as a "fat" meta-package. A simplified version of this concept is shown on the Figure 1. In the next subsections, we will try to explain issues that could be resolved in ROOT, introducing ROOT sub-package concepts. Main focus will be to conclude the ROOT layering issues, ROOT dependency management, and options management.

3.1. Layering ROOT: design goals

The main intention here is to arrange existing ROOT components into layers. For instance, as a simple ROOT layering example, could be a set of ROOT Math packages. What we would like to achieve is the possibility to enable the next chain:

$$(core) \rightarrow (mathcore) \rightarrow (mathmore). \tag{1}$$

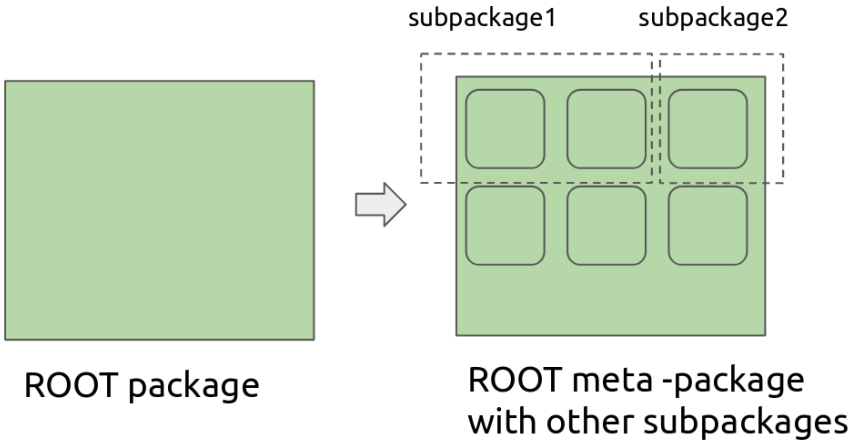

Figure 1: Evolution of ROOT package.

Layering concept allows each layer or subpackage to be enabled or disabled independently. It is available via implementation done as an overload of CMake *add_subdirectory*() function with iteration loop through special configuration file, making it similar to a package database. Identical implementation also exists in LLVM project [7]: a custom *add_llvm_subdirectory*(), *add_clang_subdirectory*() and similar implementations for other LLVM tools and projects.

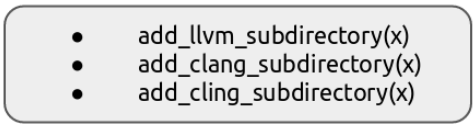

Figure 2: Examples of custom *add_subdirectory*() from LLVM projects.

We propose a new way of organization of ROOT build options, using a map that is similar to a custom database of ROOT packages. Its implementation gives the possibility to enable single ROOT library with its dependencies in one configuration step, which means that we will be able to configure, build, and deliver the ROOT layers iteratively, layer by layer. To enable this feature, we introduces the ROOT package map. On Figure 3 is shown a simplified example of ROOT package map, allowing to enable Base, IO sub-packages together with its dependencies, step by step.

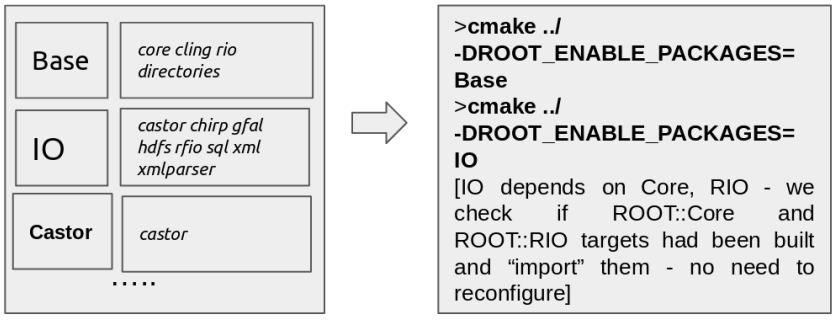

Figure 3: ROOT package map and its functionality within ROOT.

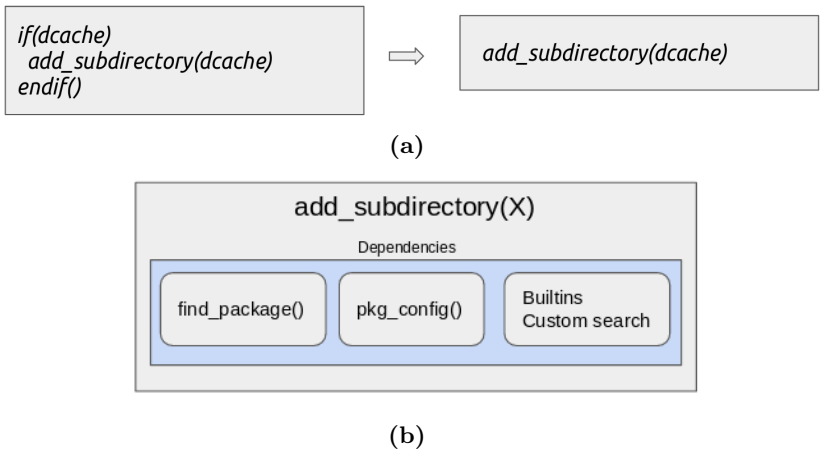

Figure 4: ROOT CMake improvements: (a) ROOT CMake dependency simplification. (b) new procedure for ROOT CMake *add_subdirectory*().

3.2. Simplification of ROOT dependencies

ROOT has a very complex map of dependencies, both internal and external. The idea is to simplify how ROOT dependencies are treated. It will allow to introduce CMake code clarity (see an example on Figure 4a) and make ROOT more modular. To enable separability of ROOT layers, inside *add_subdirectory*() was introduced a way to treat dependencies in a standalone way: ROOT builtins via separate custom search procedure and external dependencies, using CMake $find\_package()$, $pkg\_config()$ or could be even a simple integration of any other CMake based C++ package manager, such as Conan [6] (check Figure 4b).

4. Results

The preliminary results show new possibilities to build ROOT iteratively, package by package. Improvements, proposed in this paper will cover most of the use cases requested by the HEP community to enable a way to configure, build, and use ROOT in the more modular way.

As a consequence, it will help during some critical for users situations such as, when user has already built ROOT from sources and desire to extend its functionality without rebuilding ROOT from scratch or *typical ROOT developer case*, when is actively developed only one of the ROOT component and developer wants to test only this component, without re-configuring all ROOT.

New functionality will help to enable additional range of possibilities for ROOT.

- ROOT-aware package manager prototype *root-get*: it can enable the ROOT-aware package management with the root-get prototype. [5]
- *OS package management* (e.g. Fedora, Ubuntu and etc.): it will be easier to generate more granular ROOT packages.
- Package, dependency and environment manager *Conda*: it will support a root-minimal package to further improve install times.

A new functionality is expected to be enabled in ROOT 6.20.00 for ROOT C++ modules [8] as an experimental feature.

5. Acknowledgments

This work has been supported by U.S. National Science Foundation grant ACI-1450323.

References

[1] R. Brun, F. Rademakers. ROOT - An Object Oriented Data Analysis Framework. 1997 *Nucl. Inst. & Meth. in Phys. Res. A* **389**, Proceedings AIHENP'96 Workshop.
[2] GitLab. 2019. CMake / CMake · GitLab. [ONLINE] Available at: https://gitlab.kitware.com/cmake/cmake. [Accessed 24 May 2019].
[3] Henry Schreiner. An Introduction to Modern CMake · Modern CMake, 2019. [ONLINE] Available at: https://cliutils.gitlab.io/modern-cmake/. [Accessed 22 May 2019].
[4] Guilherme Amadio. Evolution of ROOT's CMake Build System. 2019. ROOT Users' Workshop (10-13 September 2018) · Indico. [ONLINE] Available at: https://indico.cern.ch/event/697389/contributions/3062044/. [Accessed 22 May 2019].
[5] Oksana Shadura, Brian Paul Bockelman, Vassil Vassilev. Extending ROOT through Modules, CoRR, arXiv:1812.03145 [cs.SE]
[6] C/C++ Open Source Package Manager. 2019. C/C++ Open Source Package Manager. [ONLINE] Available at: https://conan.io/. [Accessed 24 May 2019].
[7] GitHub. 2019. GitHub - llvm/llvm-project: This is the canonical git mirror of the LLVM subversion repository. [ONLINE] Available at: https://github.com/llvm/llvm-project. [Accessed 24 May 2019].
[8] Yuka Takahashi, Vassil Vassilev, Oksana Shadura, Raphael Isemann. Optimizing Frameworks Performance Using C++ Modules Aware ROOT, CoRR, arXiv:1812.03992 [cs.PL]