Appears in Proc 14th Working Conference on
Reverse Engineering (WCRE), pp. 60-69, 2007.

# Recovering Maintainability Effort in the Presence of Global Data Usage*

Jason W. A. Selby, Fraser P. Ruffell, Mark Giesbrecht and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

E-mail: {j2selby,fruffell,mwg,migod}@uwaterloo.ca

## Abstract

*As the useful life expectancy of software continues to increase, the task of maintaining the source code has become the dominant phase of the software life-cycle. In order to improve the ability of software to age and successfully evolve over time, it is important to identify system design and programming practices which may result in increased difficulty for maintaining the code base.*

*This study attempts to correlate the use of global variables to the maintainability of several widely deployed, large scale software projects as they evolve over time. Two measures are proposed to quantify the maintenance effort of a project. The first measure compares the number of CVS revisions for all source files in a release to the number of revisions applied to files where the usage of global data is most prevalent. A second degree of change is characterized by contrasting the amount of source code that was changed overall to the changes made to those source files which contain the majority of the references to global data.*

*We observed a strong correlation between the number of revisions to global variables references and lines of code to global variable references. In all cases the correlation between the number of revisions and global variable references was stronger. This provides evidence that a strong relationship exists between the usage of global variables and both the number and scope of changes applied to files between product releases.*

## 1. Introduction

The maintenance phase of the software life cycle has been identified as being the dominant phase in terms of both time and money [22]. Logically, one could point to the code size, structure, age, complexity, development language and the quality of the internal documentation as being the key indicators to the maintainability of a project [11]. However, few empirical studies have examined the degree to which these factors impact project maintainability. Evidence linking many of these measures to an approximation of the maintenance effort for a product has found that the correlation is weak at best for many of these factors [10, 3, 7, 1, 9]. In order to improve the ability of software to age and successfully evolve over time, it is important to identify system design and programming practices which may result in increased difficulty for maintaining the code base.

The decomposition of complex software systems into individual modules that group together related concepts and tasks improves program comprehension and maintenance. Ideally, modules are designed to exhibit a low degree of coupling between other modules and a high degree within the same module. However, to enable inter-module communication, some form of coupling must exist. Common coupling is an undesirable form of coupling introduced when modules reference the same global data [23] (and worse, common coupling can be *clandestine* in the sense that it can be introduced without explicit changes to a module; see [15]). The many reasons why global variable usage is considered harmful and should be avoided are well documented in [8, 12, 16, 23]. Examples of the unanticipated side effects of global variable usage include hidden aliasing, namespace pollution, and even hampering code reuse across projects.

We are interested in an empirical evaluation of how the use of global variables may affect software evolution. In this paper, we investigate two hypothesis concerning the impact of global variables on maintainability:

**H1** The use of global variables leads to code that requires more maintenance.

**H2** The use of global variables leads to code that is difficult to comprehend.

We interpret these hypotheses more directly by aligning the files (modules) of the system source code base by the number of references to global variables. We then observe

*This research was supported in part by a Natural Sciences and Engineering Research Council of Canada Strategic and Discovery grants.

the maintenance effort for all files by tracking the number and size of CVS commits. In particular, we test the following two hypotheses:

**H3** Files with a greater number of global variable references change more often than files with fewer references.

**H4** Files with a greater number of global variable references entail larger change deltas than files with fewer references.

If we find support for H3, we take this as evidence that H1 is true, and similarly if we find support for H4, we take this as evidence that H2 is true. Using our approach, we analyzed binaries from many popular open-source projects including Emacs, GCC, GDB, Make, Vim, and PostgreSQL.

The remainder of the paper is organized in the following manner. Section 2 details how our study was performed and the projects that we examined. In Section 3 we report and discuss the results that were obtained. Section 5 discusses related research that has examined the usage of global data and tools which have been developed to extract software artifacts from CVS repositories. Finally, we conclude and point out promising future directions based on this work in Section 6.

## 2 Methodology

This section begins by describing our approach to tracking global variable usage and to measuring the maintainability effort throughout the lifetime of a project. We then give an overview of the systems that were examined in this case study.

### 2.1 Extracting Global Variable Usage Data - `gv-finder`

Our initial examination of the evolution of global data throughout the lifetime of many open source projects resulted in the creation of a linker-like tool capable of extracting global variable usage data from object files [14]. This tool, named `gv-finder`, intercepts relocatable Executable and Linking Format (ELF) object files (non-stripped) at the linking stage of the compilation process, analyzes the files and then passes the files on to the actual linker. This process of collecting global variable information fits seamlessly into the build process and enables the analysis of evolutionary trends over entire product lifetimes.

Examination of the symbol and relocation tables in the object files yields the names of all global variables, the module in which each is defined, as well as the names of all modules which reference each global. We classify each reference as either *true*, *static* or *external*. A true global variable contains a "global" entry in the symbol table and is what one typically thinks of as a global variable – a user defined variable storage which can be referenced in any module. Usage of a true global variable is considered the most dangerous due to the implicit coupling between any and all modules which reference the same global symbol [23].

Static globals are marked as "local" in the symbol table and therefore can be referenced anywhere inside the file in which it is defined (for example, a C file scoped variable). The use of a static global does not introduce clandestine coupling however, it does carry the other potential drawbacks of using global variables. An external global is denoted by an "undefined" symbol table entry and is a symbol imported from a library (for example, `printf()` or `stdout` from the C standard library). Differentiation of external references between function calls and variable references is performed by disassembling the instructions which contain a reference to global data. If the instruction is a `jmp` or `call`, then the usage is considered a function otherwise, it is a variable. All of the data presented in this paper is restricted to references to true global data. Further details on `gv-finder` can be found in [14].

The integration of `gv-finder` into the linkage stage enables us to bypass build environment issues and, more importantly, to base our results solely upon the actual modules included in the final result. Our analysis is restricted to the specific global variable references that are present in the final executable and not those present in the entire source code base. This eliminates the possibility of counting equivalent global variable references multiple times that are not present in the executable due to reasons such as conditional inclusion of object files for specific machine architectures and operating systems. The disadvantage of our link-time analysis is that `gv-finder` requires a successful compilation of the target executable. When analyzing older releases (for example, we studied versions of Emacs over ten years old), the build process often failed due to dependencies on deprecated APIs (either library or OS). Rather than omit releases which failed to build, we deployed four different machines each recreating a specific and older build environment needed to satisfy various releases. The use of different systems introduced a minimal amount of error, since all of the machines are of the same architecture (x86, Linux), and therefore are equally impacted by external factors affecting the source code (such as conditional compilation).

### 2.2 Measuring Maintainability Effort

#### 2.2.1 The Concurrent Versions System

We propose two measures to answer our postulates, both of which harness information extracted from the Concurrent Versions System (CVS), a popular source code management system that tracks the various changes made to files and en-

ables concurrent development by many developers [2]. For example, mining information from a CVS repository can yield the number of revisions made to each file between each product release. This then enables the comparison of the number of CVS revisions for files in which the usage of global data is most prevalent to those which have fewer or no references. CVS is also able to report the number of lines changed between two revisions of a file. In an attempt to characterize the scope of the changes performed on a file, we extract this information from the repository and compare the total lines changed in files which have a large number of references to global variables to other files in the system.

CVS uniquely identifies each version of a file through the use of a revision number. The initial version of a file is assigned the revision number 1.1 after which, each time an update to the file is checked into the repository, a new number is assigned to the file (for example, 1.2). CVS revision numbers are internal to the system and have no relationship with software releases. Instead, symbolic names or tags are applied to the set of files which constitute a particular release of a system. Typically, all of the files in a repository are assigned a new tag at every release point, creating a CVS snapshot of the code which can be later referenced. Unfortunately, not all releases of the various projects that we examined were tagged. For releases which were tagged, identifying the revision number of each file was simple. However, if no release tag was present, we resorted to a brute force approach which compared the actual source code files contained in the release with each revision of the file in the repository in an attempt to find a match. In some cases (typically in early releases of a project when the development process was not formalized) we were unable to find a match for all of the files in a release and therefore limited our results to releases in which we were able to match at least 80% of the source files which constitute the binary executable examined.

Since the tagging of the source files at specific points is managed by developers and not CVS, each project that was examined had a different process in place to record the merging of branches into the main line (if this was even recorded at all in the repository). This posed a problem in uniformly comparing the number of revisions made to a file between two releases in the presence of branching. To overcome this issue we recorded two different release counts. The first is a conservative lower-bound approach which does not count revisions along a branch between two releases, thereby assuming that every branch is in fact a dead branch. Our second method is an optimistic upper-bound approach and counts every revision along a branch and possibly even follows other branches that exist between the two releases. For example, suppose that for some file the revisions 1.4.2.1, 1.4.2.2, 1.4.2.3, and 1.5 exist between two releases. If we identified that the first release included revision 1.4.2.1 and the later 1.5 then the lower-bound approach would report that a single revision was made between releases, while our upper-bound approach would find that three revisions were applied (the lower- and upper-bound approaches are later referred to as *no-branch* and *branch* respectively, in the graphs presented in Section 3). Even though the lower- and upper-bound approaches may respectively under- or over-estimate the maintainability effort applied to a file, we found that in practice there was very little difference between the two approaches.

### 2.3 Case Study

Using our approach, we analyzed the primary binaries from many popular open-source projects, including Emacs, GCC, GDB, Make, Vim, and PostgreSQL over a significant span of their developments. Specifically, we examined the following binaries (the number of releases of each binary studied and the time span of the releases is also reported):

**`temacs`** The `C` core of the GNU Emacs editor which contains a LISP interpreter and basic I/O handling (10 releases, 14 years) [17].

**`cc1`** The GNU `C` compiler (gcc) not including the libraries which are linked with it (29 releases, 7 years) [20]. We included only the "hand-written" code and not the extensive amount of automatically generated code that is incorporated into `cc1`.

**`libbackend.a`** A library linked with gcc which performs code analysis, optimization and generation (27 releases, 7 years) [20].

**`libgdb.so.a`** A library which exports the functionality of GDB through an API (11 releases, 7 years) [19].

**`make`** The GNU utility which automates the compilation process of source code (18 releases, 16 years) [18].

**`postgres`** The back-end server of the PostgreSQL relational database management system (10 releases, 11 years).

**`vim`** A popular open-source text editor modeled after VI (9 releases, 8 years).

## 3. Results

In order to visually compare the maintenance effort applied to the source files which contain many references to global variables to those that do not, we graphed the average number of revisions for all files along with the average revisions for the files with 50% of the references to global variables, and for the files with 100% of the references to global

variables (the files composing 50% of the global variable references were selected by sorting the files by the number of references and choosing the first files which sum to 50% of the total number of global variable references). Similarly, we graphed the normalized average number of lines changed in each release. If the presence of global variables is in fact detrimental to the comprehension and modification of code then we would expect a greater number of changes would be required to maintain the files containing a large number of references to global data compared to those files which have fewer references (H3) (although previous research [10] has differentiated between the various forms of maintenance, we do not in this paper). Not only do we expect the presence of global variables to increase the number of modifications required between two releases of a product, but we would also expect that the usage of global variables would increase the scope of the modifications, thereby increasing the amount of source code that is changed (H4).

Table 1 reports the details of the initial and final releases examined for each project. In an attempt to limit the amount of graphs presented we selected two representative projects and direct the interested reader to `http://plg.uwaterloo.ca/~j2selby/wcre07-results.html` for the omitted graphs. The results for `cc1` (Figures 1 and 2) and `postgres` (Figures 3 and 4) illustrate our findings. It should be noted that no significant difference between the upper and lower-bound approaches was found for `temacs`, `libbackend`, `make`, `postgres` and `vim` and therefore to improve the clarity of the graphs, the upper-bound (branch) is omitted.

Every effort was made to include all releases, both major and minor, of each project that we examined. However, some releases were either unanalyzable (due to either failed compilation or difficulties in extracting the CVS information) or omitted (a product release was issued but the files that constitute the target that we examined were unchanged). One special incident was encountered in the analysis of `vim` and `libbackend`. The results for these targets were skewed by the fact that both include a `version.c` source file which has a disproportional number of revisions and lines changed in comparison to other files (for `libbackend` this file simply stores the version number of the release in a string and similarly for `vim`). We therefore omitted this file from our analysis, however, this was the only special circumstance.

As expected, examination of the graphs illustrates that at almost all points both the number of revisions and the total number of lines of code changed are higher for the subset of files which contain a greater number of references to global variables.

The only instances where the graphs deviate from this pattern when contrasting lines of code changed to global variables is for `make` and `libbackend`. In only one instance did comparing the number of file revisions to global variable usage not follow the trend which we envisioned, namely `vim`. Further examination of these outlying points provided some insight into why they were contrary to our hypotheses. We found that for six of the seventeen releases of `make` examined, the average number of file revisions for all of the files containing a global variable reference was higher than that of the files containing the top 50% of the global variable references. At each of these six points a small group of files (2–3) which are just outside of the 50% are heavily modified. Interestingly, it is always the same small set of files which requires substantial changes, possibly indicating their importance to the system or that they require complex modification. Investigation of the last three releases of `vim` discovered the existence of three files which contain zero references to a global variable however, they were changed slightly above the average number of revisions applied to all files. We were unable to identify a single cause for the greater number of lines of code changed for the files containing at least one global variable reference at the four spikes in `libbackend`. We plan to examine this in greater depth in order to find the exact cause of this behaviour.

In an attempt to track the evolution of global variable usage throughout each of the projects we identified the top five files and functions which contain the greatest number of references to global variables in each release. Furthermore, we also examined the five globals that were the most heavily referenced in each product release. `libgdb` exhibited the least amount of fluctuation with the same four files, functions and variables remaining in the set of top five over all of the releases examined. `temacs` and `vim` were also found to be quite stable when considering files and variables. In both, only one file was displaced from the top five set while three variables remained heavily referenced in `temacs` and four in `vim`. Greater variation was displayed in the functions which contained the most global variable references. In `temacs` only one function remained in the top five, while two of five remained fixed in `vim`.

An interesting aspect of examining `cc1` and `libbackend` from GCC is that most of the `libbackend` code was split off from `cc1` in release 3.0 of GCC. In the creation of `libbackend` the five files containing the greatest number of references to global variables was extracted from `cc1`. After the split, the set of files which relied most heavily on global variables remained fairly fixed with three files remaining in the top five in `libbackend`, and four of the five in `cc1`. The specific global variables which were referenced most heavily in `cc1` were also the highest used in `libbackend` and continued to be over all releases examined. There was greater variability exhibited in `cc1` with only two of the top five global variables remaining in the set after the split.

**Table 1.** This table reports the initial and final releases examined for each binary as well as the number of thousands of lines of source code (KLOC), the total number of files examined, and the number of files that contain the greatest number of references to global variables that cumulatively account for 50%, and 100% of all global variable references respectively.

| Binary | Release | KLOC | Total Files | Total Files 50% refs | Total Files 100% refs |
|---|---|---|---|---|---|
| `temacs` | 19.25 | 109 | 57 | 10 | 53 |
| | 21.4 | 198 | 67 | 9 | 61 |
| `cc1` | 2.95 | 232 | 67 | 10 | 59 |
| | 4.1.0 | 10 | 21 | 3 | 16 |
| `libbackend` | 3.0 | 231 | 79 | 11 | 72 |
| | 4.0.3 | 233 | 152 | 20 | 133 |
| `libgdb` | 5.0 | 144 | 104 | 12 | 91 |
| | 6.5 | 217 | 208 | 15 | 45 |
| `make` | 6.63 | 13 | 16 | 3 | 15 |
| | 6.81 | 24 | 24 | 3 | 16 |
| `postgres` | 1.02 | 142 | 236 | 18 | 177 |
| | 8.13 | 355 | 358 | 26 | 292 |
| `vim` | 5.5 | 126 | 39 | 8 | 35 |
| | 6.4 | 217 | 47 | 11 | 44 |

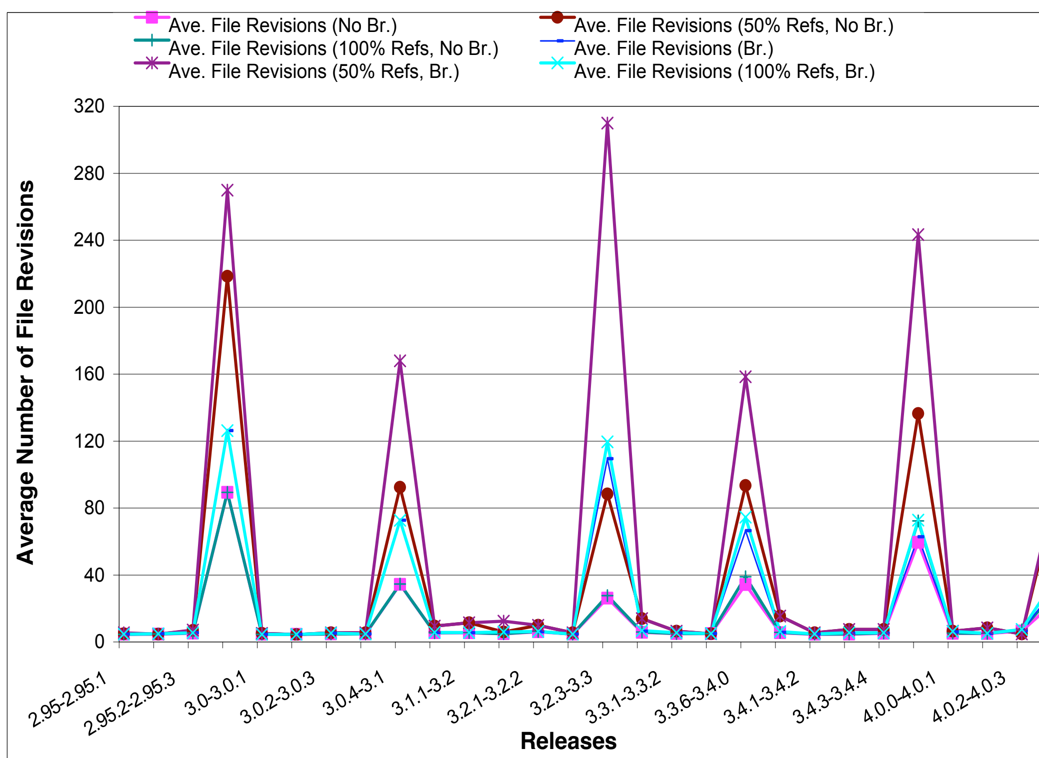


**Figure 1.** A comparison of the number of CVS file revisions for `cc1` from GCC.

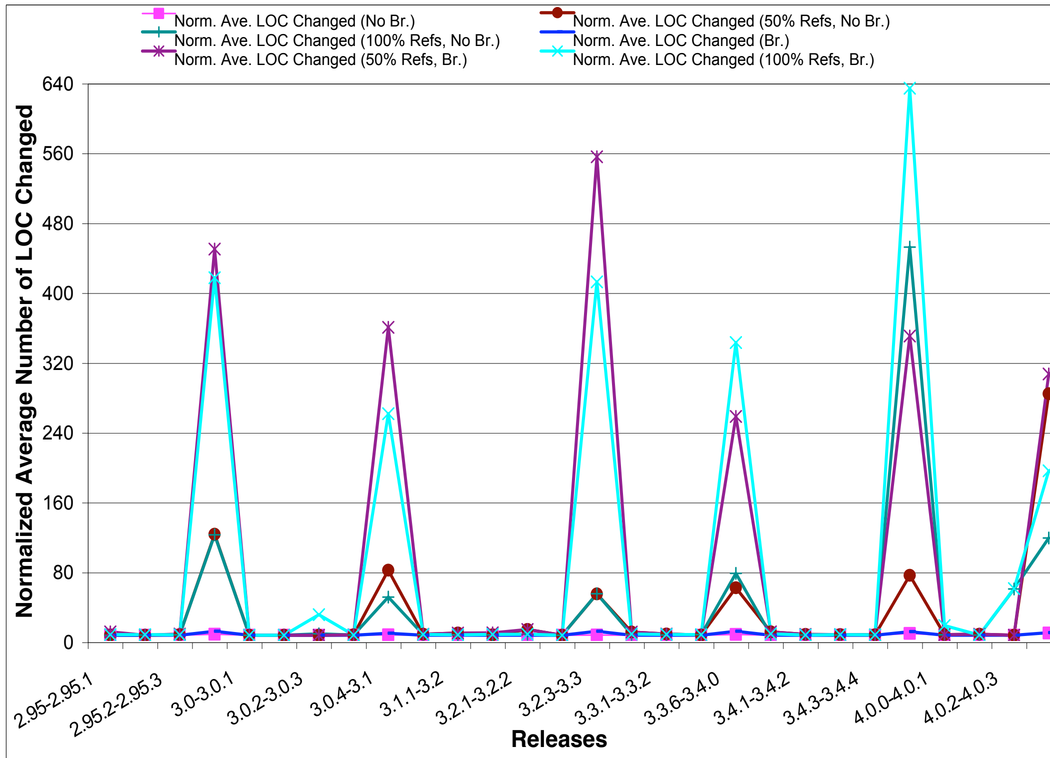


**Figure 2.** A comparison of the normalized number of lines changed between releases of `cc1` from GCC.

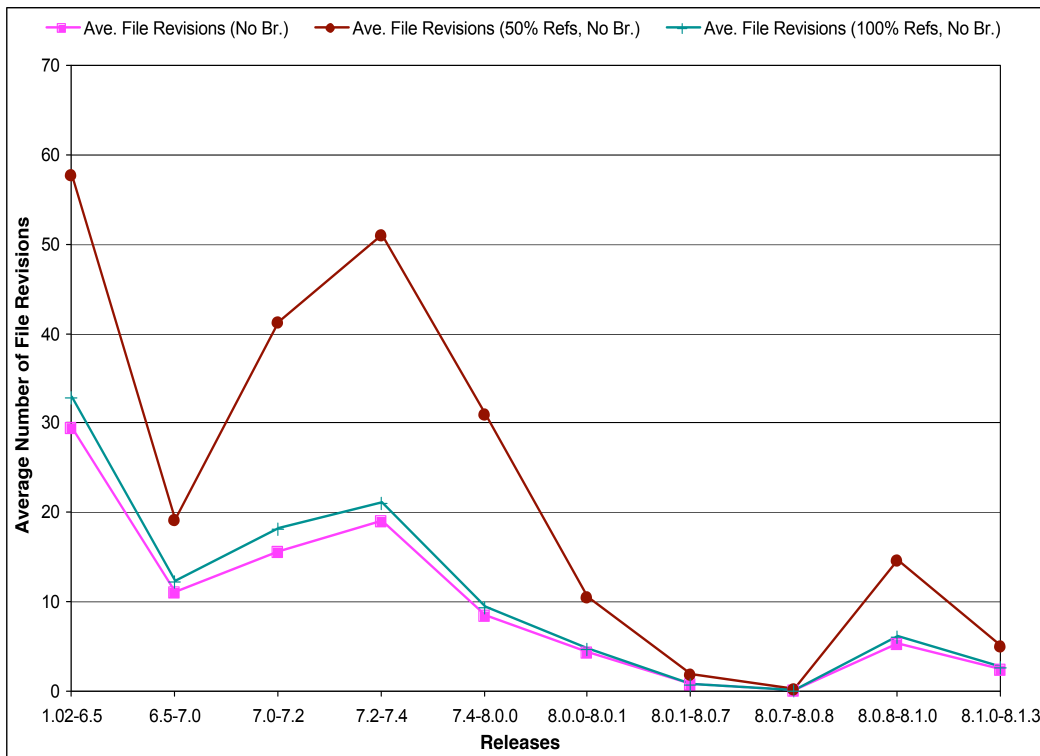


**Figure 3.** A comparison of the number of CVS file revisions for `postgres`.

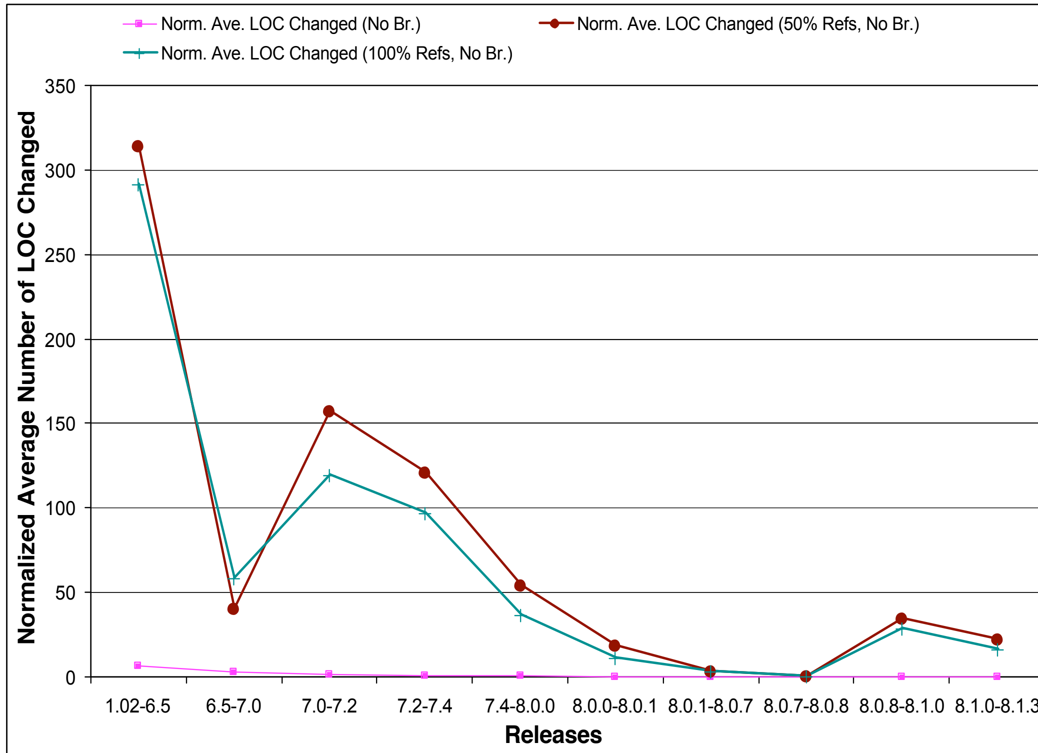


**Figure 4.** A comparison of the normalized number of lines changed between releases of `postgres`.

The set of top five files and functions remained relatively constant in both `make` and `postgres`, with three remaining in the top five over the entire lifetime that we examined. However, the most heavily referenced global variables fluctuated greatly, with none of the top five in the initial release remaining in the top five set at the final release.

Although the graphs appear to substantiate the link between global variable usage and maintenance effort, further evidence of the connection is required. Therefore, we calculated the correlation coefficients ($r$ values) of both measures. Calculation of an $r$ value enables one to evaluate the degree of correlation between two independent variables (specifically, revisions to global variables and total lines changed to global variable references). Table 2 lists the results of correlating the number of references to global variables in a file to the number of revisions checked into CVS ($r(Rev, Ref)$) and also for the total lines of code changed to the number of references to global variables ($r(Lines, Ref)$). The correlation coefficients in bold represent instances of close correlation between the two variables for an acceptable error rate of 5% ($\alpha = 0.05$), however, almost all were within a $1\%$ error rate. Strong correlation was found between both revisions to references and lines to references. However, in all cases the correlation between the number of revisions and global variable references was closer. Although, this does not establish a cause and effect relationship it does provide evidence that a strong relationship exists between the usage of global variables and both the number and scope of changes applied to files between product releases. Furthermore, this provides support for our hypotheses that files which contain a greater number of references to global variables require more changes (H3), and that these changes correspond to the modification of more lines of code (H4). Extrapolation from H3 and H4 provides evidence for the acceptance of our original hypotheses that global variable usage both increases maintenance (H1), and impairs comprehension (H2).

## 4. Threats to Validity

We should note the possible threats to the validity of our study. As stated earlier, `gv-finder` requires a successful compilation of the target executable in order to perform its analysis. In the worst case this required commenting out the offending lines of code (this, however, occurred fairly infrequently and only for small code segments). Additionally, since the build environment has changed over the course of the projects lifetime, we deployed four different machines, each recreating a specific and older build environment needed to satisfy various releases. The use of different systems introduced a minimal amount of error, since all of the machines are of the same architecture (x86, Linux),

**Table 2.** Results of correlating the number of revisions made to a file between releases with the amount of references to global variables within the file ($r(Rev, Ref)$), and for the total number of lines changed in a file to its number of references to global variables ($r(Lines, Ref)$). Correlation coefficients in bold identify instances of a close correlation. $N$ is the number of pairs examined.

| Binary | N | r(Rev, Ref) | r(Lines, Ref) |
|---|---|---|---|
| temacs | 520 | **0.27** | **0.16** |
| cc1 | 642 | **0.16** | **0.09** |
| libbackend | 2822 | **0.12** | **0.08** |
| libgdb | 1563 | **0.44** | **0.39** |
| make | 337 | **0.42** | **0.31** |
| vim | 336 | **0.33** | **0.27** |
| postgres | 3156 | **0.24** | **0.22** |

and therefore are equally impacted by external factors affecting the source code (such as conditional compilation). Although this study examined a wide spectrum of software products, all of the projects are open-source (even further almost all are developed by GNU) and therefore it is not clear that our findings are applicable to proprietary software.

In posing our hypotheses we equated the presence of global variables to increased maintenance costs in the form of both the number, and the size of the changes performed. However, other explanations are also possible. For example, a file that changed frequently might be an architectural "hotspot" for the addition of new features; thus frequent changes may be a sign of successful growth rather than poor design. Similarly, large deltas might mean that the system's design was sufficiently robust to allow for the addition of new functionality. However, in the absence of a way of automatically categorizing the intent of the individual changes, we assume that most changes are due to "fixing" rather than adding new features.

Finally, when examining the extent of the modifications performed we normalized the delta values by the file size. However, we did not normalize the number of changes to the size of the file. In future work we plan on taking this into account and normalizing the number of changes by the amount of references to global variables per line of source code.

## 5. Related Work

A tool similar to `gv-finder` is described in [21] which uses the output of `objdump` to gather global symbol information. We chose to extract the data ourselves since we already had an existing infrastructure for analyzing ELF object files and also to improve efficiency.

Schach *et al.* [15] and later Yu *et al.* [23] examined global variable usage in the Linux kernel. Their initial work in [15] discovered that slightly more than half of all modules examined suffered from some form of clandestine coupling. The latter work in [23] continued the examination of clandestine coupling between kernel and non-kernel modules in Linux. Applying definition-use analysis from compiler theory [13], they identified all modules which defined (wrote) a global variable and the others which referenced (read) each global. They found that a large number of global variables are defined in non-kernel modules and are referenced in a kernel module. Given the lack of control over non-kernel modules by kernel developers [15, 23] raised concerns over the longevity of Linux, suggesting that maintainability issues might arise given the common coupling found to exist between kernel and non-kernel modules. However, the analysis based simply on the bulk number of definitions and uses might be misleading. A more conclusive examination could use definition-use chains [13]. Def-use chains connect uses of a variable with their exact point of definition. Using a code analysis tool to construct the def-use chains, we could then identify the chains which are formed from the definition of a variable in a non-kernel module and then later used in a kernel module.

The application of data mining to various artifacts of the software development process to discover and direct evolution patterns has recently received extensive treatment, most notably in [4, 6, 5, 24]. A common measure of software change throughout much of this research is based upon the number of CVS updates to a file (CVS release numbers) and the total lines of code changed between releases.

Epping *et al.* [3] examined the connection between vertical (specification) and horizontal (inter-module) design complexities and maintainability (change) effort during the acceptance and maintenance phases of two `FORTRAN` systems. Specifically, in regards to global variable usage they examined the number of globals defined, the actual number of globals referenced and maintainability, which is characterized by change effort. The change effort metric was further categorized as being isolation effort (identifying which modules require modification), implementation effort (develop, program and test the change) or locality (the number of modules also requiring modification). Additionally, the subset of all the tasks performed during the maintenance phase which were bug fixes was identified. Results for all changes (bug and enhancement) in the maintenance phase indicated a correlation between change isolation and to both the number of global variables and the amount of references to globals. However, no link was found to exist in implementation effort or locality. When focusing strictly upon maintenance phase bug fixes, both change isolation and implementation effort were found to correlate to the usage of

global variables.

Harrison and Walton [7] applied a similar metric for maintainability as in this study to a large number of small legacy FORTRAN programs mining three years of CVS data. The measures examined included lines of code and structural complexity (number of GOTO statements and cyclomatic complexity). Their findings indicated that lines of code offered only minor insight into future maintenance costs while no correlation between any of the structural characteristics of the programs and maintenance costs were found to exist. In contrast to [3] and [7], our analysis is based upon a much larger data set encompassing many releases of seven large systems, different measures of maintainability effort are used and also the differing semantics of global data in FORTRAN compared to C.

Zimmermann *et al.* [24] applied data mining to CVS repositories in order to determine various source components (for example, files, functions and variables) which are consistently changed in unison. Integration of their tool into an IDE enabled them to suggest, with a reasonable degree of accuracy, other parts of the code which might need to be modified given a change to an element in which it has been determined to have been changed together in the past. Similar work appeared in [6], however their work focused on the higher-level granularity of classes.

It is commonly believed that by employing automatic code generators and packaged libraries, the initial software development costs could be decreased and this reduction of effort would continue into the latter maintenance phase of a project. Banker, Davis and Slaughter [1] examined how the use of these affected software complexity, which in turn increases the difficulty in performing maintenance tasks. This perception was confirmed for the use of packaged libraries for their sample (they examined the application of 29 perfective maintenance tasks to 23 COBOL programs). However, contrary to intuition, the use of automatic code generators actually lead to an increase in the amount of time spent on maintenance tasks. This is an interesting result in consideration of the projects that were examined in this study. The data collected for cc1 was limited to the "hand written" code rather than the extensive amount of automatically generated code. Comparison of the usage of global variables in the auto-generated code to that of hand-written code and isolation of which part of the code is modified could be another approach to investigating this contrary result.

## 6. Conclusions

In this paper we examined the link between the use of global variables and software maintenance effort. Harnessing information extracted from CVS repositories, we examined this link for seven large open source projects. We proposed two measures of software maintenance; specifically, the number of revisions made to a file and the total lines of code changed between two releases. Examination of the graphs illustrated that at almost all points both the number of revisions and the total number of lines of code changed were higher for the subset of files which contained a greater number of references to global variables. Further investigation using statistical analysis revealed a strong correlation between both the number of revisions to global variable references and lines of code changed to global variable references. However, in all cases the correlation between the number of revisions and global variable references was stronger. Although this does not establish a cause and effect relationship, it does provide evidence that a strong relationship exists between the usage of global variables and both the number and scope of changes applied to file between product releases. Furthermore, the resulting correlations offer support for our hypotheses that global variable usage reduces maintainability and impairs comprehension. These results suggest that the use of global variables should be avoided when possible, thereby improving the ability of software to age and successfully evolve over time.

## References


[1] R. D. Banker, G. B. Davis, and S. A. Slaughter. Software development practices, software complexity, and software maintenance performance: a field study. *Manage. Sci.*, 44(4):433–450, 1998.

[2] P. Cederqvist. *Version Management with CVS*, 2005. Available at http://ximbiot.com/cvs/manual.

[3] A. Epping and C. Lott. Does software design complexity affect maintenance effort? In *Proceedings of the NASA/GSFC 19th Annual Software Engineering Workshop*. Software Engineering Laboratory: NASA Goddard Space Flight Center, 1994.

[4] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:385–403, November 2004.

[5] M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall. Mining evolution data of a product family. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[6] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[7] M. S. Harrison and G. H. Walton. Identifying high maintenance legacy software. *Journal of Software Maintenance*, 14(6):429–446, 2002.

[8] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] C. F. Kemerer and S. A. Slaughter. Determinants of software maintenance profiles: an empirical investigation. *Journal of Software Maintenance*, 9(4):235–251, 1997.

[10] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[11] J. Martin and C. L. McClure. *Software Maintenance: The Problems and Its Solutions*. Prentice Hall Professional Technical Reference, 1983.

[12] S. McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, second edition, 2004.

[13] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[14] F. P. Ruffell and J. W. A. Selby. The pervasiveness of global data in evolving software systems. In L. Baresi and R. Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2006.

[15] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and J. Offutt. Quality impacts of clandestine common coupling. *Software Quality Control*, 11(3):211–218, 2003.

[16] S. R. Schach and A. J. Offutt. On the non-maintainability of open-source software position paper. *2nd Workshop on Open Source Software Engineering*, May 2002.

[17] R. M. Stallman. *GNU EMACS Manual*. Free Software Foundation, 2000.

[18] R. M. Stallman, R. McGrath, and P. D. Smith. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, 2004.

[19] R. M. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB:The GNU Source-Level Debugger*. Free Software Foundation, 2002.

[20] R. M. Stallman and the GCC Developer Community. *Using GCC: The GNU Compiler Collection Reference Manual*. Free Software Foundation, 2003.

[21] H. S. Teoh and D. B. Wortman. Tools for extracting software structure from compiled programs. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, page 526, Washington, DC, USA, 2004. IEEE Computer Society.

[22] J. v. Vliet. *Software Engineering – Principles and Practice*. John Wiley & Sons, New York, New York, USA, 2nd edition, 2000.

[23] L. Yu and K. Chen. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Trans. Software Eng.*, 30(10):694–706, 2004.

[24] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.