LLNL-TR-499074

LAWRENCE LIVERMORE NATIONAL LABORATORY

Software Construction and Composition Tools for Petascale Computing SCW0837 Progress Report

T. G. W. Epperly, L. Hochstein

September 13, 2011

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Program Announcement to DOE National Laboratories LAB 08-19
Software Development Tools for Improved Ease-of-Use of Petascale Systems
Software Construction and Composition for Petascale Computing
SCW0837

Proposal Submitted By:
Lawrence Livermore National Laboratory

Principal Investigator:
Thomas G. W. Epperly
Center for Applied Scientific Computing Division
Computation Directorate
Lawrence Livermore National Laboratory
7000 East Avenue, L-561
Livermore, CA 94551
(925) 424-3159 (925) 423-2993 (FAX)
epperly2@llnl.gov

Co-Investigator
Lorin Hochstein
Information Sciences Institute
University of Southern California
3811 N. Fairfax Dr., Suite 200
Arlington, VA 22203
(703) 812-3710 (703) 812-3712 (FAX)
lorin@isi.edu

Certifying Official:
John Grosh
Department Head, Computing Applications & Research Department
(925) 424-6520, (925) 423-4820 (FAX)
Grosh1@llnl.gov

Awarded Funding by Year:

FY 2009: $ 260K (LLNL)/$95K (USC-ISI)
FY 2010: $ 260K (LLNL) /$95K (USC-ISI)
FY 2011: $ 260K (LLNL) /$95K (USC-ISI)

Total funding: $ 780K (LLNL)/$285K (USC-ISI)

Use of Human Subjects in Proposed Project: No
Use of Vertebrate Animals in Proposed Project: No

Signatures:

[illegible] 9/13/2011 [illegible] 9/13/11

Thomas Epperly Date John Grosh Date

Software Construction and Composition Tools for Petascale Computing SCW0837 Progress Report

A Progress Report Submitted to the DOE Office of Science

Program Announcements:	LAB 08-19 and DE-PS02-08ER08-19
Program Area:	Software Development Tools for Improved Ease-of-Use of Petascale Systems
Program Office:	Office of Advanced Scientific Computing Research
Technical Contact:	Sonia Sachs

Investigators

Thomas Epperly, PI	Lawrence Livermore National Laboratory
Lorin Hochstein	USC Information Sciences Institute

Contents

Abstract

The majority of scientific software is distributed as source code. As the number of library dependencies and supported platforms increases, so does the complexity of describing the rules for configuring and building software. In this project, we have performed an empirical study of the magnitude of the build problem by examining the development history of two DOE-funded scientific software projects. We have developed MixDown, a meta-build tool, to simplify the task of building applications that depend on multiple third-party libraries. The results of this research indicate that the effort that scientific programmers spend takes a significant fraction of the total development effort and that the use of MixDown can significantly simplify the task of building software with multiple dependencies.

1. Introduction

The goal of this project has been to perform research and development to understand and address the challenges of configuring and building scientific libraries and applications for petascale computer systems and beyond. Our work has had two main thrusts. First, we performed empirical studies of existing applications that run on petascale systems in order to characterize the scope and nature of the build challenge faced by scientists. Second, we have worked on MixDown, a tool to simplify the compilation of multiple third party libraries involving several build toolsets.

The motivation for this project came out of our experience at Common Component Architecture [1] coding camps where configuration and build problems seemed to dominate the time spent getting codes to work together. These coding camps were designed to help people combine libraries and applications together to achieve new scientific capabilities using the CCA. Typically, these camps were scheduled for 5 days, and we would spend 3 days getting all required software to build on a shared machine. The time required to edit source code to add calls between libraries and applications was relatively short compared to the configuration and build challenge. This experience demonstrated that building all the required third party libraries required for an application is a major challenge even when representatives of each library are present. Our experience across multiple projects suggests that configuration and build is a sigificant challenge associated with composing applications from multiple software libraries.

This project is a collaboration between the University of Southern California's Information Sciences Institute (ISI) and the Lawrence Livermore National Laboratory (LLNL). ISI's focus has been on an empirical software engineering study of the configuration and build problem, and LLNL's focus has been on developing MixDown.

Our original proposal was for \$457K/year with \$307K/year allocated to LLNL and \$150K/year allocated to ISI. Our actually funding levels were \$260K/year for LLNL and \$95K/year for ISI. Due to the reduction in funding and changes in staffing, we had to reduce the scope of work performed. In particular at LLNL, we were unable to staff this project with a postdoc, so we had to staff with a flexterm employee which results in a higher cost to the project.

Due to our original funding arriving late in FY2009, we are roughly two-thirds of the way through our project. Hence, this progress report covers our progress to date, and we anticipate continuing work through FY2012.

The remainder of this report will focus on a summary of our research, results, future research objectives, findings, products, and final conclusions.

2. Research Objectives

The complete configuration and build problem is too large for us to tackle in a project of this size, so we focused on two critical parts of the overall problem: empirical software engineering measurements of the problem and the third-party library build problem. The original proposal included work on a graphical `make` debugger, but due to reduced funding and Gary Kumfert's departure from LLNL, we do not expect to complete that tool by the end of this project.

2.1 Empirical Software Engineering Evaluation

2.1.1 Goals

The goal of the empirical software engineering study of the build problem is to formalize the investigators' sense that the configuration and build problem is a major drain of software development productivity and an ignored source of errors, a phenomenon we refer to as the *build tax*. In our previous work developing scientific applications and tools, the investigators had the sense that configuration and build were taking a unexpectedly large part of the development effort. At one point in the development of Babel, we esimated that the build and configuration work was taking 25% of our development effort. In some senses Babel represents a worse-case situation because it involves so many computer languages and was expected to produce static and shared libraries. Our experience with Babel led us to perform an informal survey of developers in the DOE [2] and publish our experiences [3]. These experiences led us to pursue an empirical study of the configuration and build problem.

In order to generate a more rigorous estimate of the effort associated with maintaining the build, we performed a case study [4] of two computational science projects. While a study of two computational scientific projects is not sufficient empirical data to generalize to the wider population of computational science projects, our goal in focusing in detail on two projects was to gain insights into the role that maintaining the build plays in the software development process of computational scientists and to provide quantitative estimates on the impact of build effort that will serve as a reference point that can be used as a benchmark in future research. In addition, we wanted to provide evidence of build-related challenges in order to motivate the community to develop better build tools.

The first project is the FACETS project [5], a distributed computational science software project led by Tech-X Corporation. The second is the Flash Center [6], a collocated computational science software project based at the University of Chicago. The two codes have much in common: both incorporate simulations thermonuclear reactions, both are written mostly in Fortran, both use the MPI message-passing library [7] to exploit parallelism on HPC systems, and both project teams have access to unclassified HPC machines located at U.S. Department of Energy (DOE) facilities. Both projects also had several software development processes in common. Both projects use subversion as their version control system and 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.

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 [8], and Herblseb and Mockus used this to analyze the impact of distributed software development [9]. 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,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 by the scientists as they developed the software. The primary sources of data we used were source code, version control repositories, results of regression testing systems, and mailing list acitvity.

2.1.2 Estimating build overhead

We estimated the overhead associated with maintaining a build script by looking at four metrics:

Code volume What fraction of the total lines of code were devoted to builds scripts.

Version control activity What fraction of the total amount of version control activity was associated with modifying build scripts.

Automated regression tests How many tests failures in the automated regression tests are associated with build-related issues.

Mailing list activity How much of the email traffic on mailing lists was related to build-related issues.

2.1.2.1 Code volume

To count lines of source code and classify by language, we used a customized version of the University of Hawaii edition of *SCLC* [10].

We divided up the code into four categories:

Source Source files that are part of the simulation software itself, typically C/C++ and Fortran.

Build Source files associated with the build. Mostly make-related files, but may also include scripts.

Other Any source files that are not part of *source* or *build*. Typically associated with pre-processing input files, post-processing data, or other utilities not related to building.

Unknown Any files where it was not obvious to the authors how to classify the file.

Figure 2.1 shows how the different categories of code are distributed throughout the FLASH directory structure. Several of the build files are scattered about the repository (typically FlashConfig) files. There are also two directories with a large number of build-related files: one that contains the custom Python-based build system, and another that maintains a set of site-specific makefile parameters for different HPC systems that FLASH has previously run on.

For our estimations, we need an operational definition of "build-related code". Here, we adopt the convention that if a file is used for the process of transforming the source code into an executable, and the file is manually generated, then it is build-related. For both projects, we consider any hand-generated file used by autotools to be a build file, such as:

- configure.ac
- aclocal.m4
- Makefile.h.in
- hand-crafted makefiles

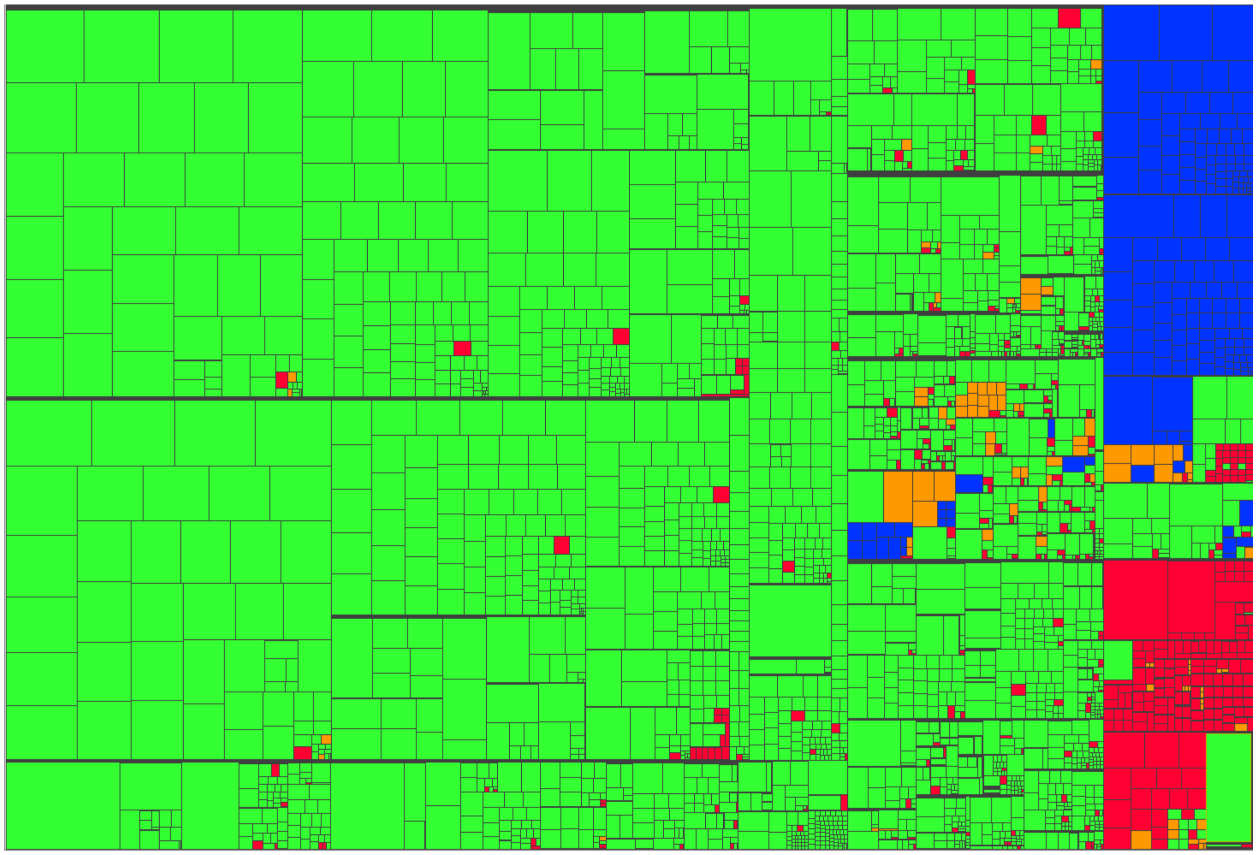
Figure 2.1: Treemap visualization of FLASH code. Green is *source*, red is *build*, blue is *other*, and orange is *uncertain*

The data from source line counts 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.

2.1.2.2 Version control activity

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. Both FACETS and FLASH use Subversion as their version control system. To estimate the build overhead, we calculate the percentage of build-related commits.

Because individual commits can involve both build-related and non-build related source files, we use both a conservative metric (consider only commits where every file involved was build-related), and a more liberal metric (any commit that touch at least one build file is build-related).

2.1.2.3 Automated regression tests

Both projects run daily automated tests using custom regression test systems. In the FLASH project, the custom regression testing system is called *FlashTest* and is available for public download[1]. FlashTest runs a suite of regression tests on different machine architectures and compilers each day and provides a publicly accessible web interface which displays the results[2]. In the FACETS project, the regression testing system is part of the *Bilder* system. Regression tests runs daily, and the team members are informed via email about the results of the test. A web-based dashboard shows the results of builds on selected machines[3].

2.1.2.4 Mailing list activity

Both the FACETS project and the Flash Center have mailing lists, although they use them differently. Because the FACETS team is a distributed project, email is an important communication tool for coordination. The FACETS project uses a private developer mailing list, which serves two purposes:

- General communication across members
- Reporting the results of daily automated tests.

We leveraged the mailing list as another source of data for estimating the amount of developer effort spent on build issues. If we assume that the number of emails on build-related issues is proportional to the amount of effort spent on build-related issues, then we can use the emails as another metric to estimate build overhead.

The challenge in using the emails to estimate build effort is to identify which ones are related to build-related issues. While we considered simple approaches such as looking for related keywords (e.g., compile, makefile), we decided to use manual inspections of a sample of the data.

Two researchers independently examined and classified all non-automated emails in the two mailing lists from May 1, 2010 to May 31, 2010 and identified which ones were build related, and then measured the inter-rater agreement using the Kappa statistic [11]. It was relatively high for FLASH (0.86) but low for FACETS (0.39), which indicates that it was more difficult to make a subjective judgment about whether a particular discussion on the mailing list was build related in the case of FACETS.

2.1.2.5 Summary of results

Table 2.1 summarizes the estimates of build overhead using the four different metrics we examined, ordered from smallest to largest estimates.

Table 2.1: Estimated build overhead by metric

	FACETS	FLASH
Lines of code	6%	5%
Mailing list	13–20%	8–10%
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,

[1]`http://flash.uchicago.edu/web/index.php/testing.html`
[2]`http://flash.uchicago.edu/website/testsuite/`
[3]`https://orbiter.txcorp.com/BilderDashboard/facets/`

which was consistent with estimates given to us by developers on each of the projects before we began our analysis. Based on mailing list traffic, the estimate roughly doubles or triples. However, the estimates are significantly larger when examining regression tests or commits to the version control repository.

What was particularly surprising was 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.

Each metric gives an incomplete view of the total development effort. In addition, because of the large spread in these metrics, it is difficult to identify a point estimate, or even an interval. However, given that the source line of code estimates are consistent with developer estimates, and are at the low range of the scale, these results suggest that the project developers underestimate the overall project time spent dealing with build-related issues.

Feedback from the developers suggests that the single biggest driver for build issues is the need to support the so-called Leadership Class Facilities. These are the large, one-of-a-kind supercomputers that occupy the upper echelons of the Top 500 list of supercomputers[4]. 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.

2.2 MixDown for Building Third-party Software Libraries

The goal of this research thrust is to design, implement, and evaluate a new tool to orchestrate the configuration and building of third-party libraries for a software application. This class of tools has come to be known as a meta-build system. Other examples of meta-build systems includes bilder [12], Contractor [13], and waf [14]. The fact that several similar tools have been created in the recent past shows that this is a problem that requires a solution.

In home and office computing, software is exchanged primarily in binary format — as compiled machine code or collections of bytecode that run in a virtual machine (e.g., Java or .Net). Windows, Java, and .Net all provide component frameworks that allow binary components to be combined into arbitrarily into hybrid applications. For example, putting an Excel graph into a Word document is creating a hybrid application with components from Excel and Word working together.

For several reasons, software is exchanged in source code form in the scientific and high performance computing community. Because of the importance of high performance, it is important to compile the software to make the best use of the available hardware. Often compilers can improve performance when compiling for a particular machine rather than compiling for a general class of machines. Some parts of the software may depend on the type of processing cores, the type of accelerators available, and the particular networking hardware. Leadership computing facilities are often cross-compile environments where particular versions of the compilers and system libraries must be used. It is not reasonable to expect a library provider to have access to all the different systems for which binary packages would be required. Even though much of the community has standardized on Linux, there are still multiple varieties of Linux, and a developer cannot easily make a binary package that will install everywhere.

Because software is exchanged in source form, a computational scientist who wishes to synthesize applications from a collection of community written tools must configure and install the third-party libraries in addition to his own software. Sometimes these libraries are already installed on the system. For example, most supercomputers have a vendor-optimized MPI preinstalled. In many cases, the computational scientist

[4]`http://www.top500.org`

may choose to ignore the preinstalled software and choose to install all their dependencies from scatch to avoid the possiblity of incorrect software versions or inconsistent build parameters.

The role of a meta-build system is to take a collection of software libraries and applications from source code form to their compiled and runnable format with a consistent set of compilers and flags to produce a correct, fast build. The meta-build system should manage dependencies between software packages to ensure that dependencies are installed before packages that require them. The key distinction between a meta-build system and a build system is that the meta-build system manages things at a package level. It assumes that each package has its own configuration and build system.

Our system, MixDown, will default to an eight step process for each library or application. The default list of steps is fetch, unpack, patch, preconfig, config, build, install, and clean. The list of steps can be customized and some steps can be empty for some packages. For example, the patch step is only used when the scientific application requires a modified verison of the distributed library. Each of these steps is explained below:

fetch download the third party package from a web server, Subversion repository, GIT repository, Mercurial repository, or filesystem.

unpack if the software package is in an archive such as `.zip`, `.tar`, `.tar.bz2`, etc., this step will extract the software package from its archive in a build area

patch apply a small change to the unpacked source usually using the Unix utility `patch` or a simple shell script

preconfig some packages require a tool to generate the configuration script

config execute the configuration script which automatically determines machine specific information needed to compile the software package

build invoke the packages build tool which applies compilers to source code to create the final, usable form of the library or application

install this directs the package to install its end-use files into a directory where other packages can reference/use them.

clean this removes temporary files created during the other steps of the process

In addition to managing the stages of each individual packages transformation to end-use form, MixDown manages the dependencies between packages. By analyzing the dependency graph, MixDown can determine the proper ordering of packages and identify which packages can be configured and built concurrently. Figure 2.2 shows the dependencies between packages required to build the Subversion source code repository system. This shows that `apr-util` must be built first. Then `apr`, `neon`, and `sqlite` can be built in parallel. Followed last by Subversion itself.

To improve the ease of adoption for MixDown, we developed a system to analyze a collection of archives (`.zip` or `.tar` files) and/or URIs to automatically generate an initial MixDown project file. Our aim was to develop heuristics that would get most of the MixDown project file correct for projects that follow common practices; it is impossible to develop something that could correctly handle packages that are designed contrary to common practice. The MixDown file may require some hand editing to correct rules or dependencies.

The MixDown project file is one of the key user facing aspects of the tool, and we have tried to make the project file simple and clean for the average case. Figure 2.3 shows a MixDown project file for the

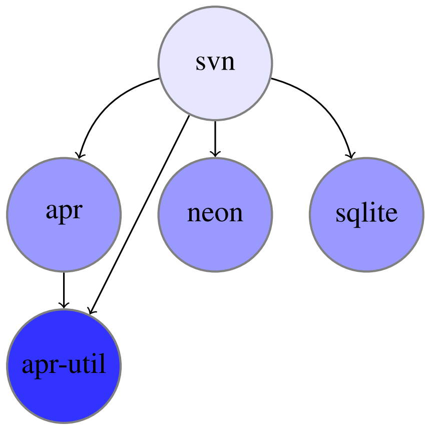

Figure 2.2: Graph showing the dependencies for the packages required for the Subversion source code repository system

Subversion source code repository. For complex situations, the software developer has the option to override MixDown's default stages and behavior. In the MixDown project file, the eight step process can be reordered, augmented, or reduced. In our experience, there are always situations where a simple approach cannot handle the reality of what a build system must do, so developers can direct MixDown to use a custom Python script to implement one or more of the steps for a given software package. This gives the developer the flexibility to do almost anything as part of the build/install when necessary. In the Subversion input file, the `Preconfig` rule for `apr` shows an example of a call to native Python code which is show separately in Figure 2.4.

We have tested MixDown with a large list of open source programs including several DOE packages. MixDown's automatically generated input file successfully built NWChem [15], PETsc [16], Sundial [17], Virtual Box [18], OpenFOAM [19], Gimp [20], Thunderbird [21], Firefox [22], MySQL [23], Deal II [24], NTL [25], CLHEP [26], and Hmmmer [27]. We have also tested it with GCC, Quantum ESPRESSO [28], GAP [29], Madagascar [30], Randomlib [31], CMU Sphinx [32], SUNDIALS [17], VLC [33], AZTEC [34], Freemat [35], Hypre [36], PyMOL [37], ScaLAPACK [38], TAO [39] and OpenFlower [40].

The next key part of MixDown is a database of settings for different machines. When compiling on DOE leadership class machines, the build system must usually choose from a limited set of compiler and use very particular build options. For example on NERSC's franklin compiler, you have a choice between PGI, Pathscale, Cray, and GNU compilers. All these compilers are invoked using Cray wrappers named `ftn`, `cc`, and `CC` for the Fortran, C, and C++ compilers, respectively [41]. Each compiler choice has recommended compiler settings. The MixDown database would have all these settings builtin.

At this point, we are designing the MixDown database. This design incorporates the file format for recording information, the way information is organized, and how MixDown determines which combinations of settings to use. After the design work is finished, we will populate the database with settings from the DOE LCFs and other common machines.

```
Name: subversion
Path: subversion-1.6.12.tar.bz2
DependsOn: apr,apr-util,neon,sqlite
Fetch: steps.fetch(pythonCallInfo)
Unpack: steps.unpack(pythonCallInfo)
Preconfig: ./autogen.sh
Config: ./configure $(_AutoToolsPrefix) $(_AutoToolsCompilers) --with-apr=$(_Prefix) \
        --with-apr-util=$(_Prefix) --with-neon=$(_Prefix) --with-sqlite=$(_Prefix)
Build: make $(_MakeJobSlots)
Install: make $(_MakeJobSlots) install
Clean: make $(_MakeJobSlots) clean

Name: apr-util
Path: apr-util-1.3.10.tar.bz2
DependsOn: apr
Fetch: steps.fetch(pythonCallInfo)
Unpack: steps.unpack(pythonCallInfo)
Preconfig: ./buildconf
Config: ./configure $(_AutoToolsPrefix) $(_AutoToolsCompilers) --with-apr=$(_Prefix)
Build: make $(_MakeJobSlots)
Install: make $(_MakeJobSlots) install
Clean: make $(_MakeJobSlots) clean

Name: sqlite-autoconf
Path: sqlite-autoconf-3070500.tar.gz
Aliases: sqlite
Fetch: steps.fetch(pythonCallInfo)
Unpack: steps.unpack(pythonCallInfo)
Preconfig: autoreconf -i
Config: ./configure $(_AutoToolsPrefix) $(_AutoToolsCompilers)
Build: make $(_MakeJobSlots)
Install: make $(_MakeJobSlots) install
Clean: make $(_MakeJobSlots) clean

Name: neon
Path: neon-0.29.5.tar.gz
Fetch: steps.fetch(pythonCallInfo)
Unpack: steps.unpack(pythonCallInfo)
Preconfig: ./autogen.sh
Config: ./configure $(_AutoToolsPrefix) $(_AutoToolsCompilers)
Build: make $(_MakeJobSlots)
Install: make $(_MakeJobSlots) install
Clean: make $(_MakeJobSlots) clean

Name: apr
Path: apr-1.3.12.tar.bz2
Fetch: steps.fetch(pythonCallInfo)
Unpack: steps.unpack(pythonCallInfo)
Preconfig: svnSteps.aprPreconfig(pythonCallInfo)
Config: ./configure $(_AutoToolsPrefix) $(_AutoToolsCompilers)
Build: make $(_MakeJobSlots)
Install: make $(_MakeJobSlots) install
Clean: make $(_MakeJobSlots) clean
```

Figure 2.3: MixDown input file for the Subversion source code repository

```python
import socket, sys
from md import commands, utilityFunctions

def aprPreconfig(pythonCallInfo):
    if socket.gethostname() == "myComputerHostName":
        pythonCallInfo.logger.writeMessage("Skipping␣APR␣Preconfig␣step␣due␣to␣correct␣hostname␣found")
        pythonCallInfo.success = True
    else:
        pythonCallInfo.logger.writeMessage("Running␣APR␣Preconfig␣step")
        returnCode = utilityFunctions.executeSubProcess("autoreconf␣-i", pythonCallInfo.currentPath, pythonCallIn
        if returnCode != 0:
            pythonCallInfo.success = False
        else:
            pythonCallInfo.success = True

    return pythonCallInfo
```

Figure 2.4: Custom Python code used for the Preconfig rule for the apr package

3. Ongoing Research Objectives

3.1 Details of build-related changes

One of the original goals of this work is to understand the nature, frequency, and severity of build problems. We are currently working on developing and applying a categorization scheme to identify the key areas of work in configuration and build. Based on analysis of the FLASH and FACETS code, we have developed the following provisional categorization scheme:

1. Adding/Modifying flags
2. Adding/Modifying path or version of a library
3. Cosmetic changes to the build code
4. Adding a new build related file
5. Modifying the program flow
6. Adding/Modifying dependencies in the rules

We are in the process of analyzing the FLASH and FACETS code using this characterization scheme to identify the types of changes.

3.2 MixDown evaluation

In the upcoming year, we plan to do usability studies of MixDown: collecting data on users interacting with the tool in order to identify any problems with the existing interface.

We will also test MixDown on a variety of DOE LCFs to determine machine appropriate settings and to verify its correctness. Our goal is to have settings available for the common DOE platforms at NERSC, ANL, and ORNL. These evaluations may also identify required modifications.

We will also develop a means to store information about each package built. The installation of each package should carry metadata about which compilers, settings, and software versions were incorporated into a particular build. This information is crucial to reconstruct the software provenance of a particular build. MixDown should produce information for each installed package using a standardized format.

3.3 Other areas

Since our initial proposal, there have been some shifts in build and configuration landscape. For example, CMake [42] has emerged as a growing standard for DOE projects being supported by the ASC program and Sandia National Laboratory. We should investigate whether there are particular challenges in CMake that are not being addressed.

We will evaluate the `make` debuggers that are already available and discover if they can be extended. We will investigate whether our ideas for graphical debugging can be incorporated in any of the existing tools.

4. Results

The results from the two case studies suggest that, while looking only at lines of code associated with build activity provides a modest estimate of build effort (5%), development effort metrics that are based on developer activity provided significantly higher estimates of build overhead. Build overhead estimates from the FACETS project estimates were higher than those for FLASH, most likely because the FACETS project has more software dependencies.

We have tested MixDown on 24 Open Source projects. For 16 of those projects, the automatically generated MixDown project file was sufficient to correctly build the project. These tests did not cover the third-party library requirements for these package, but these results are still significant because it indicates that the automatically generated project file is correct in two thirds of the cases.

5. Products

Here are out products and references on how to obtain them:

MixDown MixDown's source code repository is maintained at `https://github.com/tepperly/MixDown`. This provides open read-only access to everyone via the internet and read/write access to project contributors.

MixDown Wiki The MixDown wiki, `https://github.com/tepperly/MixDown/wiki` provides some information about MixDown, our use cases, and our design documents.

MixDown poster We presented a MixDown poster at a LLNL poster session for build/configure tools and techniques.

ESEM paper We will present a paper at the IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) on September 22-23, 2011 [43].

6. Conclusions

The results of this project show that the effort to maintain and build petascale computational science projects is a significant fraction of total development effort. Assuming that future computational science projects continue to be developed through composition of third party libraries, we expect this trend to continue.

The two projects that composed the current case studies both targeted homogeneous architectures exclusively. Given current trends in heterogeneous architectures, we expect that complexity in the build process to increase over the next several years. In particular, the trend toward cross-compilation for DOE Leadership Class Facilities makes the configuration and build process more complex because the operating system running on the system where the developers work is dissimilar from the computation nodes where the parallel code actually runs.

Meta-build tools solve one of the key problems facing scientific and high-performance computing by managing the configuration, build, and installation of third-party libraries. Improvements in meta-build tools and fundamental builds tools can manage aspects of the underlying complexity and improve developer productivity.

7. Bibliography

[1] The Common Component Architecture, http://www.cca-forum.org/.

[2] G.K. Kumfert and T.G.W. Epperly, Software in the DOE: The Hidden Overhead of "The Build", Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, 2002.

[3] P.F. Dubois, T. Epperly, and G. Kumfert, Why Johnny can't build [portable scientific software], *Computing in Science & Engineering* **5**, 83 (2003).

[4] Robert K. Yin, *Case Study Research: Design and Methods*, Sage Publications, third edition, 2002.

[5] FACETS project, http://www.facetsproject.org.

[6] Flash Center, http://flash.uchicago.edu.

[7] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, A message passing standard for MPP and workstations, *Communications of the ACM* **39**, 84 (1996).

[8] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus, Does Code Decay? Assessing the Evidence from Change Management Data, *IEEE Transactions on Software Engineering* **27**, 1 (1998).

[9] James D. Herbsleb and Audris Mockus, An Empirical Study of Speed and Communication in Globally-Distributed Software Development, *IEEEE Transactions on Software Engineering* **29** (2003).

[10] SCLC (Source Code Line Counter), http://code.google.com/p/sclc.

[11] Jean Carletta, Assessing Agreement on Classification Tasks: The Kappa Statistic, *Computational Linguistics* **22** (1996).

[12] James Cary, Bilder, available as part of the FACETS project.

[13] James F. Amundson, Contractor v1.0 documentation, http://home.fnal.gov/˜amundson/tmpcontractor/index.html.

[14] waf: The meta build system, http://code.google.com/p/waf/.

[15] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong, NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations, *Computer Physics Communications* (2010).

[16] Barry Smith, Portable, Extensible Toolkit for Scientific Computation (PETSc), `www-unix.mcs.anl.gov/petsc/petsc-as/`.

[17] SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers), http://acts.nersc.gov/sundials.

[18] VirtualBox, http://www.virtualbox.org.

[19] OpenFOAM, http://www.openfoam.com.

[20] The GNU Image Manipulation Program, http://www.gimp.org.

[21] Mozilla Thunderbird, http://www.mozilla.org/thunderbird/.

[22] Mozilla Firefox, http://www.mozilla.org/firefox/.

[23] MySQL, http://www.mysql.com/.

[24] deal.II: A Finite Element Differential Equations Analysis Library, http://www.dealii.org/.

[25] NTL: A Library for doing Number Theory, http://www.shoup.net/ntl/.

[26] CLHEP - A Class Library for High Energy Physics, http://proj-clhep.web.cern.ch/proj-clhep.

[27] HMMER3: a new generation of sequence homology search software, http://hmmer.janelia.org/.

[28] Quantum ESPRESSO, http://www.quantum-espresso.org/.

[29] GAP - Groups, Algorithms, Programming - a System for Computational Discrete Algebra, http://www.gap-system.org/.

[30] Madagascar, http://www.ahay.org.

[31] Randomlib, http://randomlib.sourceforge.net.

[32] CMU Sphinx: Open Source Toolkit for Speech Recognition, http://cmusphinx.sourceforge.net.

[33] VLC media player, http://videolan.org/vlc.

[34] Aztec: A Massively Parallel Iterative Solver Library for Solving Sparse Linear Systems, http://www.cs.sandia.gov/CRF/aztec1.html.

[35] FreeMat, http://freemat.sourceforge.net/.

[36] hypre, https://computation.llnl.gov/casc/hypre/software.html.

[37] PyMOL, http://pymol.org/.

[38] ScaLAPACK, http://www.netlib.org/scalapack/.

[39] The Toolkit for Advanced Optimization (TAO).

[40] OpenFlower, openflower.sourceforge.net.

[41] Compiling Codes on Franklin and Hopper, http://www.nersc.gov/users/computational-systems/hopper/programming/compiling-codes/.

[42] Ken Martin and Bill Hoffman, *Mastering CMake*, Kitware, Inc., 2010.

[43] Lorin Hochstein and Yang Jiao, The cost of the build tax in scientific software, in *Proceedings of the the IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*, 2011.