University of Groningen

A Tool for Optimizing the Build Performance of Large Software Code Bases

Telea, Alexandru; Voinea, Lucian

Published in:
CSMR 2008: 12TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2008

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Telea, A., & Voinea, L. (2008). A Tool for Optimizing the Build Performance of Large Software Code Bases. In K. Kontogiannis, C. Tjortjis, & A. Winter (Eds.), *CSMR 2008: 12TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING* (pp. 323-325). University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 20-06-2022

A Tool for Optimizing the Build Performance of Large Software Code Bases

Alexandru Telea*
Institute for Mathematics and Computer Science
University of Groningen, Netherlands
a.c.telea@rug.nl

Lucian Voinea
SolidSource BV
Eindhoven, Netherlands
lucian.voinea@solidsource.nl

Abstract

We present Build Analyzer, a tool that helps developers optimize the build performance of huge systems written in C. Due to complex C header dependencies, even small code changes can cause extremely long rebuilds, which are problematic when code is shared and modified by teams of hundreds of individuals. Build Analyzer supports several use cases. For developers, it provides an estimate of the build impact and distribution caused by a given change. For architects, it shows why a build is costly, how its cost is spread over the entire code base, which headers cause build bottlenecks, and suggests ways to refactor these to reduce the cost. We demonstrate Build Analyzer with a use-case on a real industry code base.

1. Introduction

Systems of millions of lines of C (or C++) code, developed by teams of hundreds of people for several platforms, are commonplace in embedded, automotive, and electronics industries. Although modular, such systems face a scale problem: whenever a header file is modified, all the source, library, and executable files which depend, directly or not, on it, must be rebuilt. Hence, even small changes to certain files can cause huge rebuild times. This slows down development, debugging, and testing speed for systems maintained by large teams working worldwide around the clock. It is important to know how costly rebuilds could be avoided, *e.g.* by header refactoring, if possible.

We present here Build Analyzer, a commercial tool [4] we created to assist developers and architects in improving the build performance of large C code bases. For developers, Build Analyzer provides direct feedback on the build cost when a given source or header file changes, letting them decide if they want to make that change visible to other team members. For architects, Build Analyzer shows how the build cost is spread over an entire system architecture, emphasizes files which cause build bottlenecks, and suggests how to refactor the header-set to improve build time.

2. Tool Overview

The architecture of our tool is shown in Figure 1. A *dependency extractor* parses all project source and header files kept in a CM/Synergy repository, and also the system headers (*e.g.* `stdio.h`), and extracts several facts. These are saved in a MySQL database. Next, the tool offers several graphical *views* to assist users answering specific questions. We detail these modules next.

2.1. Data Extraction and Cost Model

Data extraction has several parts. First, we extract file dependencies: executables → dynamic-libraries → object files → source files → recursively-included headers. Header-header and header-source relations are extracted by parsing their C code. For this, we can use the CScout commercial parser [5] or the `gcc -M` compiler option. We have also tried other tools (*e.g.* Makepp [3], CScope [1], and Ctags [2]), but none provided the complete set of qualified file dependencies we need. *CScout* is better than `gcc -M` as it also provides header symbol information, *i.e.* all global symbols (functions, data types, and macros), which we use in our refactoring analysis (Sec. 2.2). However, gcc is more robust. Still, both *CScout* and `gcc -M` take several hours to examine our entire code base, which is quite slow. Next, we parse the so-called CM/Synergy configuration records, using an own parser, to extract library-source and executable-library relations, and also the change frequency of each file over the entire project history.

Let us detail our cost model: All files f have a build cost $BC(f)$ and a build impact $BI(f)$. $BC(f)$ is the time spent to build f when any of its dependencies changes. Headers have zero build cost. For a source file s, $BC(s)$ equals the number of headers s includes, directly or not. Why this formula? We have repeatedly measured, for our studied code base, the actual build times for all sources (using the `timex` UNIX command on the gcc compiler), and found them almost exactly proportional to the total included header count of each source, regardless of header sizes. We explain this

978-1-4244-2157-2/08/$25.00 © 2008 IEEE

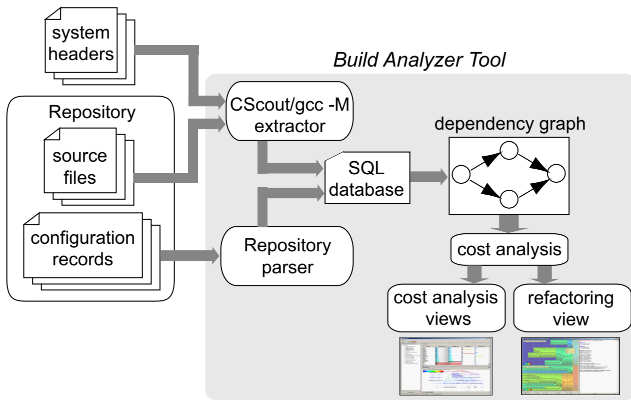

Figure 1. Build Analyzer tool architecture

since in the networked filesystem used, the I/O for reading a header dominates actual compilation costs. The build cost of libraries and executables is negligible, as linking is much faster than compilation. The build impact $BI(f)$ is the time spent to rebuild the *entire* code base when f changes. The build impact $BI(h)$ of a header h equals the sum of the build costs $BC(s)$ of all sources s using h, directly or not. For all other files, $BI = BC$, given that linking is very fast. The build impact is what we are interested in, the build cost being just used to compute the former.

Our tool computes the build costs and impacts of all files, by propagating BC values from sources through the dependency graph formed by the extracted data, and uses these values as explained next.

2.2. Build Cost Analysis Views

Build Analyzer provides several *views* to support different build-cost analyses. The main window resembles a classical IDE (Fig. 2. A code base can consist of several overlapping hierarchies, *e.g.* components, files, and interfaces. The *architecture* view can show any such hierarchy (mined by the extractor) using a tree widget. We show the the aggregated build cost and impact metrics over all elements in the architecture view, using blue-to-red colors.

The *cost-and-impact view* shows a table with several metrics for all files in the hierarchy element selected in the architecture view. Each file is a table row, and each metric (*e.g.* file name (A), impact (B), cost (D), change frequency (E)) is a column. We render the table using the table lens technique [6]: when zoomed in, we use draw the text; when zoomed out, each table row becomes a pixel row, and each cell becomes a pixel bar colored and scaled to reflect its value. Clicking the columns sorts the table on the underlying metric. Reducing tables to sets of graphs allows easily seeing outliers and value distributions. The 300 files in the file-view in Fig. 2, sorted on increasing build impact,

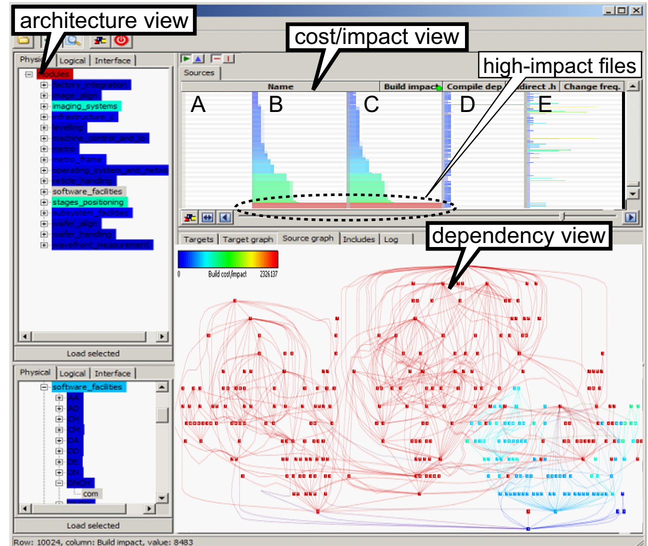

Figure 2. Build Analyzer's cost analysis views

show an exponential impact distribution. The few files at the bottom have a very high build impact compared to the rest. Developers can use this knowledge, *e.g.* by agreeing with team members when to change these files. Architects can focus refactoring efforts on these files, as they are the build 'bottlenecks' (Sec. 2.3). Another metric is the *dependency count* (C), or number of files depending (directly or not) on a given file. We noticed (as shown in Fig. 2) that the dependency count is almost proportional with the build impact for almost all targets in the code base. Computing the dependency count is over 10 times faster than computing the build impact, since the latter requires a complex graph traversal taking into account all system headers. Since our tool must respond in near-real-time to developers, we used the dependency count as a quick, yet good, approximation for the build impact.

At the bottom of Fig. 2, we have the *dependency view*. When the user selects a file f (*i.e.* a table row) in the cost-and-impact view, the dependency view shows the entire dependency graph having f at top and all files which depend on it, *i.e.* which need rebuilding when f is changed, below. These files can be colored by various metrics, *e.g.* the build cost, using a blue-to-red colormap. The graph shows how the 'change impact' actually propagates through the system. The colors help identifying bottlenecks, *i.e.* costly files.

The above views offer other assessments too:

- We can see if high-impact files also have a high *change frequency* metric. If so, these are real bottlenecks. If not, a high impact is not harmful by itself.
- We can interactively add or remove edges from the

dependency graph, and see how the build impact and other metrics change. This helps architects perform 'what if' scenarios to optimize dependencies for lowering the build cost.

2.3. Refactoring View

The build cost analysis views help users find where the build bottlenecks are, and possibly remove some, by editing dependencies. Still, high build costs can be caused by 'fat interface' headers which are included almost everywhere, and may change often. To lower build costs further, we must split these headers. We provide a *refactoring view* to assist this (Fig. 3. First, the user finds a fat header using the cost-and-impact view (Sec. 2.2). Next, Build Analyzer splits all symbols declared by that header H into two headers H_1 and H_2, in order to minimize the number of source files using symbols in *both* H_1 and H_2. Recursively applying the above yields a binary tree with H as root and each level as a possible refactoring of H into several headers H_i. Including these headers instead of H decreases build costs by decreasing the amount of included code and also the build impact (splitting fat interfaces). We call this the refactoring benefit. However, sources using large parts of a fat interface must include more headers after refactoring. We call this the refactoring cost. The refactoring view has three windows. The left windows show the refactoring tree, colored by the benefit and cost, respectively. Finding a good refactoring level amounts to looking for headers having low refactoring cost, high build-impact parents and low build-impact children. The right view details how the symbols will be split for the level chosen in the left window.

3. User Experience

The Build Analyzer tool is a commercial product [4] which has been used on a real-world embedded C code base of over 10 million lines, containing about 17000 source and 35000 header files. The current tool is already very useful, as very little was available before it. We are currently designing an improved version of the product, to address the collected user feedback, as follows. First, dependency and symbol extraction (currently done by CScout) will be massively accelerated to a few tens of seconds per file instead of hours, using incremental extraction and caching. Second, a simple, lightweight tool version should be integrated into developer tools (*e.g.* IDEs), to answer build impact questions on-the-fly in a matter of seconds. Finally, we plan to add build performance history recording and analysis, as our users mentioned that monitoring its evolution (*e.g.* degradation) is of great importance.

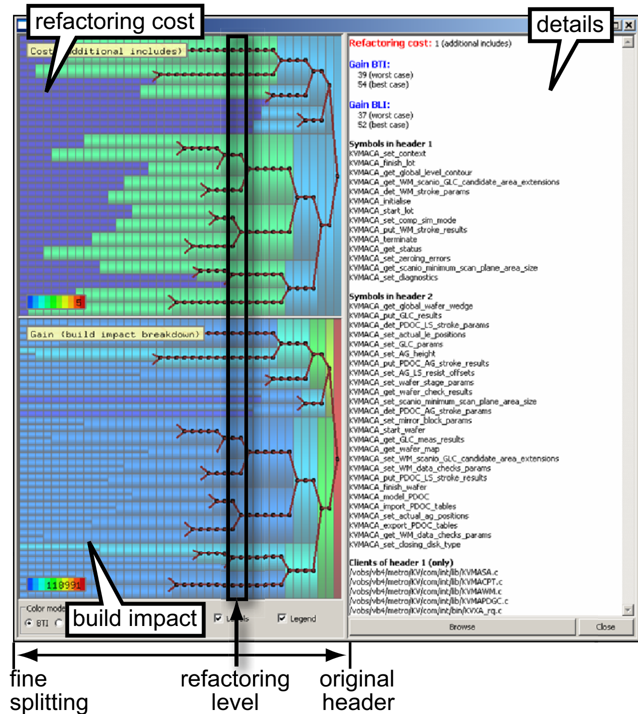

Figure 3. Refactoring view assists splitting 'fat' headers to decrease build costs

4. Conclusions

We have presented Build Analyzer, a tool that helps developers understand the causes of build bottlenecks in large C and C++ code bases. Interestingly enough, although the problem of long build times for such code is well-known, few tools exist that help users address it explicitly. In the future, we plan to refine Build Analyzer by adding more accurate build cost metrics, and also more intuitive, easy-to-use views to convey the tool feedback to the users.

References

[1] Bell Labs. The CScope code browser. 2007. `cscope.sourceforge.net`.

[2] Ctags Team. Ctags home page. 2007. `ctags.sourceforge.net`.

[3] G. Holt. Makepp home page. 2007. `makepp.sourceforge.net`.

[4] SolidSource BV. The Build Analyzer Tool. 2007. `www.solidsource.nl`.

[5] D. Spinellis. The CScout C extractor. 2007. `www.spinellis.gr`.

[6] A. Telea. Combining extended table lens and treemap techniques for visualizing tabular data. In *Proc. EuroVis*, pages 51–58. IEEE, 2006.