UCRL-ID-147343

Software in the DOE: The Hidden Overhead of “The Build”

G. K. Kumfert T. G. W. Epperly

February 28, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

Approved for public release; further dissemination unlimited

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (423) 576-8401
http://apollo.osti.gov/bridge/

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161
http://www.ntis.gov/

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
http://www.llnl.gov/tid/Library.html

Software in the DOE: The Hidden Overhead of "The Build"[1]

Gary Kumfert and Tom Epperly

28 February 2002

[1]This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Release Number: UCRL-ID-147343

Abstract

"The Build" is the infrastructure needed to convert software from source code to usable form. It is intimately tied to the software it supports, knowing about every file and automating every transformation needed to produce a working program. Every developer knows that a project spends some labor overhead on "the build." How big is this hidden overhead?

According to 34 scientific software developers we surveyed at Lawrence Livermore National Labs, among colleagues at other DOE labs, and a handful of academics the "perceived" overhead averages around 12%. Individual cases of 20% to 30% were not uncommon. In one project claiming a 20% overhead, we found supporting evidence by combing through their CVS repository.

Chapter 1

Introduction

The DOE spends significant resources on software development and use. This document tries to quantify what percentage of those resources are actually spent on build issues instead of core development. By "build issues" we mean the development, debugging, maintenance and extension of the supporting infrastructure that converts source code into its end-use form. For most projects, this means Makefiles and a few helper scripts. For more widely used projects, this can mean orchestrating several tools, files, scripts, and other paraphernalia to keep the build working.

There is no easy metric for measuring the build overhead in software. In Chapter 2 we discuss the results of an informal survey we conducted to verify that problems associated with the build are widespread. For more objective data, we examine a particular piece of software in detail in Chapter 3. Conclusions and possible directions for future work are presented in Chapter 4

Chapter 2

Survey

Between November 2001 and January 2002, we conducted a survey to better understand how much time was consumed by software build details. We knew up front that our survey could not definitively tell the percentage overhead for the build, since no respondents recorded that kind of information. However, the survey can reflect the *perceived* overhead and indicate if that overhead is suffered universally or limited to a few locations/projects. The information was generated by 19 respondents at LLNL, 12 from other DOE laboratories, and 5 from Academia. These 36 people submitted 39 responses covering at least 28 separate projects.

The structure of this questionnaire deserves some explanation. There were three divisible sections. The first (and longest) section asked 15 questions pertaining to a particular project the respondent worked on. Most respondents answered this section for the project on which they spend most of their time. A few respondents provided responses to this section for more than one project with each project on a separate survey. The second section had only four questions and was particular to the individual; irrespective of any project. The final section had the respondents name, project name, and contact information. These three sections were separated and randomized to insure the anonymity of a particular response while allowing us to keep a reasonable count of how many actual projects are represented. In some cases respondents chose not to name their project.

Our presentation of the results is divided between questions requiring a numerical response and open-ended questions. For numerical responses, average and median are presented Table 2.1. The raw data is presented in Tables A.1–A.3 in the appendix. The non-numeric responses are grouped and listed in Section 2.1. The original text of the question and the number appeared in asking sequence is preserved between Table 2.1 and Section 2.1.

2.1 Written Responses

The following questions were open-ended. In this section we attempted to itemize each of the responses. Responses that made multiple points were broken out into separate bullets. We also consolidated points and added a multiplier to the end when there was sufficient overlap.

Question# 10. Do you find the current build system limiting on this project?

- No (x10)
- No — make is fine
- No — we just keep extending it
- No — our third iteration now does everything we need.
- No, but if more automatic, it would free resources for other things
- Not Really
- Not very, but it is painful
- Yes (x12)
- Yes, configure needs serious help.
- Absolutely (x3)
- Limiting in how it interacts with other build systems, especially 3rd party libraries

per Project Questions

		LLNL		other DOE		Academia	
		Mean	Median	Mean	Median	Mean	Median
1.	How many FTE's on this project?	7.8	5	3.3	2.3	12	9.5
2.	What percentage of your time is spent on this project?	76.%	85.%	66.%	60.%	70.%	78.%
3.	How many configure/build/packaging tools do you use?	2.7	2	3.2	3	2.3	2
4.a	How many platforms currently?	4.6	4	5.6	5	4.3	4.5
4.b	How many additional platforms in the future?	1.5	2	.5	0	1	1
5.	How many programming languages in this project?	3.6	3	4.8	4	3.3	3.5
6.	Regarding 3rd party libraries…						
.a	…how many are required?	5.2	3	2.5	2	3.3	2
.b	…how many are optional?	2.2	1	5.4	2	3.7	2
7	What percentage of your time spent on this project is expended on build issues?						
.a	…development time	19.%	10.%	15.%	10.%	6.%	4.%
.b	…overall time	22.%	10.%	17.%	11.%	10.%	2.%
8.	For the project overall, what would you estimate the annual cost for maintaining this project's build (configure, release, etc)?						
.a	…in FTE's	.9	.3	.5	.3	1.3	1.5
.b	…as percentage of project	11.%	10.%	15.%	13.%	16.%	15.%
9.	Would you say that this project had a significant startup cost or one-time charge for the build? If so, what? (in person-weeks)	12	3.5	8.3	4	14	12
12.	How long does the build take? (in hours)						
.a	…min	1	.5	.2	.1	.6	.6
.b	…max	2	1	.2	.2	1.4	1
13.	For each of the build tools you use, please rate your overall satisfaction on a scale of 1–10 (1 = hate it! want to rewrite from scratch : 10 = wouldn't change it for the world)						
.a	…shell	4.7	5.	4.3	4		
.b	…make	5.5	5.5	5.3	5	5.2	6
.e	…autoconf	5.2	6.	6	6	4.5	4.5
.f	…automake	5.0	7.	3	1		
14.	Using above scale, Please rate your overall satisfaction level with your overall build system	5.5	5.5	5.8	6	5	4
per Person Questions							
16.	You are going to start a new project. How interested would you be in trying a new set of build tools? (1 = no interest : 10 = chomping at the bit	7.2	8	6.5	6	6.2	7
17.	What's the most amount of time you spent fixing one build problem? (in entire career, measured in days)	21	3	4.3	1.5	4	3

Table 2.1: Average numerical answers from survey.

- Inefficient/Inconvenient, but not limiting (x2)
- Hand change everything!
- Hard to share with other users
- Overly complex
- Tools insufficient
- Requires too much effort
- Our build is our biggest risk to our project's success
- I don't know any different
- Not once its configured
- It's finally in pretty good shape, but does have a few limitations
- Required config files supplied through static methods.
- Cannot keep makefiles and Windows project files coordinated.

Question# 11. What critical piece of functionality do you wish your current build system had?

- None (x8)
- Flexibility/More amenable to change (x3)
- C++ precompiled headers
- Build C++ shared libraries (x2)
- Integrated testing and build
- Better dependency checking and analysis/full disclosure (x3)
- Something less slow, awkward, and fragile
- Handled mixed languages (link library issue) (x2)
- Protect against headers out of sync with associated libraries
- Better handling of dependencies on external packages
- Better detect and conditionally compile code if an external package has been provided
- Integrated test environment
- Like coordinating multiple developers
- Currently builds as an application, want to also build as library
- Go to a new platform without finding where libraries are, setting compiler flags, etc. (x3)
- Automatically determine how discovered libraries were built.
- An elegant way of sucking compiler options/names for different architectures out of a file/database/somehow
- #include searches
- Handles dependencies across directories (x2)
- Like to have some automake-like capabilities
- Nice to have more standard configure options (e.g. `configure --prefix`)
- Call system admins
- Determine changes in system configuration (i.e. what tools have changed)
- Easier to understand/learn (x3)
- Better user interface for developers
- Better dependence check than timestamps
- Export build/configure information to users to USE libraries after they've been installed
- Understand that one action can simultaneously produce multiple targets
- Add functionality to the project without exhausting link-line buffer
- Hierarchical composability (using configuration of an included package)
- More advance language constructs
- Ours is a fairly simply library, the build tools are adequate
- Make syntax sucks — better syntax
- Complete Automation (in my dreams!)
- Better management of dependency files when a .h file goes away
- GUI / more interactive features for users (x2)
- Better option than extremely long command lines of options to configure

- Easily handle a single tree with sources, .o's, and binaries from multiple platforms.
- A configure script. We scare a lot of potential users away as soon as they realize we expect them to edit our makefiles.
- Identical build on NT and Unix
- Coordinate testing on all platforms at once.
- Update windows project file when files are added/removed from repository
- Warn about non-standard function calls, without wading through stuff you flag as known platform specific.

Question# 15. What would it take for this project to switch to a different build tool?

- Move mountains. So ingrained.
- See another project converted to new tool in action first. (x2)
- At least as flexible as what I'm using now.
- Cross Platform (x2)
- Easier to use (x2)
- Real transition ease/ Low adoption cost (x3)
- IDE
- Run on line-oriented Unix machines
- Clear advantage
- Stable
- Recognized piece of software
- Significant user group
- Current build would have to break first
- Someone do it for us
- Medium resistance
- Cleaner / better syntax
- Easily build shared libraries across platforms
- Funny you should ask, we're in the process of switching tools right now. (x3)
- Make my lunch
- Good marketing
- Vendor change controls
- Unlikely
- No compelling reason right now, but may revisit when in production mode
- Support all our platforms
- Better user interface
- Prove a time/effort saving
- Address the issues above
- Unpack on standard UNIX without special stuff
- My retirement
- Significantly simplified build files, package dependencies
- A lot — how much time do I have to spend learning the new thing?
- Nothing for this project, but would use something new for a new project
- Less overall effort — A complete swap out to a new system must cost less than three tweaks of my current system, or else I'd just keep tweaking
- No way in Hell
- Must handle parallel builds across directories (our current system does)
- Must satisfy all our needs and be clearly simpler/better
- Supplement make
- Depends on complexity of build tool
- Convince me it will save us time
- Replicate everything we already do
- Resources. We have limited resources considering everything we need to do. Re-engineering the build system is not "research" for us, so a compelling argument must be made that the cost will free up resources in the future.

- Quite frankly, the tool would have to be custom built for our project. It seems that any alternative that could meet our diverse needs (extremely diverse platforms and configurations, on-site and off-site, custom tool integration, etc.) would either be more clumsy than what we do now, or it would require person-years of effort to customize to our environment. I don't have much faith in all-in-one tools when you've got such a wide array of needs, and this question sounds like an advertisement for an all-in-one tool.

Question# 18. Do you have a favorite war story about building software?

- No, but building and documenting sucks big time.
- No comment (x4)
- Yes
- Many Scars
- Too many to chose from
- Wrote own version of make in 1982
- Once had three projects not talking together and had to configure/build with one system
- Many war stories — no favorites
- Magic number issues
- Working w/ libraries is hard.
- Linkers are getting "better"
- C++ templates are a great idea, but they really suck! People do workarounds, sed expansions template instantiations, and break stuff BIG time.
- Change something and the other files don't get [sic]
- Build for one platform erases all others in the installation directory.
- I once spent a week tracking down a three character typo. The compiler flag was missing for a "handful" of compilation units and induced a failure only at runtime.
- I tried to write an autoconf/configure test for Tcl/Tk installation - started from a canned test in another package (the cut & paste approach to autoconf, the only way to go!). Turns out my own home Tcl/Tk installation couldn't be recognized. I spent quite a while trying to tweak the configure script, only to discover that all along (for years!) I had been doing a "make" on Tcl/Tk, but never the subsequent "make install", so my local configuration was "off the chart" in terms of where to find certain files, etc.
  Of course, I decided to leave my Tcl/Tk setup as is, and make the configure script work for it, just to cover all the other poor bastards out there who were also omitting that final "make install" step for Tcl/Tk...! It's one gruesome configure test, but it works!

Question# 19. Comments?

- People are undereducated re: autoconf and automake. Aggravation comes from misuse.
- If there exists a better tool, we should be using it.
- I am very frustrated with the time I waste writing makefiles
- Very hard to evolve using current tools. If you order directories one way and want to change a year later, you're hosed.
- Making changes is harder than it should be downstream.
- Separate answers from BIG projects from those of small projects.
- Have to write lots of scripts — make is insufficient.
- Hate autoconf, resulting config scripts fail 50% so I have to hack it. . . or . . . I pass 160 characters of mystical options to get it to work.
- No known build tool allows specifying high-level target types. Automake predefines transformations, want user defined.
- Choose build system wisely
- Better design code around the build system, than adapt build system after the fact.
- Tend to expect what we have. Communicating benefits is important to overcome inertia of existing tools.
- Portability is a big issue.
- Whenever building 3rd party software, I expect there to be problems. I expect to tweak makefiles and even source code.

- There's no assertion capabilities... no easy way to suspend and query user for guidance.
- The problems with the UNIX environment are not the simple tools, its the non-uniformity in compilers. The problems are systemic to UNIX, not enough standardization.
- I'm skeptical of tools that aren't "built-in" on systems, like make is.
- CASE tools are pricey, but can drastically improve developer productivity... of course, if distribution of software is via source code, this is not an option.
- Research environments tend to prefer spending money on people than good software tools.
- Automatically generated makefiles are a mixed blessing. Many users are not as sophisticated as they think in "fixing" them. And of course, any correct change can't be fed back upstream to the makefile generator.
- Want a consistency checker, where you make a change in the Makefiles and can see how that change cascades through the build process.
- I really hope there's something better to be made. My fear is that it would be too complex, even worse than what we already have.
- Please blast out the findings of this survey when they're done!!!

2.2 Discussion

Several pieces of information from Table 2.1 deserve highlighting. The overall average and median build overhead (Question 8.b) are 11.91% and 10%, respectively, with a maximum 35.71% and a minimum of 0%. Broken down by groups: the average and median are 10.42% and 10%, respectively for LLNL; 13.28% and 11.25%, respectively for other DOE labs; and 15.67% and 15%, respectively among academic respondents. The variations between different survey groups are probably not significant.

Both the average and maximum overheads are significant. This is a substantial amount of effort spent on build issues. Our small sample of 22 projects at LLNL accounted for 18 full time employees spent annually on build issues. If our numbers are indeed representative of all DOE software projects, the total price tag is noteworthy.

We searched for correlations in our numeric data with plots and one-dimensional least squared regressions. We thought perhaps that build overhead might correlate with some of the other factors. In addition to looking at the plots, we also evaluated the r-squared factor for the least squares fit. Both approaches led to the same conclusion; the data does not show any simple meaningful correlations between answers.

There were also some unanticipated features of this survey.

The average number of programming languages (Question 5) is surprisingly high at 3.85. Having multiple languages in a piece of software increases the complexity of the software as well as the level of difficulty when porting to new architectures. The top three languages were easily C, C++, and Fortran77. One DOE project claimed first place in this question with 8 languages, two LLNL projects tied for second with 7.

A second noteworthy result is that the overall average number of required third party libraries (Question 6.a) is also high at 4.21. The line between standard and third party libraries is far from clear. For the purposes of this survey, we defined third party libraries as anything that is not typically found on a UNIX operating system and/or having a distinct name or project within the DOE. For instance, the standard C++ runtime library was excluded in the count, but MPI was included[1]. This number is an indicator of software reuse. Lack of build tool support for handling multiple third party libraries was a commonly cited complaint in the free responses. There was anecdotal evidence to to suggest that the work involved in coordinating a number of third party libraries in the build does not scale well, however the numerical data from this survey was inconclusive. One project insisted that they had a stunning 20 required third party libraries and only a 7.5% build overhead. Conversely third place response of 11 required libraries reported the maximum build overhead at 35.7%. The type of languages the libraries are implemented in and the software maintenance devoted to the library at its source are likely significant factors that our survey failed to capture.

To summarize whether or not respondents felt their current build system was limiting (Question 10), we found 23 indicating that it was and 21 indicating that it was not. In general the responses were neither tepid nor bi-polar, but uniformly distributed across the board. Respondents were also asked to rank their satisfaction with their current build system (Questions 13 and 14) on a scale from 1 to 10 where 1 meant "hate it! want to rewrite from scratch" and 10 meant "wouldn't change it for the world." On this scale, the average satisfaction with their current project's build is 5.46. Broken out among the top three tools, average satisfaction with make (31 of the 39 responses) was

[1] This caused some consternation, particularly among those that only worked with vendor supplied MPI libraries.

5.39. Satisfaction with autoconf (19 out of 39 responses) averaged at 5.47. Finally, average satisfaction for using shell in their build (16 responses) was 4.56. These three tools were the only ones used by a sizable percentage of the respondents.

Chapter 3

Extracting an Objective Measure from CVS

Developers do not record the time they spend on build issues vs. code development. What is recorded, however, is all the changes made to every file within their software repository. We decided to count each change to each file as one unit of work, and see if this objective measure would support or refute the kinds of build overheads reported in our survey. We could have taken into account which changes are large and which ones are small, but there were problems with this strategy. Large changes may not adequately represent the time, thought, and debugging that went into the changes. Similarly, very small changes may be the result of hours and even days of hunting down a reclusive bug.

The Components Project at LLNL [2] has already identified the build as a major impediment to widespread adoption of component technology in the DOE [5]. The multi-lab Common Component Architecture (CCA) Forum [1] has also acknowledged this problem and has asked this same group to develop a component packaging standard — clearly an issue pertaining to the build.

The flagship product of the Components Project at LLNL is Babel [3, 4], a tool that enables C, C++, FORTRAN 77, Python, and Java software to interoperate on various platforms using compilers from various vendors. All this interoperability brings about serious configuration issues in their build. This project follows current best practices from the Open Source community (using autoconf, automake, and libtool) in managing the build. Members of this project casually estimate that their build overhead is a minimum of 20%.

As a first step, we listed all the active files in their CVS repository and started categorizing their function. All files were assigned an attribute from each of the following four categories: **Stage**, **Source**, **Visibility**, and **Role**. A **Stage** could be one of *original-form*, *intermediate-form* and *final-form*, depending on whether that file served as input to a program, output from one program and input to another, or simply existed in its final form, respectively. The **Source** category denoted whether the file was handmade, the output of a program executed by the developer, the output of a program executed by the user, or simply reused directly from another source with the attributes *handmade*, *dev-generated*, *user-generated*, and external respectively. The **Visibility** of a file was either *distributed* with the software, or used *internal*-ly. The **Role** of the file was one of *core-code*, *test-code*, *example-code*, *documentation*, *build-support*, *version-support*, *test-support*, or *example-support*. This is displayed graphically in Fig 3.1.

To understand how these categories interact, we will show some files and how they fit into these categories. The

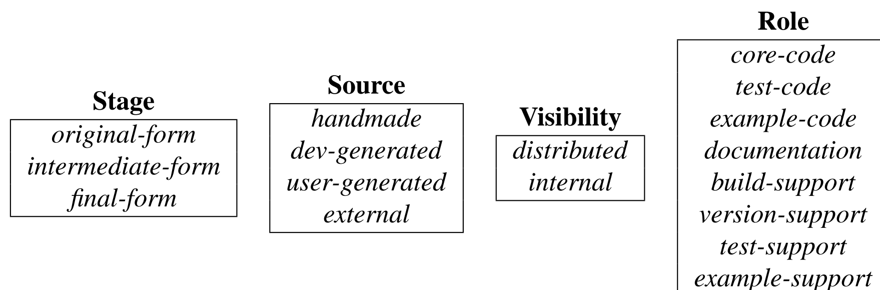

Figure 3.1: Categories assigned to all active files in CVS repository of Babel

Stage	files	lines	changes	max change
intermediate-form	403	94,123	4,243	91
original-form	610	96,820	3,141	126
final-form	174	218,915	600	152
total	1,187	409,858	7,984	

Table 3.1: File counts, line counts, number of changes and max. changes by stage

Source	files	lines	changes	max change
handmade	870	133,541	4,874	126
dev-generated	311	253,358	3,104	152
external	6	22,959	6	1
user-generated	0	0	0	0
total	1,187	409,858	7,984	

Table 3.2: File counts, line counts, number of changes and max. changes by source

`README` file would be *final-form*, *handmade*, *distributed*, *documentation*. Since Babel uses automake and autoconf, all `Makefile.am` files are *original-form*, *handmade*, *distributed*, *build-support*, automake then generates the resulting `Makefile.in` files which are *intermediate-form*, *dev-generated*, *distributed*, *build-support*. Actual `Makefile`'s that are produced from `Makefile.in`'s would be considered *final-form*, *user-generated*, *internal*, *build-support*, but Makefiles (along with most—but not all— user-generated files) are not part of the CVS repository, and hence are not included in our analysis like the `Makefile.am` and `Makefile.in` files are.

To understand what these attributes mean, it would be helpful to know the the number of files, the number of lines, the number of repository commits involved. For the entire project, these numbers are 1,187 files, 409,858 lines, and 7,984 commits. In Tables 3.1–3.4 we break out these numbers according to each attribute.

Finally, in Table 3.5 we filter out all but the handmade files and examine their characteristics by their **Role**. We will restrict our discussion to this table in particular.

3.1 Discussion

Table 3.5 is probably the closest measure of human effort because it focuses on the handmade files. The build support files constitute 13.7% of the overall line count and 27.5% of the overall number of changes. Build-support is the third highest category by line count and second highest by number of changes. These numbers suggest that build-support is a significant percentage of overall development activities. It also suggests that debugging the build is as complex as debugging a 18301 line program.

Developers can safely ignore the dev-generated files most of the time. However, it should be noted that end users may go where developers fear to tread. They may try to read or edit the dev-generated files. For example, a user might want change something in the build by editing a Makefile as opposed to editing Makefile.am as a developer would because they do not understand automake and autoconf. Such users will likely be overwhelmed by the size and complexity of the Makefile's in Babel. They are not written for human readability.

This metric that we devised is coarse and imperfect, but it is objective. The 27.5% of all changes seems remarkably

Visibility	files	lines	changes	max change
distributed	1,105	407,964	7,617	152
internal	82	1,894	367	
total	1,187	409,858	7,984	

Table 3.3: File counts, line counts, number of changes and max. changes by visibility

Role	files	lines	changes	max change
build-support	415	98,746	4,224	152
core-code	205	68,867	1,666	49
test-code	213	30,896	1,253	19
documentation	145	176,918	326	17
test-support	109	8,520	297	48
version-support	59	376	109	5
example-code	27	2,399	88	9
example-support	14	23,136	21	2
total	1,187	409,858	7,984	

Table 3.4: File counts, line counts, number of changes and max. changes by role

Role	files	lines	changes	max change
core-code	205	68,867	1,666	49
build-support	227	18,301	1,342	126
test-code	213	30,896	1,253	19
test-support	109	8,520	297	48
version-support	59	376	109	5
documentation	22	4,005	104	17
example-code	27	2,399	88	9
example-support	8	177	15	2
total	870	133,451	4,874	

Table 3.5: File counts, line counts, number of commits, and max commits for any file; for handmade files only broken down by **Role**

consistent with the 20% reported build overhead. The gap between these two narrows even more when considering that the first number is probably dependent on development time whereas the build overhead is measured in overall time.

On the other hand, we must acknowledge that results are indicative and not definitive. Similar detailed analysis across a range of projects would be needed before investing too much faith in the accuracy of this metric. The best we can say here is that a surprising number of changes are being committed to the CVS repository. If one equates a change with a unit of work, then a 20% build overhead for this project should not be surprising.

Chapter 4

Conclusions and Future Work

Developing and maintaining a project's build takes a significant percentage of a projects overall development time. Time spent on the build distracts from core software development and developer productivity. Pressures for more software reuse, more language interoperability, and greater portability will most likely cause this build overhead to grow as a percentage of overall person/years.

Our survey shows that people perceive the average build overhead to be around 12% with some projects having overhead up to 35%. We acknowledge that our survey measures only perceptions of the build overhead. To measure the actual overhead, we would need to monitor developer activities and maintain detailed logs about where time was being spent. Moreover, we acknowledge that the survey was not conducted over a true random sample of scientific software developers, nor did it succeed in capturing simple correlations between project characteristics and expected build overhead. This is not surprising since we started the survey just to confirm that high build overheads are not localized to our current project/location.

With software builds, the devil is in the details. To arrive at some function

$$B = F(x_1, x_2, \ldots, x_n)$$

where x_i represents independent characteristics of a software project and B is expected build overhead requires further study. We would need a much larger and more sophisticated questionnaire. We'd need to have the questionnaire web-enabled to get a significant sample of scientific software developers. More in-depth studies of other projects' software repositories would also be needed. These studies would have to be conducted in concert with a developer on the project since the process of assigning attributes to active files is labor intensive, and project specific.

This avenue of investigation is only preliminary, but it raises two very tantalizing questions that remain unanswered:

1. How many dollars per year does the current build overhead cost the DOE?
2. How close is the current build overhead to the minimum possible?

Appendix A

Raw data

	Question #'s											
LLNL #'s	1	2	3	4a	4b	5	6a	6b	7a	7b	8 (FTE's)	8 (percentage)
1	5	100.00%	4	3	0	3	7	1	5.00%		0.2	4.00%
2	5	50.00%	4	5	3	4	3	0		30.00%	0.5	10.00%
3	20	80.00%	3	4	2	7	20	0	10.00%		1.5	7.50%
9	3	90.00%	1	4	0	4	1	2	10.00%	10.00%	0.1	3.33%
11	7	100.00%	2	2	0	4	11	3		5.00%	2.5	35.71%
12		50.00%	2	5	2	2	2	1	0.00%	0.00%	0.1	
13	2		1	8	0	2	1	0			0.2	10.00%
14	4	50.00%	2	2	4	4	2	2		70.00%		
16	1.25	50.00%	3	3	1	1	3	3	0.00%		0	0.00%
17	1	20.00%	2	4	1	2	4	2	20.00%		0.2	20.00%
18	20	90.00%	4	2	2	6	10	2	70.00%		2.5	12.50%
19	6	100.00%	2	6	0	4	8	8	10.00%		0.3	5.00%
20	5	80.00%	2	3	2	4	1	0	10.00%	10.00%	0.5	10.00%
21	3	100.00%	3	6	2	1	1	1	0.00%	0.00%	0	0.00%
22	8.75	100.00%	2	3	1	3	0	0	1.00%	1.00%	0.01	0.11%
23	5	75.00%	1	3	1	3	6	1	5.00%		0.25	5.00%
24	5	100.00%	3	5	0	6	0	6	25.00%	15.00%	0.75	15.00%
25	6	66.00%	4	12	3	3	2	10	0.00%	0.00%	1	16.67%
26	5.5	100.00%	4	3	3	7			50.00%	20.00%	1	18.18%
27	30	90.00%	2	3	4	5	15	0		75.00%	5	16.67%
28	8	80.00%	4	5	0	3	10		100.00%	80.00%	1.5	18.75%
29	0.1	10.00%	5	10	2	2	1	3		10.00%	0.01	10.00%
DOE #'												
4	2	80.00%	3	6	2	6	5	2	10.00%	20.00%	0.25	12.50%
5	5	100.00%	5	10	0	6	2	20		50.00%	1.5	30.00%
6	4	70.00%	2	3		3	1		5.00%			
7	4.5	100.00%	6	9		4	2	10	20.00%		1.5	33.33%
8	1	30.00%	2	4	1	4	2	5	5.00%		0.1	10.00%
10	10	50.00%	3	9	0	4	2	10		5.00%		
30	0.5	50.00%	3	3		8	2	0	5.00%	5.00%		
31	2	50.00%	3	2	0	4	5	0	20.00%	25.00%	0.25	12.50%
32	1	50.00%	2	3		4	2	1	45.00%	5.00%	0.04	4.00%
37	2.5	75.00%	3	7	0	5	2	1	10.00%	11.00%	0.01	0.40%
39	20	100.00%	2	5	2	2	7	1	5.00%	3.00%	0.7	3.50%
Academic #'s												
15	6	100.00%	3	4		2	2	0		33.00%	1.7	28.33%
33	20	75.00%	2	4	1	3	10	10	5.00%	2.00%	0.5	2.50%
34	20	100.00%	1	2	0	2	0	8	2.00%	2.00%	N/C	
35	10	40.00%	2	6	1	5	1	3		11.25%	1.5	15.00%
36	9	25.00%	2	5	1	4	2	0	0.00%	0.00%	1.125	12.50%
38	9	80.00%	4	5	2	4	5	1	15.00%		1.8	20.00%

Table A.1: Raw numeric project data (part 1)

	Question #'s													
LLNL #'s	9 (weeks)	12 (min hours)	12 (max hours)	13a	13b	13c	13d	13e	13f	13g	13h	13i	13j	14
1		1.5	1.5	6	8			8						6
2		1	1	1	9			7	7					5
3	3	4	4	8							1			4
9	0.1	0.03	0.03	5	5									
11	28	1	1		4			7					7	6
12	2	0.1	0.1											9
13	3	0.03	0.03											4
14		0.025	0.167		8									3
16		0.33	0.33		6								10	8
17	12	0.5	1											7
18		0.25	1		6			6	7					2
19	4	0.025	0.33											
20	Yes	5	5					5						
21	0	2	2	5	5									5
22	0.2	0.08	0.08		5			6						5
23	3	0.66	1		6									6
24	50	0.5	3	3	1			2						3
25	Yes	0.66	12	5	5			5						6
26	16	0.167	5	1	1			1	1	1				1
27		0.25	2	5	5									4
28		2	2	8	6							2		8
29	16	1.5	3											9
DOE #'														
4	0.5	1	1		5			8	7					8
5	16			1	5			5	1	7				
6		0.03	0.03		6			6						
7		0.167	0.167	1	6			1	1	1	10			1
8		0.08	0.08		5			5						7
10				3	5									5
30				5	5									5
31		0.08	0.17	6	6			9						7
32		0.17	0.17		2			6						5
37	4	0.08	0.08	10	8			8						8
39	25	0.3	3		8									8
Academic #'s														
15		0.7	0.7											
33		1	2		4			2						7.31
34	0				8									8
35	30	0.5	1		6			7						4
36	12	0.05	1		1									3
38		0.6	2		7									3

Table A.2: Raw numeric project data (part 1)

	LLNL		DOE		Academic	
	Interest (16)	days spent (17)	Interest (16)	days spent (17)	Interest (16)	days spent (17)
1	8	12.5				
2	3	5				
3	8	15				
4	3	1.5				
5	8	250				
6	7	4				
7	5	3				
8	10	3				
9	10	88				
10	5	5				
11	10	3				
12	8	0.25				
13	9	2				
14	7	0.5				
15	1	1				
16	8	2				
17	9	2				
18	8	0.5				
19	10	5				
20				0		
21			5	1		
22			10	5		
23			3	3		
24			10	22		
25			9	0.25		
26			6	2		
27			2	0.25		
28			4	0.02		
29			5	2		
30					5	10
31					3	
32					7	5
33					9	0.04
34			9	0.5		
35					7	1
36			8	15		
37						
38						
39						
40						

Table A.3: Raw data for personal questions

Bibliography

[1] The Common Component Architecture (CCA) Forum Website. `www.cca-forum.org`.

[2] components@llnl.gov Website. `www.llnl.gov/CASC/components`.

[3] Tom Epperly, Scott Kohn, and Gary Kumfert. Component technology for high-performance scientific simulation software. In *Working Conference on Software Architectures for Scientific Computing Applications*, Ottowa, Ontario, Canada, October 2000. International Federation for Information Processing. Also available as Lawrence Livermore National Laboratory technical report UCRL-JC-140549.

[4] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001. Also available as Lawrence Livermore National Laboratory technical report UCRL-JC-140349.

[5] Gary Kumfert, Bill Bosl, Tamara Dahlgren, Tom Epperly, Scott Kohn, and Steve Smith. *Achieving Language Interoperability Using Babel*. CASC/ISCR Workshop on Component and Object-Oriented Technologies for Scientific Computing, Wente Vineyards, Livermore, CA, July 2001.