The Pervasiveness of Global Data in Evolving Software Systems*

Fraser P. Ruffell and Jason W.A. Selby

School of Computer Science, University of Waterloo,
Ontario, Canada, N2L 3G1
{fruffell, j2selby}@uwaterloo.ca

Abstract. In this research, we investigate the role of common coupling in evolving software systems. It can be argued that most software developers understand that the use of global data has many harmful side-effects, and thus should be avoided. We are therefore interested in the answer to the following question: if global data does exist within a software project, how does global data usage evolve over a software project's lifetime? Perhaps the constant refactoring and perfective maintenance eliminates global data usage, or conversely, perhaps the constant addition of features and rapid development introduce an increasing reliance on global data? We are also interested in identifying if global data usage patterns are useful as a software metric that is indicative of an interesting or significant event in the software's lifetime.

The focus of this research is twofold: first to develop an effective and automatic technique for studying global data usage over the lifetime of large software systems and secondly, to leverage this technique in a case-study of global data usage for several large and evolving software systems in an effort to reach answers to these questions.

1 Introduction

A focus of the software engineering discipline has been, and continues to be, the development and deployment of techniques for yielding reusable, extensible, and reliable software [3]. One proven approach toward obtaining these goals and others, is to develop software as a collection of independent modules [9, 6]. This technique is especially effective when the individual modules experience a low-degree of inter-dependence or *coupling* [18, 11]. Modules which are self-contained and communicate with others strictly through well-defined interfaces are not likely to be affected by changes made to the internals of other unrelated components.

Although designing software which exhibits a low degree of coupling is highly desirable, if the modules of a software system are to communicate at all, some form of coupling must exist. In [11] the following seven types of coupling are

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

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 396–410, 2006.
© Springer-Verlag Berlin Heidelberg 2006

defined in increasing severeness: no coupling, data coupling, stamp coupling, control coupling, external coupling, common coupling, and content coupling.

The focus of this paper is on the second most undesirable form: common coupling. This manifestation of coupling implicitly occurs between all modules accessing the same global data. In [20], this form of coupling is referred to as *clandestine* coupling. Many different programming languages, old and new alike, provide support for global data, and as illustrated later in this paper, common coupling is rampant in many large software systems.

In this research, we investigate the role of global data in *evolving software systems*. It can be argued that most software developers understand that the use of global data has many harmful side-effects and thus should be avoided. We are therefore interested in the answer to the following question: if global data *does* exist within a software project, how does global data usage *evolve* over a software project's lifetime? Perhaps the constant refactoring and perfective maintenance eliminates global data usage or, conversely, perhaps the constant addition of features and rapid development introduce an increasing reliance on global data? We are also interested if global data usage patterns are useful as a software metric. For example, if a large number of global variables are added across two successive versions, is this indicative of an interesting or significant event in the software's lifetime? The focus of this research is twofold: first to develop an effective and automatic technique for studying global data usage over the lifetime of large software systems and secondly, to leverage this technique in a case-study of global data usage for several large and evolving software systems in an effort to attain answers to these questions.

This paper is organized as follows. Section 2 discusses the many pitfalls and possible reasons for using global data. Section 3 then presents our thoughts and expectations on global data usage in the three software systems examined. In Section 4, an in-depth discussion on our tool `gv-finder` is presented. Section 5 provides an overview of the systems examined in this study, followed by the results and analysis of applying `gv-finder` to these systems in Section 5. Finally, Section 7 concludes.

2 Global Data

The notion of *scope* is intrinsic to the declaration of source code entities (e.g. class, function, datum) in all programming languages. Scope is simply a maximal region of code for which the declared source code item is bound to [6]. Depending on language semantics, the scope may vary from a single compound statement (local scope), to any and every source file in a software project (global scope). The focus of this paper is on the set of variables within an application's source code that are declared with a global scope. As discussed later in Section 4 we further build upon this definition to include other variables of interest.

Global variables can be used indiscriminately in any module within a software project, and thereby all modules referencing the same global data are implicitly coupled. For this reason, it is well known that the use of global variables poses

a real and serious threat to software maintainability. The resulting rampant coupling can greatly impair program comprehension through a wide range of unanticipated side-effects; from hidden aliasing problems, namespace pollution, to even hampering code reuse across projects. Without a full understanding *all* of these and other less obvious implications of using global data, misconceptions about the safety of read-only access to globals, or the judicious use of file scope globals (e.g. statics in `C`) will continue to exist. For an in-depth discussion as to why global variables are harmful, the reader is referred to [6, 9, 17, 20].

Despite the well known drawbacks of using global data, there are a small number of *valid* reasons justifying their use. For example, global data can be used to emulate named constants and enumerations in those languages which do not support them directly [9]. However, situations which require the use of global data are rare and can almost always be avoided.

Previous work by Briand *et al.* [2] found that the degree of common coupling inside of a system is related to fault-proneness. Yu *et al.* [20] and Schach *et al.* [16] also examined common coupling in terms of the effects on maintainability and quality, respectively.

The application of data mining to various entities of the software development process to discover and direct evolution patterns has recently received extensive treatment, most notably in [4, 13, 21].

3 Exploration of Global Variables in Software Evolution

Two opposing views of software evolution exist. In the first view, early releases of a software project are seen as pristine, and that as the software *ages*, entropy takes hold and it enters into a constant state of decay and degradation [12]. Accordingly, one may hypothesize about the pervasiveness of global data with this view in mind:

> *Since evolving software is in a perpetual state of entropy, the degree of maintainability will decrease partially due to an increase in both the number and usage of global variables within an aging software project.*

Conversely, another view is that software is in a constant state of refactoring and redesign and, along with perfective maintenance, one can conclude that the early releases of a software project are somewhat unstructured, and as the project *ages* the design and implementation become more stable and mature. With this view in mind, one can suggest the following about the pervasiveness of global data in evolving software:

> *As software evolves in an iterative development cycle of constant refactoring and redesign, the degree of maintainability will increase partially due to an increase in both the number and usage of global variables within a growing software project.*

Although both hypotheses are convincing when viewed in isolation, it appears to us that it is more likely that neither will apply uniformly to *all* evolving soft-

ware systems. Instead, we propose that the defining characteristics of each software system (such as the development model, development community, relative age, project goals, etc.) are the factors determining which viewpoint is more influential. In particular, we adopt the three-pronged classification of Open Source Software (OSS) as defined in [10]. The three types of OSS, and predictions on the global data usage for each, are:

1. **Exploration-Oriented**. Research software which has the goal of sharing knowledge with the masses. Such software usually consists of a single main branch of development, which is tightly controlled by a single leader. In such a project we predict to see very few global variables and, as the software evolves, a decrease if any change in global variable usage.
2. **Utility-Oriented.** This type of software is feature-rich, and often experiences rapid development, possibly with forks. In this category of software, we expect to see a relatively high reliance on global data, which will gradually increase over the software's lifetime, possibly with periods of refactoring.
3. **Service-Oriented.** Software in this category tends to be very stable and development is relatively inactive due to its large user-base and small developer group. Unlike exploration-oriented software, where a single person has complete authority, a small number of "council" members fulfill the decision-making role. For software of this classification, we predict global data usage to be higher than exploration-oriented software but less than utility-oriented software. As the software evolves, we also expect to observe a decrease in reliance upon global data.

4 Methodology

Our objective was to study both the presence and role of global data in several large-scale software systems, and therefore, it was important to devise an approach for *automatically* collecting such data. Whereas in [20], global data usage was collected for a *single* version of the Linux kernel, our focus was more extensive, as we were interested in examining *numerous* versions of multiple software projects. Consider one of the three case studies presented later in Section 5: *GNU Emacs*. In total, 15 versions of Emacs were examined (across a 14 year time period), the accumulative source code base consists of roughly 4 million lines of code. Clearly, examining the pervasiveness of global data over the evolution of such a large-scale software system requires an automated process. This section provides an overview and discussion of the design and implementation of our global data collection tool called `gv-finder`.

Initial approaches to developing the global data collection tool included hand coding a parser, and the modification of `gcc`. However, the approach decided upon was to write a stand-alone tool similar to a linker, which takes as input a collection of relocatable object files. Relocatable object files are usually produced as the output from either a compiler or assembler, and contain the machine code representation for some source code entity (e.g., a file or a concatenation of files)

along with information needed by both the linker and loader [15]. The following two observations lead us to adopt this approach:

- If a source file uses global data which happens to be instantiated within a different source file, the corresponding relocatable object file will contain a symbol table entry indicating that the global data is undefined. When the linker is invoked with the complete set of object files used for constructing the target application, it will replace any reference to the undefined global data with the address of the global data instantiated in one of the other object files.
- If a source file instantiates and exports global data, the corresponding relocatable object for that file will contain a symbol table entry declaring the data as global. The linker uses the address of a global symbol (also found in the symbol table) to resolve references to the same global data occurring in other external source files, as well as for any internal references.

Therefore, by inspection of the set of object files which constitute the final executable application, one can determine (a) the names of all global data and the corresponding module in which they are defined and (b) for each global data, the name of the modules which refer to it. In addition to satisfying all of our design criteria, this method offers the advantage of being portable across different compiler suites. This may be useful if an application only compiles with a certain version of a compiler, or a specific company's compiler — native compilers for a given platform target a common standard object file format.

Our analysis of relocatable ELF object files makes the following distinction between different types of global data:

- **External Global Data.** If one or more object files contain an undefined reference to global data, but no object file is found to provide a matching definition, we consider the global data to be *external* to the application. This occurs when the application makes use of a library which exports global state. A common example is the use of `stdout` from the `C` standard library. This is the least severe type of global data since the application itself is not responsible for the design of the libraries it depends upon.
- **Static Global Data.** If an object file contains a definition of global data which is marked as "local" then the global data is classified as *static*. This occurs in languages such as `C` and `C++` where global data is declared with the `static` keyword[7, 19]. Static global data can only be used in the file which declares the variable, and therefore can not introduce "clandestine" coupling [20] with other external modules. However, all the other disadvantages associated with using global data are applicable to static data, and therefore we feel it is important to make the distinction as static data is still potentially dangerous and undesirable.
- **True Global Data.** If an object file contains a definition of data marked as "global", the data is then classified as *true global data*. This data can be referenced in any other module without restriction, simply by referring to the data's name. This is the most dangerous type of global data since every

module which references the exported global variable becomes implicitly coupled [20].

It should be noted that this approach to global data analysis requires the target application to be compiled. This turned out to be a challenge for the very early versions of the software studied in Section 6, as language standards, system header files, and the required build tools have also evolved independently, and tend not to be backward compatible. However, for global variable analysis, it is only required that the source files compile, even if the resulting executable does not run correctly (or at all). Therefore, with a relatively small investment in time, we found that many of the older versions could be compiled by strategically adding fix-up macros, re-using configuration files across different versions and, in the worst-case scenario, simply removing offending lines of code (less than 100 lines of code were commented out in any given release).

5 Case Study

Over the course of this study we examined one example of each of the three classifications of OSS projects defined in [10]. Specifically, the Vim, Emacs and PostgreSQL projects were examined.

5.1 Vi IMproved (Vim)

The Vi IMproved (Vim) editor began as an open-source version of the popular VI editor, and has now eclipsed the popularity of the original Vi. Vim was created by Bram Moolenar, who based upon it another editor, Stevie[1]. Development of Vim centres around Moolenar, with other developers contributing mostly small features, however, the process relies upon the user community for bug reports. In terms of the classification of open-source software defined in [10], Vim is considered an example of a *utility-oriented* project.

Sixteen releases of Vim dating back to 1996 were studied (four earlier versions which target the Amiga were unanalyzable). Table 1 displays the Vim chronology of the examined releases. Most of the releases are considered minor, however, releases 5.3 and 6.0 are major, contributing at least 50 KLOC each to the system.

Table 1. Chronological data for the releases of Vim examined in this study [5, 8]

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
4.0	05/1996	43594	59966	5.5	09/1999	94247	127055
4.1	06/1996	43891	60396	5.6	01/2000	94964	128102
4.2	07/1996	44017	60600	5.7	06/2000	96225	129681
4.3	08/1996	44621	61606	5.8	05/2001	95548	128864
4.4	09/1996	44693	61751	6.0	09/2001	140182	187196
4.5	10/1996	44742	61875	6.1	03/2002	142091	189632
5.3	08/1998	79260	107876	6.2	06/2003	156700	209680
5.4	07/1999	93771	126383	6.3	06/2004	162441	217501

5.2 GNU Emacs

The Emacs editor is one of the most widely used projects developed by GNU. It was originally developed by Richard Stallman, who still remains the project maintainer. Given the development process and community that supports Emacs, we consider it to be an *exploration-oriented* project.

Our examination of Emacs consisted of fifteen releases stretching as far back as 1992. Details pertaining to the releases that we studied can be found in Table 2.

Table 2. Chronological data for the releases of Emacs examined in this study

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
18.59	10/1992	56216	74752	20.5	12/1999	105324	146655
19.25	05/1994	75412	104608	20.6	02/2000	105336	146693
19.30	11/1995	80824	112780	20.7	06/2000	105437	146849
19.34	08/1996	100514	140000	21.1	10/2001	137615	197481
20.1	09/1997	100406	140357	21.2	03/2002	137835	197814
20.2	09/1997	100408	140357	21.3	03/2003	138035	198130
20.3	08/1998	104193	145258	21.4	02/2005	138035	198130
20.4	07/1999	105170	146422				

5.3 PostgreSQL

As an example of *service-oriented* OSS, the PostgreSQL relational database system was examined. PostgreSQL is an example of a exploration-oriented (research) project that has morphed into a service-oriented project. The system was initially developed under the name POSTGRES at the University of California at Berkeley[14]. It was soon released to the public and is now under the control of the PostgreSQL Global Development Group.

Table 3. Chronological data for the releases of PostgreSQL examined in this study

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
1.02	08/1996	102965	175538	7.4	11/2003	222694	349461
6	07/1997	98062	162253	8.0.0	19/01/2005	242887	382686
7.2	02/2202	252155	276496	8.0.1	31/01/2005	242991	382865
7.3	11/2002	194822	308305				

We studied seven releases, of which three are considered major releases (1.02, 6.0 and 8.0.0). Version 1.02 (aka Postgres95) was the first version released outside of Berkeley, and incorporated a SQL frontend into the system. Although, the PostgreSQL project is composed of many programs, we limited our study to the PostgreSQL backend server. Table 3 outlines the date and size changes of the PostgreSQL server for the releases examined.

6 Experimental Results and Discussion

In this section we report and discuss the results gathered through the use of `gv-finder` on the selected open-source projects. Specifically, we examine the evolution of the projects in terms of their size (lines of code), the number of global variables referenced, their reliance upon global variables, and finally, the extent to which global data is used throughout the system.

6.1 Changes in Number of Lines of Code

Over the lifetimes of the projects that we studied, each has at least doubled in terms of their code size. Size data collected includes uncommented, non-white space source lines of code (SLOC), and total lines of code (LOC). Referring back to Tables 1, 2, and 3 we see the changes in source lines of code as well as total lines of code for Vim, Emacs and PostgreSQL, respectively.

As expected, each project shows a small increase in size over the minor releases as a result of perfective maintenance which can be attributed primarily to bug fixes. However, the large increases stems from the the major releases when new features were added to the systems.

Interestingly, the LOC decreases substantially from version 7.2 to 7.3 of PostgreSQL. Examination of the documented changes revealed that support for a specific protocol was removed. However, it is unclear if this change alone accounts for the 70 KLOC that was removed from the system.

6.2 Evolution of the Number of Global Variables

Initially it was hypothesized that the number of global variables would decrease over the lifetime of a project as the developers had more time to perform corrective maintenance and replace them with safer alternatives. However, this was not what was discovered. In fact, we found that the number of global variables present in *all* of the systems examined grew alongside the code size as demonstrated in Figs. 1, 2, and 3. In the figures, the number of distinct global variables is classified as being either true, static or external. To further clarify the figures, consider Fig. 1. Examination of Vim release 5.3 shows that the total number of global variables identified is 684. These 684 references are composed of 426 true, 238 static, and 20 external global variables.

The finding that the number of global variables increases alongside the lines of code might suggest that the use of global variables is inherent in programming large software systems (at least those programmed in `C`). This is even more interesting given that according to the classifications in [10], PostgreSQL and Emacs are developed under a stringent process that ideally would attempt to limit the introduction and use of global variables.

6.3 Evolution of the Reliance Upon Global Variables

In an attempt to evaluate how reliant the systems are upon global data, we recorded the number of lines of code that reference global data. Using this, we

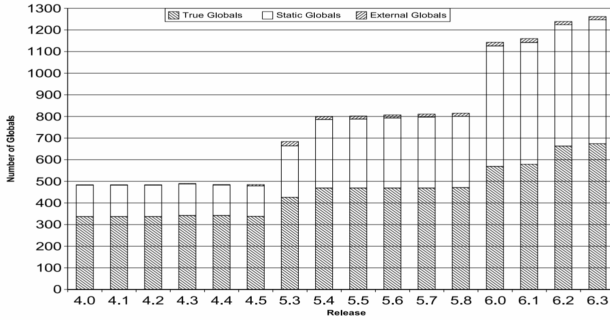

Fig. 1. The number of true, static and external globals identified by `gv-finder` in Vim

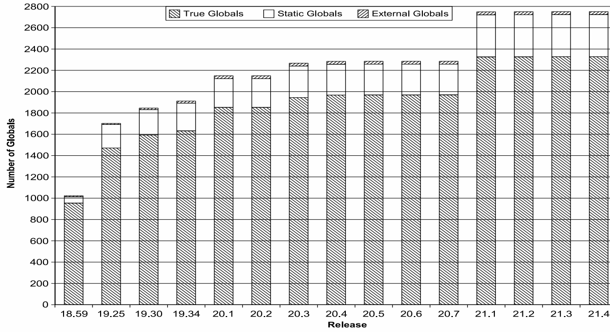

Fig. 2. The number of true, static and external globals identified by `gv-finder` in Emacs

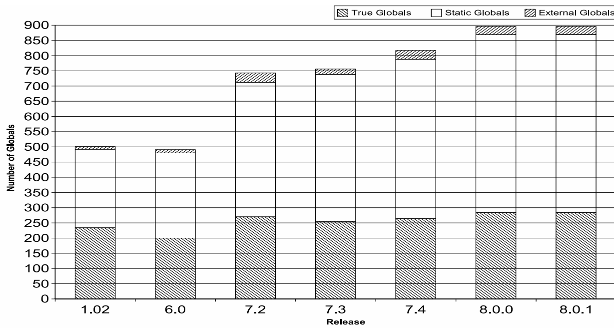

Fig. 3. The number of true, static and external globals identified by `gv-finder` in PostgreSQL

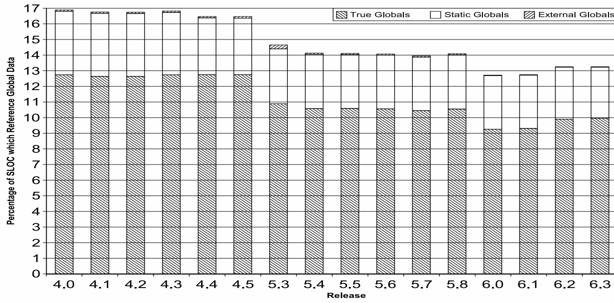

Fig. 4. The percentage of references to global data per line of source code. The percentages are classified as either true, static and external globals as identified by `gv-finder` in Vim.

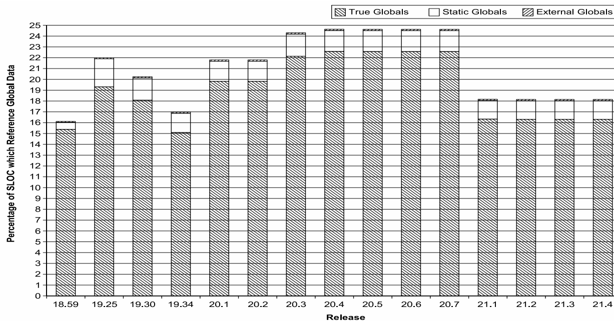

Fig. 5. The percentage of references to global data per line of source code identified by `gv-finder` in Emacs

are able to report the percentage of lines of source code (SLOC) that reference global data as displayed in Figs. 4, 5, and 6.

In this form the figures diminish the actual reliance of the projects upon global data. This can be attributed to two factors. First, each line that references a global variable is only counted once, even if it might reference multiple global variables. However, the most important factor is simply that the number of lines of code is growing much faster than the number of globals. Therefore, even though the number of global variables present in each system is growing, the use of SLOC as the divisor negates this fact.

To gain a better perspective on the reliance of global data, we plotted the number of references to global data divided by the total number of globals. Examining Figs. 7, 8, and 9, a wave pattern is observed for all projects. This might indicate that the original intuition that global variables were added to code as a quick fix in order to ship the initial release, after which their number would decrease, was simply too limited. The wave pattern that is evident in the figures could be interpreted as the iterative process of adding new features, and hence new globals, to the system and the later factoring out of them over time.

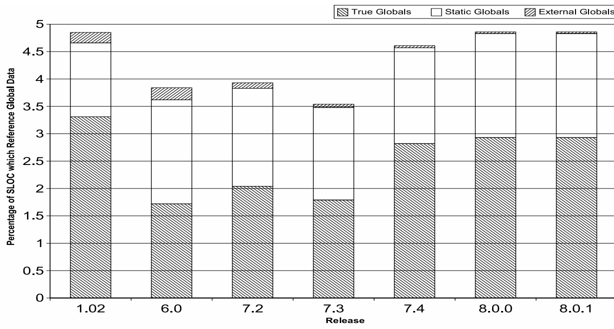

Fig. 6. The percentage of references to global data per line of source code identified by `gv-finder` in PostgreSQL

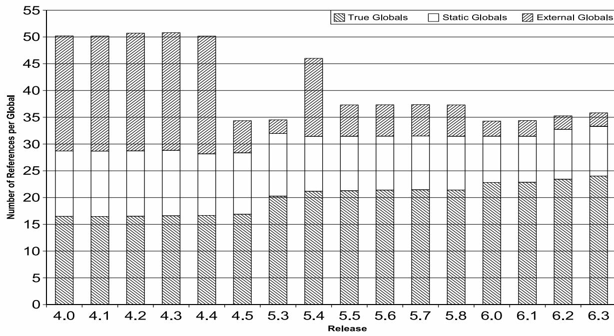

Fig. 7. The number of references per global variable discovered by `gv-finder` in Vim

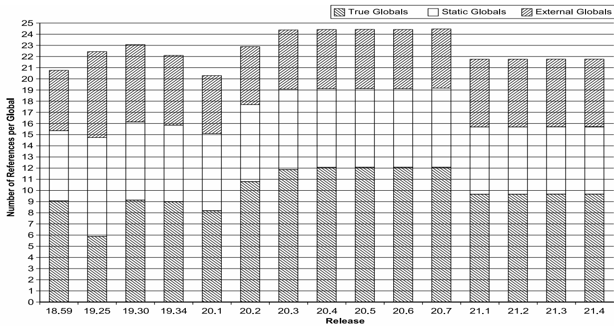

Fig. 8. The number of references per global variable discovered in Emacs

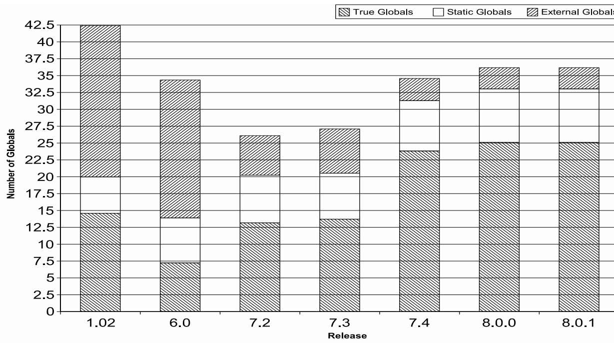

Fig. 9. The number of references per global variable discovered in PostgreSQL.

However, contrary to our intuition the reliance upon global data appears to peak at mid-releases. This may indicate that the addition of new features in major-releases are the result of clean, well-planned designs. It appears that the process of identifying bugs and patching them as quickly as possible results in the introduction of the majority of references to global data. As the frequency of bug reports curtail the developers are able to focus on refactoring the hastily coded bug fixes, thereby reducing the reliance upon globals.

6.4 Evolution of the Extent of Use of Global Variables

Finally, in order to examine how widespread the use of global variables is throughout the systems, we collected data pertaining to the number of functions which make use of global data as displayed in Figs. 10, 11, and 12. The extent of usage of global data in Vim and Emacs is considerably higher than in PostgreSQL. The percentage of functions which reference global data is greater than 80% for both of the editors, while the percentage in PostgreSQL is approximately 45%.

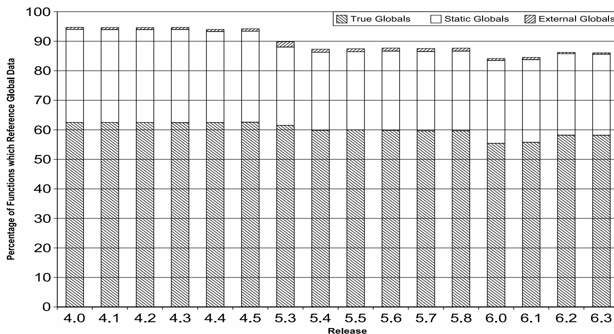

Fig. 10. The percentage of functions which reference global data. The percentages are classified as either true, static and external globals as identified by `gv-finder` in Vim.

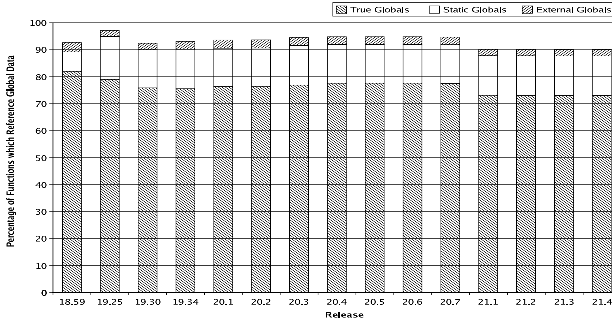

Fig. 11. The percentage of functions which reference global data as identified by `gv-finder` in Emacs

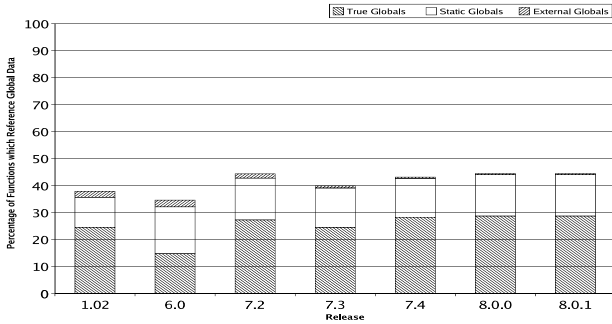

Fig. 12. The percentage of functions which reference global data as identified by `gv-finder` in PostgreSQL

We should note that there are some threats to the validity of our work. As noted earlier, we were unable to examine every single release of all three projects. The application of `gv-finder` to all releases would result in a more precise view of the evolution of global data usage across the entire lifetime of the projects. However, we believe that the examined releases provide sufficient insight into the projects in order to base our findings.

Additionally, the usage pattern of global data discovered by our work may not be visible in other types of software. Specifically, our findings are the result of the examination of open-source projects, two of which are text editors. Therefore, it is not clear if our results would hold for a wider spectrum of software (for example, closed-source projects). In order to draw any further conclusions we plan on examining a larger number of projects, ranging from compilers to multimedia players.

7 Conclusion

In this study we performed a detailed analysis of the pervasiveness of global data in three open-source projects. Our contributions are twofold. First, the categorization of a project as either service-, utility- or exploration-oriented does not appear to be indicative of the usage of global data over its lifetime. In conjunction with the fact that the number of global variables increase alongside the lines of code could indicate that the use of global data is inherent in programming large software systems and can not be entirely avoided. Second, and most interesting is the finding that the usage of global data followed a wave pattern which peaked at mid-releases for all of the systems examined. This might suggest that the addition of new features in major-releases are the result of proper software design principles while the corrective maintenance performed immediately after a major-release may result in increasing the reliance upon global data. Later phases of refactoring (preventative maintenance) appear to be able to slightly reduce this reliance.

Acknowledgments

We would like to Mark Giesbrecht, Michael Godfrey and the students of Prof. Godfrey's CS846 class at UW where this work was initially developed, Cory Kasper, and the FASE referees for their comments on this work.

References

1. Vim F.A.Q. Available at http://vimdoc.sourceforge.net/vimfaq.html.
2. L. C. Briand, J. Daly, V. Porter, and J. Wüst. A comprehensive empirical validation of design measures for object-oriented systems. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 246, Washington, DC, USA, 1998. IEEE Computer Society.
3. F. P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.
4. M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall. Mininig evolution data of a product family. In *MSR'05: Proceedings of the International Workshop on Mining Software Repositories*, May 2005.
5. S. Guckes. Vim history - release dates of user versions and developer versions. Available at http://www.vmunix.com/vim/hist.html.
6. A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
7. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
8. J. Magid. Historic linux archive. Available at http://www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Sep-29-1996/apps/editors/vi/.
9. S. McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, second edition, 2004.

10. K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85. ACM Press, 2002.
11. A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *J. Syst. Softw.*, 20(3):295–308, 1993.
12. D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287. IEEE Computer Society Press, 1994.
13. M. Pinzger, M. Fischer, and H. Gall. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3):183–196, April 2005.
14. PostgreSQL Global Development Group. *PostgreSQL 8.0.0 Documentation*, 2005.
15. L. Presser and J. R. White. Linkers and loaders. *ACM Comput. Surv.*, 4(3):149–167, 1972.
16. 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.
17. S. R. Schach and A. J. Offutt. On the nonmaintainability of open-source software position paper. *2nd Workshop on Open Source Software Engineering*, May 2002.
18. W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems J.*, 13(2):115–139, 1974.
19. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
20. L. Yu, S. R. Schach, K. Chen, and J. Offutt. Categorization of common coupling and its application to the maintainability of the linux kernel. *IEEE Trans. Software Eng.*, 30(10):694–706, 2004.
21. T. Zimmermann, P. Weissgerber, 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.