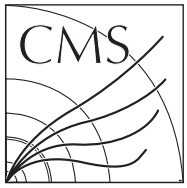
Available on CMS information server

CMS CR -2009/079



The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

12 May 2009 (v2, 14 May 2009)

Optimization of the CMS software build and distribution system

S Muzaffar, G Eulisse

Abstract

CMS software consists of over two million lines of code actively developed by hundreds of developers from all around the world. Optimal build, release and distribution of such a large-scale system for production and analysis activities for hundreds of sites and multiple platforms are quite a challenge. Its dependency on more than one hundred external tools makes its build and distribution more complex. We describe how parallel building of the software and minimalizing the size of the distribution dramatically reduced the time gap between software build and installation on remote sites, and how producing few big binary products, instead of thousands small ones, helped finding out some integration and runtime issues of the software.

Presented at *Computing in High-Energy and Nuclear Physics (CHEP), 21-27 March 2009,Prague,Czech Republic,15/05/2009*

Optimization of the CMS software build and distribution system

S Muzaffar, G Eulisse

Northeastern University, Boston, MA, USA

Shahzad.Muzaffar@cern.ch, Giulio.Eulisse@cern.ch

Abstract. CMS software consists of over two million lines of code actively developed by hundreds of developers from all around the world. Optimal build, release and distribution of such a large-scale system for production and analysis activities for hundreds of sites and multiple platforms are quite a challenge. Its dependency on more than one hundred external tools makes its build and distribution more complex. We describe how parallel building of the software and minimalizing the size of the distribution dramatically reduced the time gap between software build and installation on remote sites, and how producing few big binary products, instead of thousands small ones, helped finding out some integration and runtime issues of the software.

1. CMS software build and distribution

The Compact Muon Solenoid (CMS) experiment on the Large Hadron Collider (LHC) at CERN [1] has been using SCRAM [2] (Software Configuration and Management) to build and release its software CMSSW. SCRAM resolves the issues of external tools' configuration, software build, runtime environment and installation [2]. It provides an easy and simple way for software developers to build subsets of their software without rebuilding everything. It basically transforms user defined build rules, provided via a so-called BuildFile [2], into a Makefile and runs gmake [3] to actually build the user code. To avoid parsing of unmodified BuildFiles, SCRAM generates internal caches for future use. Figure 1 shows an overview of the SCRAM workflow.

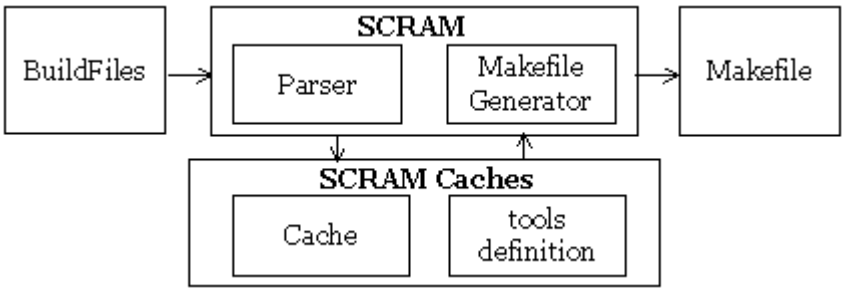


Figure 1. SCRAM workflow

CMSSW depends on more than 100 external tools [4]. For consistent build and distribution of these external tools for hundreds of sites and multiple platforms, CMS has been using PKGTOOLS and apt-

get [5]. PKGTOOLS is a collection of homemade scripts, which help building external tools from sources using RPM as package manager and apt-get as distribution manager.

1.1. Software build issues

Active development of CMSSW, which consists of over two million lines of code divided in 1100 packages [4], exposed scalability issues in earlier versions of SCRAM. To compile a few source files in a developer area, the overhead introduced by SCRAM became larger than the time needed to actually compile the code itself.

For caching the information of over 2400 CMS build products (shared libraries, plugins and executables), SCRAM was generating over 110MB of internal cache and well over 70MB of Makefiles. Its runtime memory usage went over 800MB, which was killing its performance on many machines.

Due to such a heavy usage of system resources, it became nearly impossible to build even a single CMSSW package in reasonable time. Things got worst when developers worked in their AFS area on public machines shared by other developers, which is the case for most of CMSSW developers. With no parallel build support, it was taking more than 10 hours to build full CMSSW release.

1.2. Distribution issues

Rapid changes in external tools' versions exposed a few issues with the way PKGTOOLS [5] was working. It was only building external tools for which the SPEC files were modified. So if SPEC files for other external tools, which depended on these new SPEC files, were not changed as well, then the packages "higher up" in the hierarchy were not rebuilt. This causes multiple versions of the same external tool downloaded and installed for single version of CMSSW. Figure 2 shows that two versions of *ToolY* were installed for the installation of *ToolA* version V2 because *ToolC* was not rebuilt using the new *ToolY*.

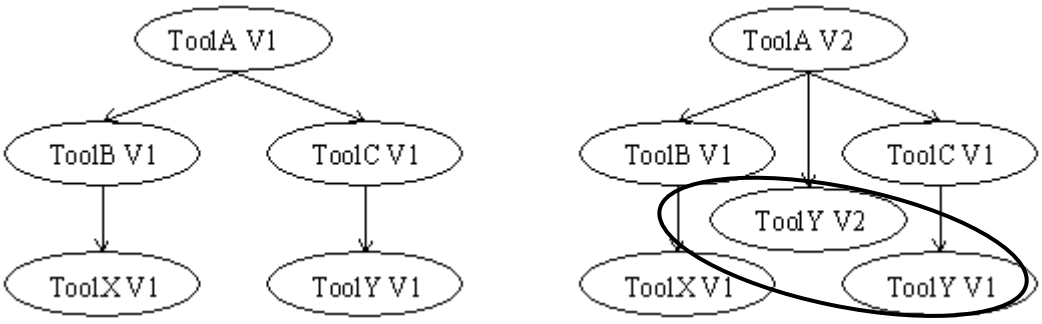


Figure 2. PKGTOOLS: Multiple versions of same tools shipped

2. CMS software build and distribution optimizations

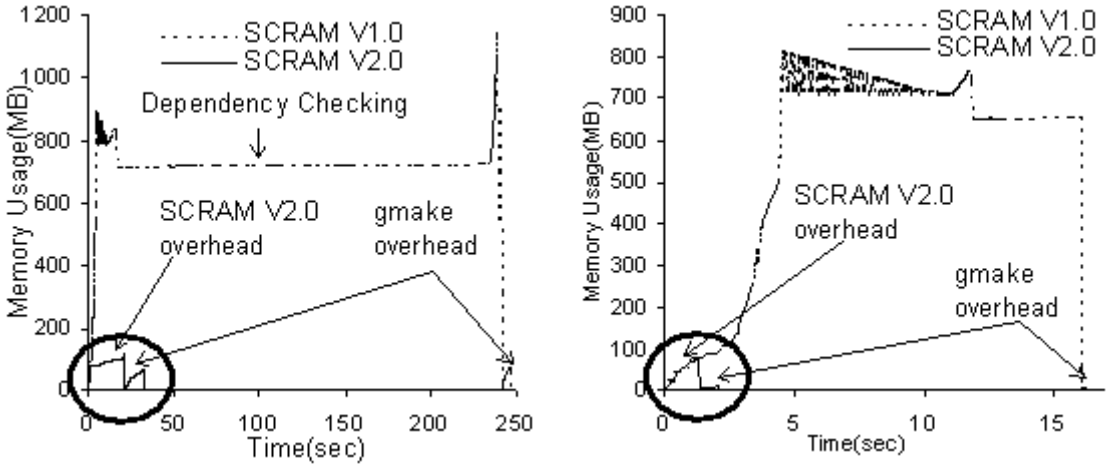
With all these build and distribution overheads, it was clear that CMS should improve or change the software development tools it is using. In addition, not being able to build CMSSW in parallel was a big disadvantage too. Instead of testing new tools and asking hundreds of developers to migrate to a new interface, we identified the areas where our current tools have problems.

2.1. SCRAM and build rules optimizations

SCRAM, being written in PERL, was taking a lot of time finding dependencies for CMSSW packages and its external tools. Its large internal caches and Makefile were also slowing down its performance. Improvements in these areas dramatically reduced SCRAM overhead, these are described in detail below.

2.1.1. Dependency Checking. For a full CMSSW release, SCRAM version V1.0 was taking around 220s for calculating dependency information. As SCRAM uses gmake to actually compile and build,

so there was no reason why SCRAM should solve all the dependencies itself. So we fixed our build rules and moved all the dependency tracking logic from SCRAM in to gmake. This turned out to be a big performance gain. Figure 3 shows SCRAM version V1.0 and V2.0 overheads for a full CMSSW release and for a typical user development area with few CMSSW packages in it.

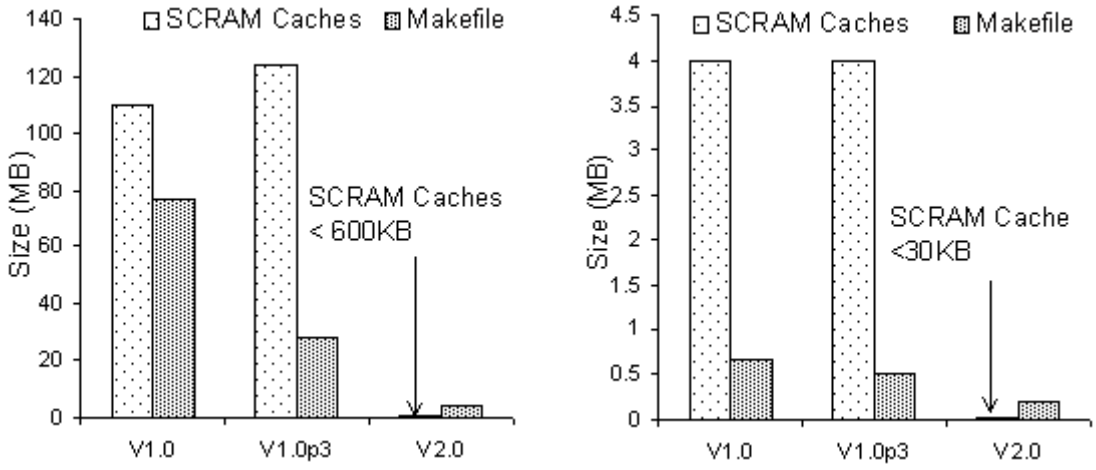


CMSSW Full Release Developer Area

Figure 3. SCRAM version V2.0 runtime memory and time overhead

2.1.2. SCRAM caches. Over 800MB of runtime memory usage was due to large caches generated by SCRAM in order to avoid parsing of unchanged BuildFile. Looking in these cache files we discovered that there was duplicate information and a lot of things were actually not needed. So cleaning up these caches, removing product dependency information and saving only minimal information shrunk cache size from 110MB to 20MB. To keep disk usage small we saved these cache files in compressed form.

2.1.3. Makefile size. The build rules for CMSSW were not optimized and SCRAM V1.0 was generating over 30KB of Makefile fragments for each CMSSW build product. The result of that was a Makefile with size of over 70MB. We rewrote and optimized CMSSW build rules, which resulted in a very compact Makefile of size less than 4MB for a full CMSSW release.



Full CMSSW Release Developer Area

Figure 4. SCRAM caches and Makefile overhead

2.1.4. Parallel build support. CMSSW build rules used with SCRAM version V1.0 were not allowing us to build CMSSW in parallel. A major effort was done to rewrite build rules in such a way that we can make use of the parallel build option of gmake ("gmake –j") [3]. This change dramatically reduced CMSSW build time. Figure 5 shows that CMSSW build time reduced to 90 minutes from around 10 hours on an 8 core machine.

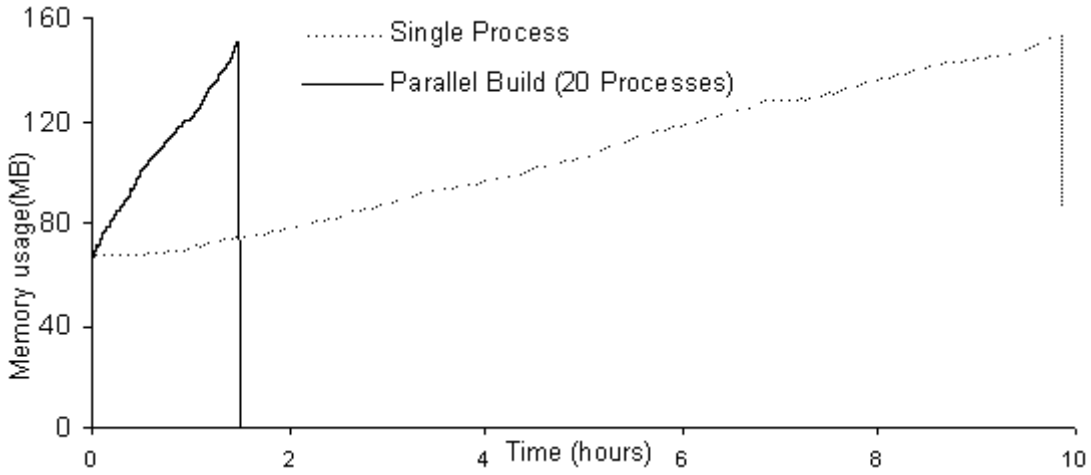


Figure 5.CMSSW full release build time on an 8 core machine

2.1.5. Big shared libraries. CMSSW build generates thousands of small shared-libraries and plugins, which affect its runtime performance [6]. Introducing new build rules for building CMSSW also allowed us to build a few big-shared libraries instead of thousands of small ones. This enabled us doing different profiling and code coverage tasks [7]. It also helped us identifying the issues like

- C++ template code replicated in hundreds of shared libraries and plugins.
- Copying of source files between different CMSSW packages resulted in same symbol definition in many libraries and plugins, which meant unpredictable runtime behavior.

2.2. PKGTOOLS optimization

Installation of multiple base level external tools for a single CMSSW release was clearly a problem in the PKGTOOLS workflow. Also not being able to build independent tools in parallel was slowing down build time. So the PKGTOOLS scripts were re-written and their logic was changed to automatically rebuild all top-level tools for which a base tool is changed. Building independent external tools in parallel saved a lot of build time and we have managed to build all the externals tools needed for CMSSW in less than 4 hours.

3. Conclusions

All these optimization and cleanup allowed CMSSW developers to spend more time on their code instead of waiting for the compilation to finish. Now we are able to build multiple *Integration Builds* each day for all the supported platforms for each CMSSW release cycles [4]. Clean distribution of CMSSW and its externals resulted in faster download and installation of new CMSSW releases on remote sites.

4. Acknowledgements

We would like to thank members of CMS software tools development project for their support, help and resources they have provided us. Special thanks to all CMS software developers and CMS site managers for their positive feedback, which helped us to identify these issues. Thanks to National Science Foundation (NSF) of the United States of America for their support.

References

[1] CMS Collaboration, “Technical proposal”, CERN/LHCC 94-38, 1994

[2] Wellisch, J., Williams, C. and Ashby, S., “SCRAM: Software configuration and management for the LHC Computing Grid prohect”, ”, Computing in High-Energy and Nuclear Physics (CHEP), La Jolla, California, 2003

[3] Gmake home page, http://www.gnu.org/software/make

[4] Pfeiffer, A. et al, “CMS software Infrastructure Tools”, IEEE/NSS, 2008
Lange, D., “Software Integration and Development Tools in CMS”, Computing in High-Energy and Nuclear Physics (CHEP), Prague, 2009

[5] Argiro, S., Elmer, P., Eulisse, G. and Tuura, L., “CMS packaging system or: how I learned stop worrying and love RPM spec files”, Computing in High-Energy and Nuclear Physics (CHEP), Victoria, BC, 2007

[6] Eulisse, G., Tuura, L. and Elmer, P., “HEP C++ meets reality”, Computing in High-Energy and Nuclear Physics (CHEP), Prague, 2009

[7] Elmer, P., Eulisse, G., Tuura, L. and Innocente, V., “CMS software Performance Strategies”, Computing in High-Energy and Nuclear Physics (CHEP), Prague, 2009