Journal of Physics: Conference Series **664** (2015) 062049 doi:10.1088/1742-6596/664/6/062049

ROOT6: a Quest for Performance

Danilo Piparo
CERN CH-1211, Switzerland
E-mail: danilo.piparo@cern.ch

Abstract. The sixth release cycle of ROOT is characterised by a radical modernisation in the core software technologies the too kit relies on: language standard, interpreter, hardware exploitation mechanisms. If on the one hand, the change offered the opportunity of consolidating the existing code base, in presence of such innovations, maintaining the balance between full backward compatibility and software performance was not easy. In this contribution we review the challenges and the solutions identified and implemented in the area of CPU and memory consumption as well as I/O capabilities in terms of patterns. Moreover, we present some of the new ROOT components which are offered to the users to improve the performance of third party applications.

1. A Big Change: CINT to Cling

The ROOT [1] tool kit featured from the very beginning a C++ interpreter. Until the sixth ROOT release cycle, the interpretation of C++ code was delegated to CINT [2], an interpreter covering most of ANSI C (including C99) and ISO C++03. Despite the rich set of functionalities CINT offers and the success it had during years of production usage, it is not adequate to cope with the new C++ standards, most notably C++11. This limitation is a a blocker for the support of refection and I/O in presence of user classes and data models written according to the newest C++ paradigms and does not allow to evolve the interfaces of the components of ROOT itself beyond C++03.

The solution adopted to overcome this hurdle was to replace CINT with the Cling [3] interpreter, based on the LLVM compiler infrastructure [4]. This transition represents a veritable advancement in the software technologies leveraged by ROOT. For example, Cling is the first of its kind, offering just in time compilation of C++ code. The challenges involved in this evolution were many. Entire ROOT components had to be rewritten to comply with the interfaces offered by the new interpreter, notably the type system. All the features existing in ROOT 5 had to be supported as well as an ambitious set of new ones. All of this under the scrutiny of a large user base. This user base includes also systems consisting in millions of lines of code in which ROOT is integrated, for example the software stacks of the LHC experiments.

This kind of investment is an opportunity. It allows to study and improve the strategies adopted to evolve scientific software, for instance with agile techniques.

2. Clang, the AST and ROOT

The C++ front-end of LLVM, Clang, provides a very efficient implementation of an Abstract Syntax Tree (AST). This entity holds all the information expressed in the source code, like

classes, functions, templates and statements. This data structure can be persisted on disk in two forms, the Pre-Compiled Header (PCH) and the Pre-Compiled Module (PCM) - see figure 1.

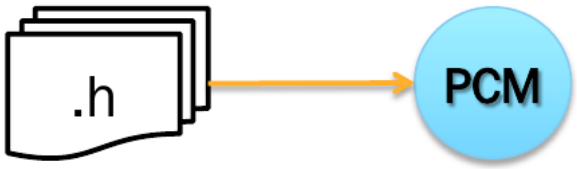

Figure 1: The PCMs can be considered for interfaces the entity which corresponds to libraries for implementations. They are a condensed version of a consistent set of headers Clang can create. While the Objective-C PCMs could be created, the time scales relative to the technology for the creation of C++ PCMs were not compatible with the LHC long shut down plans.

The former can be thought as a cache for header files. Only one PCH can be loaded at runtime by the compiler. The latter has on the other hand has the granularity of single AST nodes and many PCMs can be loaded by the compiler during one invocation. A relevant commonality between the two formats is that they can be queried lazily by Clang.

The original design of ROOT 6 is shown in figure 2. It relies on the presence of both PCHs and PCMs for the C++ language with the objective to leverage the information contained there for reflection, I/O and interactive function calls.

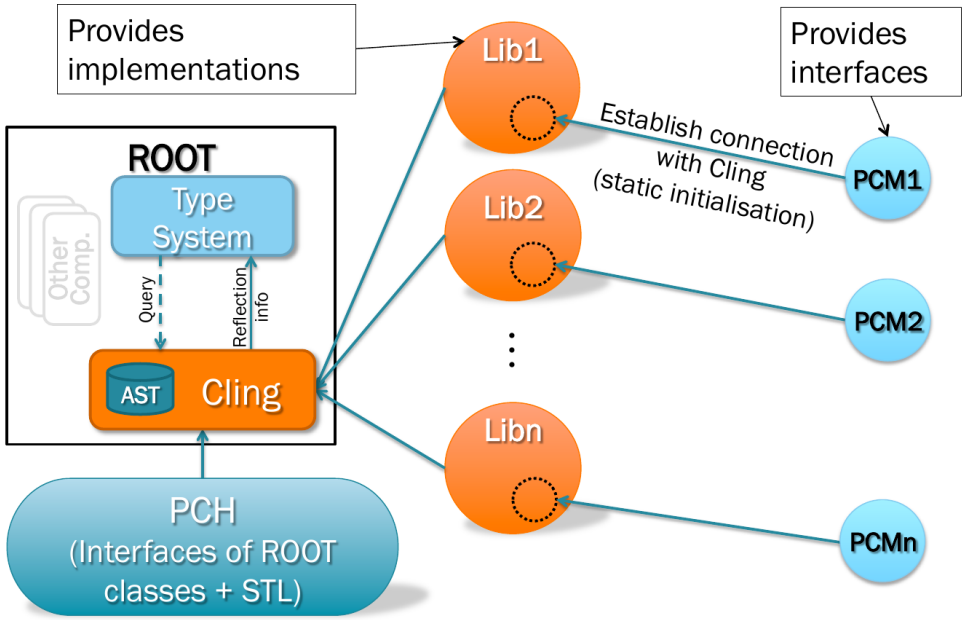

Figure 2: Simplified sketch of the ROOT 6 design. At library load time a connection is established between the PCM relative to a dictionary and the interpreter. Reflection information contained in PCMs and PCH is queried lazily by the interpreter. Dictionaries are a thin layer to interface the interpreter and the persisted AST nodes.

3. A Change of the Original Design

C++ PCMs were a bleeding edge technology during the creation of ROOT 6 and a sturdy implementation of this concept by the Clang team by the beginning of the LHC Season 2 was

expected. Unfortunately, the arrival of PCMs was delayed. Without a persistent representation of the portion of the AST relative to each single dictionary, all headers had to be parsed at runtime (see figure 3), with significant penalties both in terms of runtime and memory footprint.

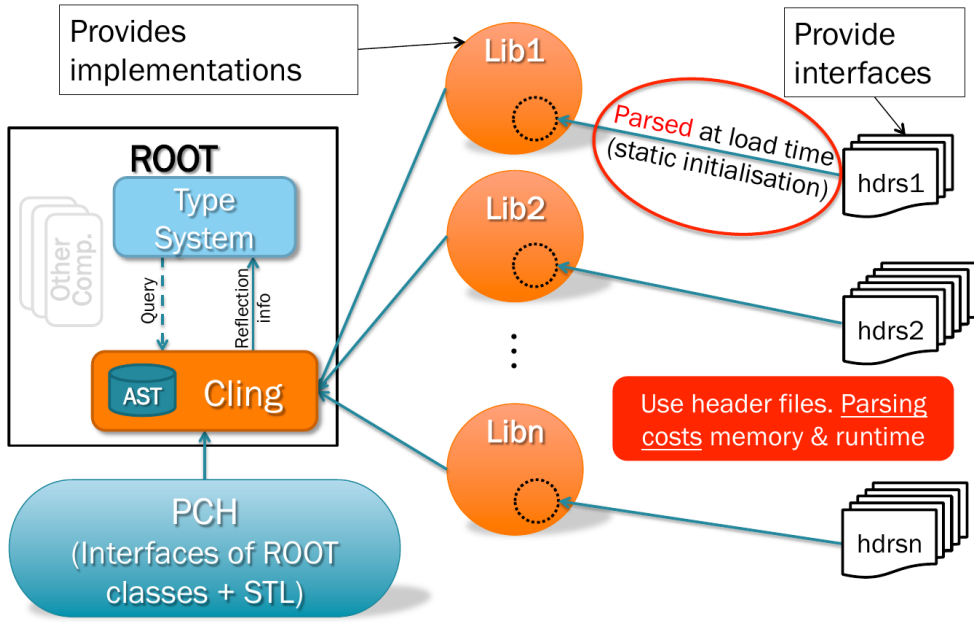

Figure 3: In absence of PCMs, the only way to make known to the interpreter the interfaces of the entities implemented in the library is to parse all the headers. Clang, as every other compiler, offers optimised mechanisms for parsing quickly and efficiently source code. Nevertheless the impact on the runtime and memory footprint for complex systems like the LHC experiments' software stacks was measured to be severe. Parsing needed to be reduced.

In order to reduce header parsing to the bare minimum and increase performance of ROOT, two strategies were adopted.

The former is relative to the treatment of the information necessary for I/O. The description of the layout of the classes was extracted from the AST during the generation of the dictionaries and persisted in special ROOT files, called "ROOT-PCMs". At library load time the ROOT-PCMs are read and the information directly injected in the ROOT type-system as shown in figure 4.

The latter is relative to the interactive usage and is called "parsing on demand". The header relative to a certain dictionary were not parsed in bulk at loading time, but the parsing is delayed until a function implemented in the library has to be called.

4. The Profiling Toolbox

In order to study the performance figures of ROOT 6, different profilers were used. The main ones were IgProf [5], the Valgrind [6] suite and some other simpler solutions described in the following.

The IgProf profiler was chosen both for memory and runtime studies. The main reason was the very little overhead imposed to the running application and the possibility to "snapshot" counters at any given point in time, for example, in the context of LHC data processing, when transitioning from one event to the other. The possibility to share reports via its web interface was also extremely useful to exchange information among the ROOT and the experiments' experts.

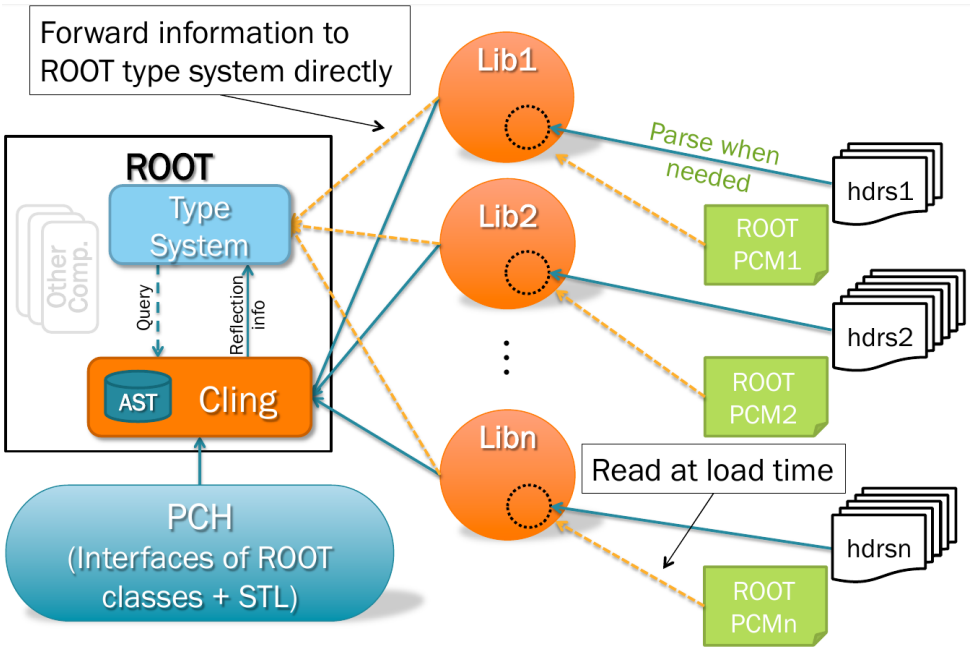

Figure 4: The ROOT PCM is a ROOT file the format of which is optimised to guarantee maximum reading speed. It contains the information necessary to perform I/O of objects, e.g. the layout of selected classes and is created together with dictionaries.

Of the Valgrind family, both Callgrind and Massif were considered. The former for the measurement of very short executions (see for example section 7), the latter to complement IgProf for the measurement of memory footprints.

It is worth noting that other effective strategies were adopted. For example, Kernel data structures were queried via the `TSystem::GetProcInfo` in order to be able to print on screen the memory footprint before and after the invocation of a certain method. This kind of simple approaches certainly does not scale to complex systems (and is not meant for those) but it is necessary to get overall impressions of the performance figures before running more complex profilers.

5. Performance Figures: the CMS Case

The solutions mentioned in section 3 were not enough to fit the new ROOT within the version 5 envelope. Several improvements in all corners of the tool kit were put in place and properly validated but their characterisation is beyond the scope of this document.

In order to show the effect of the aforementioned improvements, the runtime and memory consumption of ROOT 6 and ROOT 5 are compared using production CMS work flows. The software stacks are identical up to the ROOT version integrated. Two processes have been studied, the simulation of $pp \rightarrow t\bar{t}$ events at a centre of mass energy of 13 TeV and their reconstruction. The runtime of the event loop is identical in the two cases. The RSS memory consumption of ROOT 6 compared with the one of ROOT 5 is: 6% smaller for generation and simulation and 4% bigger for reconstruction.

6. Ensure Correctness: Testing

A fully automated test suite featuring a complete coverage is necessary for a campaign aiming to increase software performance to succeed. Developers must always be in condition to verify

Nightly

Site	Build Name	Update	Configure		Build		Test		
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass
moonshot-arm64-03	master-aarch64-ubuntu14-gcc49-dbg	1	0	0	0	0	0	2	1066
moonshot-arm64-04	master-aarch64-ubuntu14-gcc49-opt	1	0	0	0	0	0	1	1067
lcgapp-cc7-x86-64-10.cern.ch	master-x86_64-cc7-gcc48-opt	1	0	0	0	0	0	0	1089
lcgapp-cc7-x86-64-10.cern.ch	master-x86_64-cc7-gcc48-opt-classic	1	0	0	0	0	0	0	236
ec-fedora20-x86-64-3	master-x86_64-fedora20-gcc48-opt	1	0	0	0	0	0	0	1089
ec-fedora20-x86-64-3	master-x86_64-fedora20-gcc48-opt-classic	1	0	0	0	0	0	0	236
macitois13.cern.ch	master-x86_64-mac1010-clang36-opt	1	0	0	0	8	0	1_{-2}	1112^{+2}
macitois14.cern.ch	master-x86_64-mac1010-clang36-opt-classic	1	0	0	0	2	0	0_{-1}	236^{+1}
macitois18.cern.ch	master-x86_64-mac108-clang34-opt	1	0	0	0	8	0	1_{-2}	1112^{+2}
macitois17.cern.ch	master-x86_64-mac108-clang34-opt-classic	1	0	0	0	2	0	1	235

Figure 5: Some of the results displayed by the official web interface of the ROOT builds and testing status: `http://cdash.cern.ch/index.php?project=ROOT`. The first two lines are relative to builds on 64-bits ARM systems.

correctness after operations such as improvement of an algorithm or data structure as well as the introduction of new compilers or external tools. A significant effort was invested in the expansion of the ROOT test suite (see figure 5) in order to both increase coverage and check correct functioning of all external plug-ins, for example Davix or xRootd. The goal was to target test driven development. A particular attention was also dedicated to the growth of supported hardware architectures (x86_64, ARM, Power PC LE) and operating systems (various Linux distributions, OsX, Windows).

7. Caring About Details: Start up Time

For a tool kit like ROOT, the attention cannot be focused only on the integration in large software systems (see section 5): every detail counts. This is the reason why work was invested to improve the very first feature seen by a user: the start up time of ROOT. See figure 6.

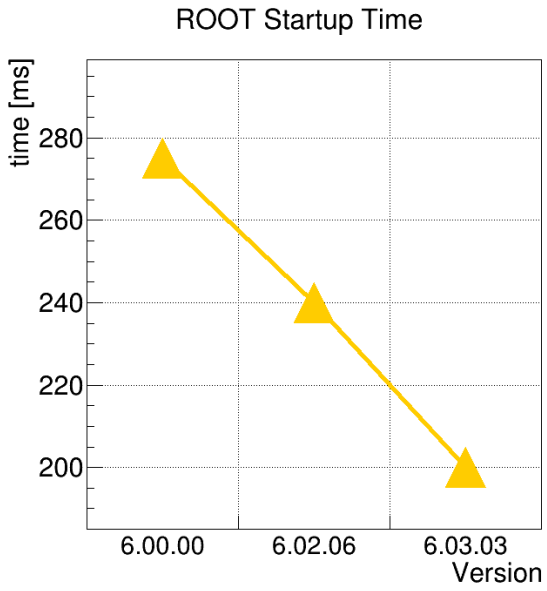

Figure 6: Start up time of ROOT in ms versus the tool kit version. A steady improvement is clearly shown.

The ingredients behind this reduction are the caching of the I/O information of all most used ROOT classes as well as STL interfaces in the PCH file, the usage of the unordered STL containers and a redesign of the format of the "ROOT-map" files which hold the meta data exploited by ROOT to manage automatic loading of plug-ins at runtime.

8. Leveraging Modern Compilers

Up to now only algorithmic and design changes aiming to performance increase were mentioned. On the other hand, the so called "technical" improvements can greatly influence the runtime behaviour of scientific applications. The current CMake [7] based build system of ROOT, allows to optionally enable the maximum level of optimisation allowed by both Clang and GCC [8], including the non IEEE compliant treatment of floating point numbers. These optimised builds are also useful from the testing point of view. Indeed they allow to check the numerical stability of the code of ROOT. The results of this runtime improvement is illustrated in figure 7.

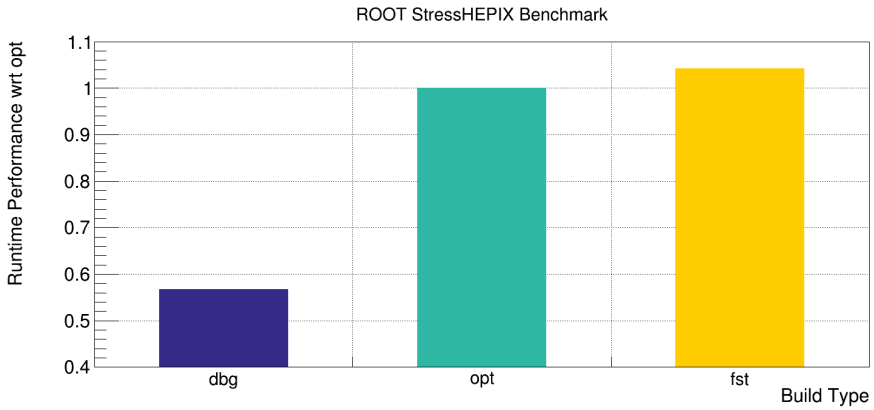

Figure 7: Results of the StressHepix benchmark of ROOT. The result is expressed relative to the runtime performance of the optimised build, the higher the better. Three optimisation levels are studied: debug, optimised and fast. They roughly correspond to the "-O0","-O2" and "-Ofast" GCC flags. The compiler used was GCC 4.9.2.

9. Conclusions

ROOT 6 offers, in addition to its Clang and LLVM based interpreter Cling, many new features while granting backward compatibility: the performance figures are fitting the envelope of the previous ROOT release cycles, therewith leaving the floor to a future steady development of the tool kit.

The migration from CINT to Cling has clearly shown that agile principles are an asset when betting on cutting edge software technologies and that close collaboration with users and clients is a clear benefit for software projects of the size of ROOT.

The requirements set by the LHC experiments were satisfied but room for further improvement in the area of performance is still left. The quest for performance will continue, for example introducing even more sophisticated containers in the ROOT core components, exploiting more vectorisation and improving the integration of the tool kit with software profilers.

References

[1] Brun R et al. 1997 *Nucl. Inst. Meth. In Phys.* vol 389
[2] Goto M *C++ Interpreter - CINT* CQ publishing ISBN4-789-3085-3
[3] Vassilev V et al. *Cling The New Interactive Interpreter for ROOT 6* 2012 J. Phys.: Conf. Ser. 396

[4] LLVM http://llvm.org
[5] Eulisse G Tuura L *IgProf profiling tool* Proc. CHEP04, Computing in High Energy Physics, Interlaken
[6] Nethercote N Seward J *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation* Proceedings of ACM SIGPLAN 2007
[7] CMake http://www.cmake.org
[8] The GNU Compiler Collection https://gcc.gnu.org