ATLAS software configuration and build tool optimisation

Grigory Rybkin
(on behalf of the ATLAS Collaboration)
Laboratoire de l’Accélérateur Linéaire, Université Paris-Sud, CNRS/IN2P3, Orsay, France
E-mail: Grigori.Rybkine@cern.ch

Abstract. ATLAS software code base is over 6 million lines organised in about 2000 packages. It makes use of some 100 external software packages, is developed by more than 400 developers and used by more than 2500 physicists from over 200 universities and laboratories in 6 continents. To meet the challenge of configuration and building of this software, the Configuration Management Tool (CMT) is used. CMT expects each package to describe its build targets, build and environment setup parameters, dependencies on other packages in a text file called requirements, and each project (group of packages) to describe its policies and dependencies on other projects in a text project file. Based on the effective set of configuration parameters read from the requirements files of dependent packages and project files, CMT commands build the packages, generate the environment for their use, or query the packages. The main focus was on build time performance that was optimised within several approaches: reduction of the number of reads of requirements files that are now read once per package by a CMT build command that generates cached requirements files for subsequent CMT build commands; introduction of more fine-grained build parallelism at package task level, i.e., dependent applications and libraries are compiled in parallel; code optimisation of CMT commands used for build; introduction of package level build parallelism, i.e., parallelise the build of independent packages. By default, CMT launches NUMBER-OF-PROCESSORS build commands in parallel. The other focus was on CMT commands optimisation in general that made them approximately 2 times faster. CMT can generate a cached requirements file for the environment setup command, which is especially useful for deployment on distributed file systems like AFS or CERN VMFS. The use of parallelism, caching and code optimisation significantly—by several times—reduced software build time, environment setup time, increased the efficiency of multi-core computing resources utilisation, and considerably improved software developer and user experience.

1. Introduction
ATLAS [1] is a general-purpose particle physics experiment at the Large Hadron Collider (LHC) at CERN, the European Laboratory for Particle Physics near Geneva. More than 2,500 physicists from over 200 universities and laboratories in 6 continents participate in the experiment. A large software suite was developed to simulate and analyse data from the experiment. The software code base is over 6 million lines written mainly in the C++, Fortran, and Python languages and organised in over 2000 packages. Configuring and building such software releases is a challenging task already. ATLAS uses the Configuration Management Tool (CMT) [2] to address the problem. CMT formalizes software production, especially configuration management, around a package-oriented principle. CMT permits description of the configuration requirements through
text files, called requirements and projects files, and several environment variables, and deduces from the description the effective set of configuration parameters needed to operate the packages, i.e. to build, use, or query them. The CMT build performance is crucial because of the size of the software and since ATLAS maintains or develops 4–5 full software release branches and more than 5 relatively smaller physics analysis releases carrying out their integration builds for several platforms (with different compiler versions, optimised, for debugging) on a daily basis [3]. The performance of the CMT setup and querying commands is very important and may even be critical since they are used when launching every data production or analysis software job, in particular, on Worldwide LHC Computing Grid (WLCG) sites, by various other tools, e.g., [4], [5], [6], as well as by the software users and developers interactively. In this paper, we describe the work on CMT optimisation and present the results achieved.

2. Build time optimisation

2.1. How CMT works

The build of a CMT package is typically done with the "(cmt) make" command which

- invokes the CMT commands to generate Makefiles—according to the description in the requirements files—that, in turn, may contain CMT command invocations
- passes Makefiles to Make [7] in order to do the package build—create the libraries, applications, and perform the actions (commands)

Such CMT command invocations may be quite numerous, e.g., for the AtlasEvent project with 358 packages the number is 6320 invocations. The average number of CMT command invocations being about 18 per package. Originally, upon each CMT command invocation, CMT read the requirements files of all the used packages.

2.2. New package build procedure

The package build procedure was re-designed so as to introduce a CMT command that reads the effective configuration and, based on that, generates the cached requirements files that do not reference any other requirements files and contain all the necessary information for the subsequent CMT build command invocations. This reduced the number of the requirements files reads to (as a rule) one per used package. In the context of ATLAS software, on average, the build command reading the package requirements files spends ≈ 1.0s, while reading the cached requirements file it spends ≈ 0.1s, i.e. ~ 10 times less. Also, by combining some of the previously used commands in new commands, the number of CMT build command invocations was reduced. As a result, the CMT build time overhead decreased ≈ 2.3–3.0 times.

2.3. More build parallelism

Originally, the way to build a CMT project was to run the “cmt broadcast make” command in the package with dependencies on all the project packages. This command builds one package at a time, taking into account packages dependencies. We introduced a new so-called BCAST build mode in which the build is run with the simple "(cmt) make" command, is performed on several independent packages at the same time and, therefore, is parallelised at package level. Depending on the project structure and number of processors available, this mode may reduce the full build time by several times compared to the usual “cmt broadcast make”.

2.4. Even more build parallelism

We further modified package build procedure so as to compile even dependent applications and libraries in parallel whenever possible. The dependencies are imposed at link stage only. This allows for more fine-grained parallelism at package task level and enhanced use of multi-core machines resources.
2.5. Default setup
The default CMT setup was optimised in two ways. Now CMT runs Make with the \texttt{-j NPROCESSORS} option launching \texttt{NPROCESSORS} build processes in parallel, making good use of multi-core machines resources. The so-called QUICK mode is enabled which means that all the Makefiles are only regenerated if any requirements file changes. The mode does not affect source the files dependencies calculation which are always recalculated if a file or any dependency changes. This ensures faster and more efficient development cycle since most of the time only the build commands are run, all the Makefiles are generated once and are only regenerated when necessary.

3. CMT commands optimisation
Using program profiling tools [8], we identified the most time-consuming parts. Common for all the commands was a function extensively used in the parsing of the requirements files which we re-wrote making use of a standard library function. Another improvement that affected virtually all the commands was to introduce a cache in order to optimise the use of a function accessing the file system. The algorithm of the heavily used command that generates source files dependencies was revisited, in particular, replacing the use of one standard library function with another standard library function performing considerably better. The results are the following:

- running the setup is \(\approx 1.7\) times faster, the number of \texttt{stat}, \texttt{lstat} calls is reduced by \(\approx 2.5\) times—addresses the problem of long setup time at WLCG sites, in particular
- querying commands like “cmt show ...” are \(\approx 2\) times faster
- “cmt build dependencies” command is \(\approx 4\) times faster

In addition, the enhanced mechanism of source files dependencies generation at compilation time was implemented, giving a full build time gain of \(\approx 10\%\) (for C/C++ based projects). On average, the time gain is \(\approx 1\) hour for each ATLAS full release branch.

4. Runtime environment setup
In order to cache the environment setup, the \texttt{-requirements} option was implemented so that the “cmt -requirements setup” command generates a standalone, or cached, requirements file that can then be used with the “cmt setup” command for the environment setup script generation. This is especially useful for deployment on distributed file systems like AFS or CERN Virtual Machine File System (CERN VMFS) [9].

5. Conclusions
Significant build performance optimisation was achieved thanks to the CMT build procedure re-design based on making use of caching and parallelism. The CMT overhead decreased by \(\approx 5\) times and represents less than 1–5\% of full build time. The enhanced parallelism at both package task and package levels allows CMT to use the multi-core machines resources more efficiently and reduces the full build time by several times. Thanks to the considerable code optimisation, most of the CMT commands, in particular, the configuration querying commands and the runtime environment setup became \(\approx 2\) times faster. The latter can also be cached allowing for further optimisation. These optimisations allow ATLAS to successfully build, develop, and use its software, and considerably improve software developer and user experience.

References
[1] ATLAS experiment http://cern.ch/atlas/
[2] Arnault C 2001 Experiencing CMT in software production of large and complex projects Proc. Int. Conf. on Computing in High Energy and Nuclear Physics CHEP 2001 (Beijing, China) http://www.cmtsite.net/
[3] Luehring F, Obreshkov E, Quarrie D, Rybkine G and Undrus A 2010 Organization and management of ATLAS nightly builds J. Phys.: Conf. Series 219 042045, Undrus A 2012 Evolution of the ATLAS nightly build system J. Phys.: Conf. Series 396 052070

[4] Obreshkov E et al. 2008 Organization and management of ATLAS offline software releases Nucl. Instrum. Meth. A 584 244–251, Albrand S et al. 2010 Organization, management, and documentation of ATLAS offline software releases J. Phys.: Conf. Series 219 042012

[5] Arnault C, De Salvo A, George S and Rybkine G 2004 The deployment mechanisms for the ATLAS software Proc. Int. Conf. on Computing in High Energy and Nuclear Physics CHEP 2004 (Interlaken, Switzerland)

[6] Rybkin G 2012 ATLAS software packaging J. Phys.: Conf. Series 396 052063

[7] GNU Make http://www.gnu.org/software/make/

[8] GNU Gprof from GNU Binutils http://www.gnu.org/software/binutils/

[9] Callgrind http://www.valgrind.org/

[9] De Salvo A et al. 2012 Software installation and condition data distribution via CernVM FileSystem in ATLAS J. Phys.: Conf. Series 396 032030