The Cost of the Build Tax in Scientific Software

Conference Paper · September 2011

DOI: 10.1109/ESEM.2011.54 · Source: DBLP

2 authors, including:

Lorin Hochstein
Netflix
53 PUBLICATIONS **740** CITATIONS

SEE PROFILE

The cost of the build tax in scientific software

Lorin Hochstein
USC/ISI, Arlington, VA
lorin@isi.edu

Yang Jiao
Virginia Tech, Blacksburg, VA
jiaoyang@vt.edu

***Abstract*—All compiled software systems require a build system: a set of scripts to invoke compilers and linkers to generate the final executable binaries. For scientific software, these build scripts can become extremely complex. Anecdotes suggest that scientific programmers have long been dissatisfied with the current software build toolchains. In this paper, we describe preliminary results from a case study of two projects to estimate the fraction of effort devoted to maintaining these scripts, which we refer to as the 'build tax'. While estimates based on line counts are on the order of only 5%, estimates based on activity-related metrics suggest much higher values.**

***Keywords*-scientific computing; makefiles; case studies; software repositories**

I. Introduction

When a developer sets out to write a program in a compiled language, she will invariably write two: the main program itself, and a secondary one that invokes compilers and linkers to transform the main program from source code to executable binary. These secondary programs are commonly implemented as *makefiles*, a script-based technology which dates back to the mid-seventies [3]. As a community, we software engineering researchers have not paid much attention to the development of build scripts. However, for scientific software, writing and maintaining these build scripts can be a substantial headache. In this paper, we refer to this additional effort overhead to maintain build scripts as the *build tax*. In order to estimate the magnitude of the build tax, we performed a case study of two computational science projects. By focusing on two projects, we hope to gain insights into the software development process of computational scientists and provide initial estimates on the impact of build effort that will serve as a starting-off point for future studies, as well as to motivate the development of better build tools.

The projects we selected for our case studies have much in common: both incorporate simulations of thermonuclear reactions, are written mostly in Fortran, and have access to unclassified supercomputers located at U.S. Department of Energy (DOE) facilities. The first is the FACETS project[1], a distributed computational science software project led by Tech-X Corporation. The second is the Flash Center[2], a collocated computational science software project based at the University of Chicago.

A. Background: FACETS and FLASH projects

The FACETS (Framework Application for Core-Edge Transport Simulations) project was started in early 2007 with the goal of providing a framework for simulation plasma confinement for the fusion energy platform. The project is supported by funding from the U.S. Department of Energy (DOE). The project team is distributed across eleven organizations, including commercial companies, U.S. government laboratories, and university labs. Because the project is spread across multiple organizations, the physics components are developed in different institutional cultures, where each culture represents unique challenges and code development practices that impact software engineering issues.

The Flash Center was started at the University of Chicago around 1997 with the goal of studying thermonuclear flashes, events of rapid or explosive thermonuclear burning that occur on the surfaces and in the interiors of compact stars. To conduct this research, the Flash Center developed a simulation code called *FLASH* [1]. The FLASH code simulates the thermonuclear explosions of stars, and can be used for various other astrophysical, cosmological and computational fluid dynamics simulations. The FLASH code is used both by scientists affiliated with the Flash Center as well as external users, who can obtain access to the source code at no cost if certain conditions are met.

B. Prior beliefs

Our prior beliefs about build effort going into this project were based on the prior work of Kumfert and Epperly [5]. They ran a survey of computational scientists at DOE labs and universities and found reported overheads of about 12%, with individual cases into the 20–30% range. We believed that the FACETS project would lie at the higher end of this scale because it has a large number of external libraries as dependencies and

[1] http://www.facetsproject.org

[2] http://flash.uchicago.edu

has a wider variety of legacy practices, and that the Flash project would be in the middle end of the scale.

C. How analysis was done

In the literature, previous research on development effort in commercial software project has leveraged change request data to estimate effort by using the length of time a change request is open. For example, Eick et al. used this method to analyze code decay[2], and Herblseb and Mockus used this to analyze the impact of distributed software development [4]. For our study, while both projects under investigation had local installations of issue tracking tools, neither project consistently uses those tools for issue tracking. Therefore, so we could not rely on issue open/close dates to estimate effort.

Instead, we analyzed other data from the software repositories that were generated naturally be the scientists as they developed the software. The primary sources of data we used were source code, version control repositories and regression testing results.

Both FACETS and the Flash Center use subversion as their version control system. Both projects have developed their own custom regression testing solution. Each night, the test system executes numerous tests of code on multiple machines and displays the results of the tests in a web interface. The FACETS regression test system also sends out emails to the mailing list with the result of the tests.

II. Estimating build overhead

A. Build systems

1) FACETS build system: FACETS uses the GNU autotools toolchain[3] to generate executables from source code. This toolchain will be familiar to most users of UNIX-based systems that have had to build a C-based open source software package from source code.

The FACETS project has implemented their own package management system on top of autotools, called *Bilder*[4]. The system is implemented entirely in bash shell scripts. FACETS compiles a single executable and instantiates components at runtime based on an input file.

2) FLASH build system: The FLASH build system is based on a combination of Makefiles, Python scripts, and FLASH-specific configuration files. The FLASH code is designed as a collection of components. For any particular execution run, a subset of the components are compiled and linked together to form an executable. The user specifies these components at compile-time by editing a FLASH configuration file. The Python scripts use these configuration files to assemble together the relevant FLASH components and generate the appropriate makefile.

B. Code volume

1) FACETS code volume: FACETS comprises about 1.5 MSLOC (millions of source lines of code). To count lines of source code and classify by language, we used the University of Hawaii edition of *SCLC*[5], using the trunk of the FACETS code repository as of January 1, 2011. We considered any hand-generated files used by the autotools toolchain to be makefile-related. We did not count non-source files (e.g., data files, images, documentation), nor did we count auto-generated makefiles or intermediate products, which are not maintained in the repository.

We did an initial classification of source files by programming language. In most cases, the filename extension clearly indicates the type of file. For example, *.cpp* extensions were classified as C++, *.f* extensions were classified as Fortran, *.sh* extensions were classified as shell scripts. In some casese, the extension did not provide enough information to classify file type. For example, many files had *.in* extensions, because they were template files in various languages. In these cases, we manually defined rules that were used to properly classify these files, and we augmented SCLC to support language classification by rules.

The code is primarily Fortran and C/C++. However, there are over a hundred thousand lines of code that are not primarily source code, but are makefile scripts, as well as scripts in other languages, such as Unix shell (bash), Python, IDL, Perl, Matlab, (Emacs) Lisp, and XML.

2) FLASH code volume: FLASH comprises about 420 KSLOC. FLASH is about 420 KSLOC, most of which is Fortran code, primarily Fortran 90 with some Fortran 77. In addition to some C, we see many of the same scripting languages as in FACETS (makefile scripts, Python, shell scripts, IDL, Perl, Matlab).

3) Estimating build effort by size of code: If the amount of time spent on build-related development issues is proportional to the time spent editing build-related code, then we can estimate the percentage of time spent on build-related issues by examining the source code alone.

We divided up the code into four categories: 1. *Source*: source files that are part of the simulation software itself, typically C/C++ and Fortran. 2. *Build*: source files associated with the build. Mostly make-related files, but may also include scripts. 3. *Other*: any source files that are not part of *source* or *build*. Typically associated with

[3] http://www.gnu.org/software/autoconf

[4] https://ice.txcorp.com/trac/bilder

[5] http://code.google.com/p/sclc

pre-processing input files, post-processing data, or other utilities not related to building. 4. *Unknown*: any files where it was not obvious to the authors how to classify the file.

If a file is used for the process of transforming the source code into an executable, and the file is manually generated, then we considered it build-related. For both projects, we consider any hand-generated file used by autotools to be a build file

For the FACETS projects, we include all shell script files that the comprise the *Bilder* system as build-related. For the FLASH project, we include the Python build scripts that make up the FLASH build system, as well as the FLASH configuration files that serve as inputs to the build system.

Build-related code represents about 6% of the entire FACETS repository and 5% of the FLASH repository.

C. Version control activity

The data from the previous section provides a sense of how much build-related code is in the project, but that analysis assumes that time spent on build-related issues is directly proportional to the amount of build-related code. However, where some files will be touched infrequently, others will be modified many times over the lifetime of the project. For example, it seems reasonable that build-related code does not require as much modification over the lifetime of the project as source code, because the build configuration does not change as often as bugfixes and new code features.

An alternative way to measure the build overhead is to measure the amount of time that the developers spend editing build-related code. While this cannot be directly measured in retrospect, we can estimate this value using the version control history of the software. As mentioned in Section I, both FACETS and FLASH use Subversion as their version control system. To estimate the build overhead, we calculate the percentage of build-related commits.

The FLASH repository contained 11292 revisions in the range February 22, 2005 – June 14, 2011. However, we only considered revisions that affected at least one file that fell into any of the four categories mentioned in Section II-B. We neglect commits to files unrelated to development, such as documentation files or data files. This brings the number down to 5184 commits.

If we consider only commits where every file involved was build-related, **we get an estimate of 19%**. If we consider a more liberal definition, where any commit that touches at least one build file is build-related, this yields **an estimate of 37%.**

The FACETS project has a more complex version control repository. The codebase is distributed across 28 separate subversion repositories, and extensive use of subversion externals enables developers to check out the entire source tree with a single checkout command.

In the range January 16, 2000 – June 15, 2011, there were 12355 commits of interest. If we consider only commits where every file involved was build-related, **we get an estimate of** 58%. If we consider a more liberal definition, where any commit that touch at least one build file is build-related, this yields **an estimate of 65%.**

D. Automated regression tests

Both projects run daily automated tests using custom regression test systems. Regression tests runs daily, and a web-based dashboard shows the test results.

For FACETS, we examined test data in the range January 01, 2010 – December 15, 2010. There were 2812 regression tests run, including builds, across 3 different machines. **Build failures were 11% of all failures.**

We can use these results to estimate the percentage of effort spent fixing regression errors that are build-related. We assume a failure was build-related if either the build failed, or tests passed when run on one machine but failed when run on a different machine.

Out of the 1120 test failures, there were 243 that passed on one machine but failed on another. This fraction represents 22% of all failed tests. If we count build-related failures as a fraction of total, we get 30%

For Flash, we examined test data over a period from June 30, 2009 to June 30, 2010. There were 156778 tests executed by FlashTest across 8 platforms in this time period.

Compilation-related failures were 13% of all failures. As with the previous analysis, we also considered more general build-related failures as either failure in compilation, or tests passing on one machine but failing in text or execution on another machine.

There were 8731 tests that passed on one machine but failed in test or execution on another machine. If we combine that with the number of failed compilations, 47% **test failures appear build-related**.

III. Discussion

Table I summarizes the estimates of build overhead using the three different metrics we examined, ordered from smallest to largest estimates.

Table I
Estimated build overhead by metric

	FACETS	FLASH
Lines of code	6%	5%
Regression tests	11–30%	13–47%
Version control repository	58–65%	19–37%

Note the large variation in estimates across the different metrics, with lines of code being both the smallest and the simplest to measure. Only a small fraction of the codebase is made up of build-related scripts, which was consistent with estimates given to us by developers on each of the projects before we began our analysis. However, the estimates are significantly larger when examining regression tests or commits to the version control repository.

We were surprised by the discrepancy between lines of code and version control repository: we expected results to be similar here. However, these results suggest that build scripts are modified much more often than one would expect given the small fraction of overall code that they represent. Even though the FLASH project has much fewer external dependencies than FACETS, about a fifth to a third of the code files committed were build-related.

Feedback from the developers suggests that the single biggest driver for build issues is the need to support the so-called Leadership Class Facilities, the large, one-of-a-kind supercomputers. Such machines offer the highest level of performance, but at a cost. Often, such machines require cross-compilation, which adds an additional level of complexity. The associated non-standard installations, use of compiler scripts, and frequent software upgrades makes maintaining stable builds on leadership machines a challenge for code teams.

IV. Threats to validity

When examining any case study, interpreting and generalizing the results is challenging because of the context-dependent nature of the results. Here, we briefly touch on some particular threats to validity when interpreting the results of this study.

The major threat to validity in this analysis is that of *construct validity*. In this case, the construct we are trying to measure is development effort: time spent on writing and maintaining build scripts. Because we can't measure this directly, we are forced to rely on indirect measures. We attempted to mitigate this by employing triangulation, using multiple metrics.

There is an implicit assumption underlying this work that the amount of build-related overhead is constant for a mature software project such as FACETS or FLASH. In fact, we have reason to believe that this is not the case, since we expect that the build-related development will increase when porting to new platforms or when incorporating new libraries. In this study, we did not have the opportunity to analyze the temporal nature of the build effort in more detail.

V. Conclusions and future work

The results from these case studies suggest that the build tax represents a larger fraction of the total development effort than simple line counts would indicate. While it is always risky to generalize from two cases, the implications are important for the HPC community, as these results give a sense of the real costs associated with maintaining complex build systems and motivate the need for better build tools.

There are several avenues we intend to pursue. We plan to validate this initial analysis through a more detailed qualitative examination of the data. We also plan to look in more detail at the temporal nature of the build-related development effort. We hypothesize that build-related activity is episodic, occurring mainly when porting to new machines or incorporating new libraries, but this hypothesis has not been tested. We also would like to qualitatively assess build-related development activity, by addressing questions such as: What are the types of build-related problems that developers run into? Which ones are the hardest to resolve? How do developers typically fix them?

VI. Acknowledgments

This work has been supported under DOE award DE-SC0002347. The authors gratefully acknowledge the help of the FACETS team and the Flash Center Code Group, without whom this work would not have been possible. The FLASH code is supported by the U.S. Department of Energy under Grant No. B523820 to the Center for Astrophysical Thermonuclear Flashes at the University of Chicago. The FACETS code is supported by the DOE SciDAC program. We would also like to thank John Cary, Scott Kruger, Anshu Dubey, Tom Epperly, and Sveta Shasharina for their feedback on an earlier version of this paper.

References

[1] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35:512–522, October 2009.

[2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27:1–12, 1998.

[3] S. I. Feldman. Make – a program for maintaining computer programs. *Softawre: Practice and Experience*, 9(4):255–265, 1979.

[4] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally-distributed software development. *IEEEE Transactions on Software Engineering*, 29(6), June 2003.

[5] G. Kumfert and T. Epperly. Software in the DOE: The hidden overhead of "the build". Technical Report UCRL-ID-147343, Lawrence Livermore National Lab., Feb. 28 2002.