Journal of Physics: Conference Series **396** (2012) 052021 doi:10.1088/1742-6596/396/5/052021

A CMake-based build and configuration framework

M Clemencic and P Mato

CERN, CH-1211 Genève 23, Switzerland

E-mail: marco.clemencic@cern.ch

Abstract. The LHCb experiment has been using the CMT build and configuration tool for its software since the first versions, mainly because of its multi-platform build support and its powerful configuration management functionality. Still, CMT has some limitations in terms of build performance and the increased complexity added to the tool to cope with new use cases added recently. Therefore, we have been looking for a viable alternative and we have investigated the possibility of adopting the CMake tool, which does a very good job for building and is getting very popular in the HEP community. The result of this study is a CMake-based framework which provides most of the special configuration features available natively only in CMT, with the advantages of better performances, flexibility and portability.

1. Introduction

Software projects always need tools to configure the build options and build the final applications or libraries from sources. These are tasks that developers have to repeat several times every day, so it is extremely important to ensure that the process is under control and reproducible. For *open source* projects it is even more important to have simple tools, because the released versions of the project may be taken and compiled by end users, which probably are not expert developers.

Configuration and build tools range from self-developed shell scripts to *projects* inside Integrated Development Environments (IDEs). Though, the needs and the problems are common, so some developers worked on common solutions, which are then used by other developers. The configuration and build tool chosen for a project depends mainly on the kind of requirements the developers have, for example the level of portability that she wants to achieve, the size of the project or the maintainability of the files used to describe the project to the tool.

The original developers of the LHCb Software framework Gaudi[1], the foundation of all LHCb software, had to choose a configuration and build tool matching some specific requirements. For LHCb, like for any other High Energy Physics (HEP) experiment, it is fundamental to be reproduce the results obtained using an application and compare them with the results obtained with other versions of the same application, so we need to be able to easily switch from one version of the application to another one, including the external libraries used by them. LHCb software was organized around the concept of *packages*, which are grouped in *projects*, to allow for a modular and flexible development model.

The requirements for modularity, flexibility and control imposed by the developers of the LHCb Gaudi Software Framework were met by the tool CMT[2, 3], which has been used so far by LHCb, ATLAS and other groups. Unfortunately, with the growth in size and complexity

of the software projects in LHCb, CMT started to show limitations, the main one being its slowness on projects with a large number of packages.

Now, after more than 10 years, the panorama of configuration and build tools has evolved considerably. In particular, a new application appeared: CMake[4]. It demonstrated to be a powerful and simple tool, more efficient than the competitors, so that it has been adopted by several projects, including, for example, the Linux desktop environment KDE[5][1].

2. Motivations

CMT has got some unique features that are extremely useful, but, on the other hand, it also has some limitations.

The software of HEP experiments is usually developed by a large number of non professional programmers, i.e. the physicists participating in the experiment. In such a team, more than in others, it is vital to keep the boilerplate code to a minimum, both for the configuration system and the actual applications. The programmability of CMT, together with the possibility of importing configuration code from other packages in form of functions or macros, makes it possible to conceive real configuration and build frameworks on top of it.

CMT is designed around the concept of packages. Packages are very easy to add to or remove from a project, so that it is possible to reorganize the packages moving them from one project to another without having to modify the configuration files[2]. From this design choice derives the possibility of overriding packages in derived projects, more or less as in OOP a derived class can have methods that override those of the base class. This makes it possible to decouple the release cycles of projects, so that a bug can be fixed in a version of the derived project without having to wait for the fixed version of the base one.

CMT is also useful when developing on different platforms. It is available as pre-compiled binary for Linux, MacOSX and Windows, and, being an open source project, it can be compiled for other platforms.

The configuration files are read by CMT every time that the `cmt` command is invoked. This makes CMT extremely dynamic and responsive to changes in the configuration files. Unfortunately there is a price to pay. While the impact of this approach is negligible for small projects, the time spent in the lookup and parsing becomes appreciable when the list of used packages grows to the order of $\sim$100 (as in LHCb projects) and it is unbearable when such a project is accessed via a network filesystem (which has higher latency than a local one)[3].

The language used to write CMT configuration files allow the creation of *patterns* (similar to macro functions of the preprocessor of the C language) and the tuning of the build via *tags* (similar to the macro values of the C preprocessor). Unfortunately tags, once set, cannot be "undefined" as in C, and the logic on tags is limited to "positive" checks (i.e. it is not possible to express natively a check like "if not A"). We can work around these limitations, but only up to a certain point, and the code required is hard to read and difficult to maintain.

After 10 years of use of CMT, with a better and deeper understanding of our needs and the appearing of several new configuration and build tools on the market, we decided to see if there was a suitable replacement.

CMake, with its portability, its advanced language and its extensibility, seemed to be a good candidate. Moreover, some expertise was available at hand, because CMake has been used to improve the builds of other HEP projects, notably Geant4[6, 7] and ROOT[8–10].

[1] It should be noted that the feedback that KDE developers gave to the CMake ones, produced the powerful and flexible product that it is now.

[2] Actually, we do need to modify few files when we do this kind of movements, but only those describing the content of the project and not the real configuration files.

[3] Some measurements can be found in Table 2.

3. Requirements

Despite the fact that both CMT and CMake generate the necessary files for a lower level (native) build system, they have been developed in different contexts, with different philosophies, so there are differences in the way the configuration files must be written.

Luckily, over the years, LHCb (in collaboration with ATLAS) moved from a plain use of CMT to the development of a configuration framework written in the CMT language. We extended and customized CMT through patterns, so that the casual developer does not need to know the technical details of the CMT configuration, but it is enough to know a very limited set of "functions", to be used in the uncommon case where the standard template configuration is not enough.

Thanks to the specialization we developed on top of CMT, we do not need to implement using the CMake language all the functionalities and features proper to CMT: it is enough to implement only our own specialized language. It is unavoidable to have differences between the configuration files written for CMT and CMake, of course, because of the difference in the syntax, but it must be possible to map the concepts almost directly from one to the other.

In this context, we can define what are the requirements that the build and configuration framework written in CMake has to fulfill to be a valid replacement for the one written in CMT.

3.1. Modularity and Flexibility

We have been using extensively the possibility, available in CMT, of organizing the code in packages and projects.

In the context of the migration, we can define the packages as the minimal units that can be versioned. Packages can *use* other packages, in the sense that a package can require the build of another one to use the libraries produced there.

Projects are essentially collection of packages and are the minimal released unit. They can depend on each other, which means that the packages in a derived project can use the packages of the base one. As already mentioned before (section 2), packages can be easily added or removed in a project, so that it is pretty common that a package is moved from a project to another if needed. It is also common for a derived project to override a package present in a base project in order to pick up a urgent bug-fix, without having to wait for a new release

The development model of LHCb software is based on the possibility of overriding packages. Regular and casual developers create (via custom tools) temporary local projects that derive from a released one. In these small projects, without the need to recompile the whole code base, they can develop new packages, extend existing ones or fix bugs.

It is mandatory that the new framework allows to organize the code in concepts equivalent to those of packages and projects. Moreover, the bind between projects and packages has to be loose as it is in CMT, so that it is possible to add, remove or override packages.

3.2. Simplicity

The new framework has to be easy to use for the casual developer, and, if possible, even easier than the previous one.

LHCb developers are used to a limited set of elements that they can build:

Linker library: collection of functions and classes used as foundation for other elements

Component library: plug-in library, loaded at runtime to extend the software framework (most of the code in a Gaudi-based project is compiled into component libraries)

Executable: application that can be executed (almost never used outside the software framework itself)

In addition to the elements to be built, we also need to consider files that must be made available without the need of a build step. For example, our software projects include scripts

(mainly in the Python scripting language[11], in which case, the files must be installed in a standard location so that they are accessible to the users. Another use case is that of data files, which have to be made available either via installation in a standard location or via the definition of environment variables.

The new framework must include simple functions to declare any of the three entities that can be built, as well as functions to declare the files to be installed without the need of a build step.

3.3. Runtime Environment

As mentioned earlier, we need to be able to have several versions of a given application immediately accessible on the system. CMT solves the problem by relying on the environment variables used by the operating systems to locate executables, libraries and Python modules (`PATH`, `LD_LIBRARY_PATH` and `PYTHONPATH`).

To use a given version of an application, the environment must be prepared accordingly. CMT helps by providing hooks and functions to manipulate the environment. The runtime environment to be used is computed dynamically by CMT at every invocation, and, with a special command, can be set in the command line shell. CMT also defines automatically some environment variables useful to locate the source location of the packages (e.g. to access data files).

It is important that the manipulation of the runtime environment is possible also in the new configuration framework, although it does not need to be as dynamic as in CMT.

3.4. Smooth Migration

The software of LHCb consists of about 30 projects for a total of more than 700 actively developed packages. All these projects are maintained by different people and have different release cycles.

The new framework must be designed and implemented to allow a migration as smooth as possible, also allowing the developers to switch to it at different moments.

4. Features and Shortcomings of CMake

The differences between CMake and CMT should be considered in light of the requirements on the new configuration and build framework.

The modularity and flexibility required, preserving the current structure of the projects, can be realized via the built-in functions to query the filesystem and the concept of *subdirectories* already present in the core of CMake.

The language of CMake configuration files is very powerful, even though it has some uncommon features and could be difficult to read at first. The possibility of encapsulating complex code in macros and functions allows us to hide the implementation details and provide an easy and clean interface to the developers.

One of the features native in CMT that we rely heavily on is the inheritance of the link and compile flags through *linker libraries*. This means that, when a library A is linked against a library B, the paths to the header files used by B are added automatically to the compile commands used on the sources of A, and the link command of A includes all the libraries that where used to link B too. Unfortunately, CMake has got thas this feature only for the link command and not for the include paths.

The use cases considered when developing CMake did not include the possibility of manipulating the environment in the way we need. So the implementation of this feature will require the development of some auxiliary tool.

The concept of dependencies between packages, as implemented in CMT, is not present in CMake, but it is possible to create dependencies of individual build targets of one subdirectory

on targets defined in other subdirectories. This feature of CMake is more useful than the plain dependencies between packages that CMT provides, because it allows a finer granularity of the dependencies, with the benefit of more efficient parallel builds and more focused partial builds. Moreover, the targets of one project can be exported to other projects, so that the dependencies between targets can be used also across projects, as it happens for the dependencies between packages in CMT.

A big advantage of CMake over CMT is the presence of modules to locate external libraries. With CMT we used a collection of self-developed (in collaboration with other CMT users) *interface packages*, i.e. CMT packages defining compile and link flags to use external libraries. To be able to control the versions of the external libraries, these interface packages are pointing to predefined locations where we have custom builds for all those libraries. This configuration forces us to build our code on those custom builds, making our build set-up not very portable. The CMake modules to locate external libraries can be used both to locate the system versions or the versions in our custom set-up, making the set-up easier to maintain and more portable.

What CMake lacks in features is compensated by the completeness of its language. The inheritance of compile and link flags or the manipulation of the environment can be easily wrapped in custom functions that will be used in the configuration files.

5. Naming Conventions

In the design of the new CMake-based framework we could reuse the names used in the CMT one, but it would bind us to terms that are not common in CMake jargon, or, worse, that have different meaning. Moreover, when and if CMT will be completely replaced by the new framework, we would be stuck with legacy names that will just be confusing because not connected to CMake. So we decided to keep the concepts, but use the CMake names for those concepts (see Table 1).

Table 1. Mapping between the names used in the CMT-based framework to the ones in the CMake-base one. The names that are not in the table are identical in the two cases. The name *project* in CMake is used only to refer to the project currently being built, but "other projects" are referred to as packages, as for any other external library.

CMT	**CMake**
package	subdirectory
linker library	library
component library	module
project	project/package
external package	package

6. Design

The CMake-based framework has been designed around the concepts of *projects*, *subdirectories*, *toolchains* and *properties*.

Projects are, as in CMT, a collection of subdirectories, but, unlike in CMT, have also the more important role of entry points to the build system. The project configuration file is used to bootstrap the configuration process and to provide a context to the configuration of the subdirectories. It can include some global settings, but mainly it defines which other projects have to be known to the configuration and, through one main function, it collects the configurations of the subdirectories, defines some global (project-wise) targets and generates project-specific files.

Subdirectories, while hosting the actual code, define the elements to be built: executables, libraries ("linker libraries" in CMT) and modules ("component libraries" in CMT). They can

also declare which files have to be installed (like Python modules or scripts). Other elements can be declared in the subdirectories too, for example unit tests or custom targets.

Toolchains are a concept used in the context of cross-compilation, i.e. when we compile binaries on a system to use them on another one. Stretching a little bit the definition, we can effectively use the concept of toolchain available in CMake to describe the special set-up (the collection of custom builds of external packages) that we have been using so far in CMT. The advantage of this approach (apart from being a clean and simple way to use a compiler different from the default one on the system) is that it is enough to disable the toolchain to pick up the system version of the external libraries instead of the custom builds, making the projects easily portable to different systems.

Properties are a special feature of CMake. Many of the objects known to CMake (like targets and directories) have standard properties to which we can add custom ones. Properties are used by CMake to pass informations around, from the configuration files to the core system, for example there are properties to define link options or the name and the location of the output file. We can use the same mechanism to pass informations from a subdirectory to the project, for example to contribute to a global target or to some globally generated file like one describing the changes to be applied to the environment.

7. Implementation

The code had been structured in 4 main blocks:

Main CMake module: the core of the framework, containing the functions to be used in the subdirectories and the main function to be called from the project configuration file

Compile options module: the place where to define common compile and link flags, separated from the main module to be more easily tuned

Toolchains modules: modules defining the list of external libraries with their versions, based on a *common* module that defines some useful functions

Contributed *find* modules: modules written using the standard CMake conventions to find external libraries for which a standard module is not available

The following sections describe how the different components of a project are configured and what this means in terms of the framework.

7.1. The Project Configuration

This is probably the simplest configuration file in a project. It consists of a file called `CMakeLists.txt` (the name used for the configuration files of CMake) containing some minimal boilerplate code needed to locate and load the main CMake module, then the declaration of the project as a call to the function `gaudi_project` passing to it the name and version of the current project and the list of name and versions of the projects the current one uses, for example:

```
gaudi_project(MyProject v2r7
              USE SomeBase v7r0
                  SomeCoreProject v1r10)
```

Despite its simplicity, the function `gaudi_project` is the core of the framework. It sets some common variables and defines some basic build options. Then it looks for subdirectories that must be added to the configuration, i.e. those that contain files called `CMakeLists.txt` (section 7.2).

At this point it can locate the *used* projects (using standard CMake methods), recursing to locate also the projects indirectly required. The subdirectories exported (section 7.4) by the used projects are imported, except for those also present in the current project.

The discovered local subdirectories are sorted taking into account the dependencies that they declare on other subdirectories (section 7.2) before being added to the build configuration. This step is not very elegant but it is mandatory because CMake does not allow to use a target that has not been defined yet[4], so we have to ensure that if a subdirectory A uses products of a subdirectory B, the order of inclusion is such that B is included before A.

After the addition of the subdirectories, the global targets get created from the informations collected from the subdirectories. These are usually creating files merging the partial files produced in the subdirectories.

Always using the information collected via the properties of the subdirectories, the main function produces the file describing the required manipulation of the environment (section 7.3) and the files describing the exported targets (section 7.4).

7.2. The Subdirectories

The configuration files of the subdirectories are more complex, but the operations behind the few functions involved are simpler than those occurring in the main project function.

The file `CMakeLists.txt` of a subdirectory may contain a call to the function `gaudi_depends_on_subdirs` passing to it the list of subdirectories that must be built before the current one. This function is not doing anything for the actual configuration of the subdirectory, but the project main function parses the file `CMakeLists.txt` looking for that call in order to set a property holding the list of dependencies. The property is used to define the order in which the subdirectories must be added.

The declaration of dependencies is usually followed by a list of calls to the standard CMake command `find_package`, used to locate external libraries and to define the variables to be used in the compilation flags and link commands.

The declaration of binary products to be built is done using the family of functions `gaudi_add_`:

- `gaudi_add_library`
- `gaudi_add_module`
- `gaudi_add_executable`

plus some others for less common products. All of them require, as mandatory arguments, the name of the product and the source files that compose it. Optionally we can specify the list of libraries to be added to the link command and the list of directories to be added to the include path of the compilation commands. For libraries, it is mandatory to declare, with a dedicated named option, the local directories containing the header files that must be installed, or if there is no need to install extra headers. It is possible to use the name of the packages found as arguments for the list of libraries or for the include paths, which will by automatically expanded to the arguments that are actually needed by the CMake underlying functions. Examples of calls to these functions are the following:

[4] Actually it is possible, but the final build will be incorrect.

```
gaudi_add_library(GaudiUtilsLib Lib/*.cpp
                  LINK_LIBRARIES GaudiKernel ROOT
                  INCLUDE_DIRS ROOT AIDA
                  PUBLIC_HEADERS GaudiUtils)
gaudi_add_module(GaudiUtils component/*.cpp
                 LINK_LIBRARIES GaudiUtilsLib XercesC uuid
                 INCLUDE_DIRS XercesC uuid)
```

When a local or imported target is used as a link library, the header directories it exports and those that where used to build it are added in the include path of the target being defined. This is achieved by adding to the library targets a property containing the list of include directories that were used. It should be noted that the list of source files can be declared using shell glob patterns to simplify the migration from CMT, even though it is a practice strongly discouraged by the CMake developers.

In the subdirectory configuration it is possible to declare tests to the CMake testing and reporting infrastructure CTest.

If the subdirectory needs to get some files installed, the developer can use functions of the family `gaudi_install_`.

For special environment settings, the function `gaudi_env` provides a simple interface to declare environment operations, like `SET`, `APPEND` and `PREPEND`. For example:

```
gaudi_env(SET SomeVar AValue
          PREPEND PATH /path/to/dir
          APPEND PYTHONPATH /another/dir)
```

The operations on the environment are stored in a property of the subdirectory, so that the main function can collect them and produce a comprehensive description of the required environment (section 7.3).

7.3. The Runtime Environment

CMake provide means to set environment variables, but only for limited and well defined cases. What we need, instead, is to be able to declare in the configuration files the variables that should be set in the runtime environment. These declarations must then be used to prepare the environment.

To circumvent this limitation, we developed a simple and portable tool in Python to manipulate the environment, either by wrapping the call to commands (like the Unix command `env`) or by setting the environment of the interactive shell. The tool uses a description of the changes expressed in XML using operations like *set*, *append*, *prepend* and *remove*. The main function, invoked from the project `CMakeLists.txt`, generates a project-specific XML file corresponding to the list of operations declared by the subdirectories via the function `gaudi_env` (section 7.2).

In addition to the user defined environment variables, we need to define the content of some standard variables: `PATH` (search path for executables and scripts), `LD_LIBRARY_PATH` (search path for libraries and modules[5]) and `PYTHONPATH` (search path for Python modules).

The entries to be added to these variables are of three types: user defined, conventional and detectable. The user defined values are added using the function `gaudi_env`. The conventional

[5] On Windows the varible `LD_LIBRARY_PATH` is ignored and `PATH` is used for libraries too.

values are defined by the structure of the Gaudi-based projects, where knowing the location of the project is enough to infer the locations of libraries, executables, etc. The detectable entries are those implied by the build procedure, i.e. the access path to libraries and command used or located during the build.

In some cases we need to add to the environment variables for the correct operation of some external packages. Usually it happens with external packages that require custom *find* CMake modules, so we can define some special variables that are detected by our CMake framework and used to extend the environment (e.g. `<Package>_PYTHON_PATH`).

All the operations on the environment required by a project are exported (section 7.4) to be used by derived projects.

Since some commands used during the build require that the environment is correctly set up to properly work, all the commands we are invoking are executed through a Python script that wraps them in the environment defined in the XML file described above.

7.4. Exporting Subdirectories and Targets

CMake provides a useful mechanism to make the targets of a project accessible to others. This feature allows cleaner and easier coding of CMake configurations, because all the informations that CMake needs to know when using locally build commands or libraries are exported to the other projects.

Unfortunately, the standard export feature has two limitations that make it impossible for us to use it:

- targets that are exported cannot be overridden by a project importing them,
- export files must contain a consistent set of exported targets, i.e. all the dependencies have to be met within a single export file.

While these constraints can make perfectly sense in the use cases considered in the development of CMake, we need to bypass them to be able to override subdirectories as we need and as we are used to with CMT.

The solution adopted is to re-implement in our framework the export feature of CMake, using as much as possible the same syntax and conventions used by the standard CMake export.

Using a custom export mechanism allows us to generate one export file per subdirectory, so that if one is overridden in the derived project, the corresponding export file is not imported, while all the others are, to have a consistent and complete set of targets.

Since we are generating the export files ourselves, we can add to them all the informations that need to be propagated from the subdirectories of one project to the subdirectories of another project, but that are not foreseen by the standard CMake export, like dependencies, environment variables etc.

Of course, there are drawbacks in relying on a custom export mechanism: we have more code to maintain and we need to keep our export files synchronized with the developments of CMake. Luckily enough, the export format of CMake seems to be pretty stable (since version 2.6.0 until now, version 2.8.6), and the advantages brought by this approach are worth much more than the cost of some few more lines of CMake code to maintain.

7.5. The Toolchains

CMake is used, in most cases, to check that the system used to build a project meets some requirements, then to pass to the underlying build system the appropriate options to use the required libraries. For LHCb, as for most HEP experiments, it is important to use our own versions of the external libraries and tools (including the C++ compiler).

It is possible to tell CMake to use a compiler that is not the default one, and it is also possible to modify where CMake looks for headers, executables and libraries. One way is through

environment variables or command line arguments, but it becomes soon unmanageable if we need to pass several options. In this case it is much easier to use *toolchain* files. When CMake gets a toolchain file on the command line, it parses it before any other operation, so that it can be used to configure low level settings, like the location of the compiler. Afterwards it proceeds to the parsing of the local `CMakeLists.txt`.

To implement the toolchains needed to reproduce the custom configurations we have been using with CMT, we created a common toolchain file that defines some simple functions. Then we added one toolchain file per version of the configuration, i.e. set of versions of the external packages. These specific toolchain files include the common toolchain file to be able to call the functions defined there. A reduced example of a specific toolchain file is the following:

```
cmake_minimum_required(VERSION 2.8.5)
include(${CMAKE_CURRENT_LIST_DIR}/heptools-common.cmake)

set(heptools_version 62b)

# Application Area Projects
LCG_AA_project(COOL COOL_2_8_13)
LCG_AA_project(CORAL CORAL_2_3_22)
LCG_AA_project(RELAX RELAX_1_3_0f)
LCG_AA_project(ROOT 5.32.02)

# Compilers
LCG_compiler(gcc43 gcc 4.3.5)
LCG_compiler(gcc46 gcc 4.6.2)

# Externals
LCG_external_package(Boost 1.48.0)
LCG_external_package(bz2lib 1.0.2)
LCG_external_package(Python 2.6.5p2)
[...]

# Prepare the search paths
LCG_prepare_paths()
```

At the end of the specific toolchain there is the mandatory call to a special function, which uses the informations declared via the calls to the other functions to extend the CMake search paths, for example via variables like `CMAKE_PREFIX_PATH`, so that the *find* modules look in the system standard directories only after the locations where our custom builds are installed.

7.6. From CMT to CMake

To make the transition smooth, the CMake-based framework has been written such that the installation step creates a directory structure that is equivalent to the one produced by the build with CMT. In this way, as long as we keep the CMT configuration files in the source tree, a project can be migrated to CMake always being accessible via CMT. Of course, the migration to CMake has to follow the order of dependencies between projects, from the most base one to the most derived one, because a project using CMake cannot use a project built with CMT.

To simplify the conversion of CMT configurations to CMake ones, we developed a Python script that analyzes the CMT configuration (via the CMT querying functionalities) and produces the corresponding CMake configuration file.

8. Conclusions

Although the philosophy behind CMake is quite different from that of CMT, it has been possible to implement a configuration and build framework equivalent to the one in use.

The new framework, keeping the same functionalities of the old one, has some advantages. The configuration files are simpler and cleaner than the old ones, thus easier to maintain. With the power of the CMake language, it is possible to tune the build in a finer way. The performances improved a lot too, as summarized in Table 2.

The main problems encountered in the implementation of the CMake-based framework are due to the fact that CMake was not meant to address the needs of our peculiar runtime environment (which relies on the variables `PATH` and `LD_LIBRARY_PATH`) and the special requirements we have (overriding of subdirectories). We do not need to ask CMake developers to support the configuration of the runtime environment (our custom tools are good enough), but our build system could be simplified a lot if CMake is extended to allow the exportation of custom properties and variables, and to allow the override of targets and subdirectories.

The new framework is essentially complete, apart from some polishing needed before using it in production. While completing the polishing, LHCb is planning to run further tests on this new framework during Summer 2012, in order start the migration of the projects during the LHC shutdown of 2013.

Table 2. Performances of the CMake-based build system with respect to three versions of CMT (the one used in production and the latest two). The measurements has been obtained compiling the project Gaudi (21 packages) on a machine with 8 2.33GHz Xeon processors, accessing only local files. The measurements include full sequential build, full parallel build (20 processes, compiling on a distcc cluster), no-op sequential builds (i.e. rebuild immediately after a full build) and preparation of the environment. It should be noted that the time required by the environment configuration in CMT depends heavily on the number of packages and latency of the filesystem, while with CMake it is constant. The latest version of CMT (v1r25) has been released just before CHEP2012 and features a sensible performance improvements, mainly due to a reduced stress on the filesystem.

	CMake	CMT		
Operation		v1r20p20090520	v1r24	v1r25
full sequential build	31m 38s	36m 24s	36m 0s	30m 2s
no-op sequential build	0m 12s	2m 25s	1m 4s	0m 40s
full parallel build	4m 40s	9m 19s	9m 14s	8m 6s
environment configuration	0.11s	0.45s	0.24s	0.10s

References

[1] Barrand G *et al.* 2001 *Comput. Phys. Commun.* **140** 45–55
[2] Arnault C 2000 CMT: A software configuration management tool *International Conference on Computing in High-Energy Physics and Nuclear Physics (CHEP 2000), Padova, Italy, 7-11 Feb 2000*
[3] CMT Home Page URL `http://www.cmtsite.org/`
[4] CMake - Cross Platform Make URL `http://www.cmake.org/`
[5] KDE Project URL `http://kde.org/`
[6] Agostinelli S *et al.* (GEANT4) 2003 *Nucl.Instrum.Meth.* **A506** 250–303
[7] Geant4: A toolkit for the simulation of the passage of particles through matter URL `http://cern.ch/geant4`
[8] Brun R and Rademakers F 1997 *Nucl.Instrum.Meth.* **A389** 81–86
[9] ROOT — A Data Analysis Framework URL `http://root.cern.ch/`
[10] Building ROOT with CMake URL `http://root.cern.ch/drupal/content/building-root-cmake`
[11] Python Programming Language – Official Website URL `http://python.org/`