Optimization of the CMS software build and distribution system

To cite this article: S Muzaffar and G Eulisse 2010 J. Phys.: Conf. Ser. 219 042047

View the article online for updates and enhancements.

Related content
- Deployment of the CMS software on the WLCG Grid
  W Behrenhoff, C Wissing, B Kim et al.
- CMS data quality monitoring web service
  L Tuura, G Eulisse and A Meyer
- Rise of the Build Infrastructure
  Giulio Eulisse, Shahzad Muzaffar, David Abdurachmanov et al.

Recent citations
- Rise of the Build Infrastructure
  Giulio Eulisse et al
- Explorations of the viability of ARM and Xeon Phi for physics processing
  David Abdurachmanov et al
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 hundred external tools make its build and distribution more complex. We describe how parallel build of software and minimal distribution size dramatically reduced the time gap between software build and installation on remote sites, and how producing few big binary products, instead of thousands of small ones, helped finding out the integration and runtime issues.

1. CMS software build and distribution

The Compact Muon Solenoid (CMS) [1] experiment at the Large Hadron Collider (LHC) at CERN 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, run time environment and installation. It provides an easy and simple way to software developers to build subset of their software without rebuilding every thing. It basically transforms user defined build rules, provided via BuildFile, in to 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 SCRAM workflow.

![SCRAM workflow diagram](image)

**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 [5] and apt-get [6]. PKGTOOLS is collection of homemade scripts, which help building external tools from sources using RPM [7] 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 SCRAM version V1.0 scalability issues. To compile few source files, in a developer area, SCRAM overhead became more than the time needed to actually compile the code.

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

1.2. Distribution issues

Rapid changes in external tools versions exposed few issues with the way PKGTOOLS was working. It was only building external tools for which the SPEC files were modified. So if SPEC files for external tools, which depend on these new SPEC files, were not changed then those were not re-build. This caused multiple versions of same external tool downloaded in order to install for each single version of CMSSW. Figure 2 shows that two versions of ToolY were needed to install ToolA version V2 because ToolC was not re-build using 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 was using. Instead of testing new tools and asking hundreds of developers to migrate to new interface, we search for the areas where the tools we used had problems. Also not able to build CMSSW in parallel was a big disadvantage too.

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.

2.1.1. Dependency Checking. For a full CMSSW release, SCRAM version V1.0 was taking over 200 seconds for calculating dependency information. As SCRAM uses gmake to actually compile and build, there was no reason why SCRAM should solve all the dependencies itself. So we fixed our build rules and move 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.
2.1.2. SCRAM caches. Over 800 MB 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 were 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 shrink cache size to 600 KB from 110 MB as seen in Figure 4. To keep disk usage small we saved these cache files in compressed form.

2.1.3. Makefile size. Build rules for CMSSW were not optimized and SCRAM V1.0 was generating over 30 KB of Makefile fragment for each CMSSW build product. Result of that was over 70 MB of Makefile. We rewrote and optimized CMSSW build rules, which resulted in very compact Makefile of size less than 4 MB for a full CMSSW release see Figure 4. An intermediate SCRAM version V1.0p3 was released with only build rules optimized.

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 “gmake –j” [3] parallel build option. This change dramatically reduced
CMSSW build time. Figure 5 shows that CMSSW build time reduced to 90 minutes from around 10 hours on a 8 core machine.

![Figure 5: CMSSW full release build time on a 8 core machine](image)

2.1.5. Big shared libraries. CMSSW build generates thousands of small shared-libraries and plugins, which affect runtime performance [8]. New build rules for building CMSSW also allowed us to build few big-shared libraries instead of thousand small one, which helped us better understanding of CMS software [9]. It also helped us identifying 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 bug in PKGTOOLS workflow. Also not being able to build non-dependant tools in parallel was slowing down build time. The PKGTOOLS scripts were re-written and logic was changed to automatically rebuild all top-level tools for which a base tool is changed. Building non-dependent 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 compilation to finish. Now we are able to build multiple Integration Builds per day for all the supported platforms for each CMSSW release cycle [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. Special thanks to all CMS software developers and CMS site managers for their positive feedback, which helped us identifying 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 project, 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] APT home page, http://apt-rpm.org

[7] RPM home page, http://www.rpm.org

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

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