Assessing the Impact of Global Variables on Program Dependence and Dependence Clusters

David Binkley [1] Mark Harman, Youssef Hassoun, Syed Islam, and Zheng Li

King's College London, Centre for Research on Evolution, Search and Testing (CREST) Strand, London, WC2R 2LS, UK

Abstract

This paper presents results of a study of the effect of global variables on the quantity of dependence in general and on the presence of dependence clusters in particular. The paper introduces a simple transformation–based analysis algorithm for measuring the impact of globals on dependence. It reports on the application of this approach to the detailed assessment of dependence in an empirical study of 21 programs consisting of just over 50K lines of code. The technique is used to identify global variables that have a significant impact upon program dependence and to identify and characterize the ways in which global variable dependence may lead to dependence clusters. In the study, over half of the programs include such a global variable and a quarter have one that is solely responsible for a dependence cluster.

Key words: Dependence Cluster, Program Slice, Global Variable

1 Introduction

Global variables are generally deprecated in advice to programmers, with many authors arguing that they have negative effects [39,47,53]. The use of global variables has been argued to have harmful effects on many aspects of software engineering, including maintainability [54] and correctness [3]. Some programming practitioners have gone so far as to suggest, perhaps only semi–seriously, that programmers might be fired for using global variables [49]. Of course, the introduction of global variables can produce potential efficiency gains [45], but such global–introduction transformations are performed as a

[1] On sabbatical leave from Loyola College in Maryland

meaning–preserving compiler optimization, not as an approach to source-level code improvement.

Though the typical view in programming language texts (for example Stroup-trup's C++ book [46]) is that global variables may often be a source of problems, this view is not universally held. Some authors have even suggested that global variables should be used in place of local variables [24]. However, despite much debate and advice on the use of global variables over several decades, there remains little empirical study of the effects of global variables.

Program dependence, which captures the influence of one program component on another, is important because it has a bearing on so many aspects of software engineering. For example, program dependence has been linked to the ease with which a program can be understood, in work on program comprehension [2,21]. The effect of program dependence is also felt in software maintenance and re–engineering, where it delimits the changes that may be performed [25,50] and captures the impact that such changes will have [19].

Dependence analysis forms the cornerstone for many software engineering activities that rely upon program analysis, such as program comprehension [2,21], impact analysis and reduction [19,50], reuse [20], software maintenance [25], testing and debugging [11,30,41], virus detection [37], and restructuring and reverse engineering [4,36].

This paper presents a technique used to study the effects of global variables on dependence as well as results from empirical studies of these effects. The impact of even a single global can be very far–reaching; in some of the programs studied, a single global was found to account for most of the program's overall dependence connectivity.

The paper is also concerned with the effect that a global variable has on the presence or absence of large dependence clusters. A dependence cluster is a set of program statements, all of which are dependent upon one another. Recent work [9] has shown that dependence clusters are surprisingly prevalent. Therefore, one of the additional goals of the study reported herein is to explore the ways in which global variables can lead to dependence clusters.

As this paper will show, global variables can be the cause of dependence clusters. The ability to identify the causes of dependence clusters has implications for work related to these analyses. For example, in program comprehension, several authors have considered hierarchical decomposition as a navigation mechanism that manages the cognitive complexity of program dependence [2,21]. However, the presence of a dependence cluster will lead to a degenerate collapsed hierarchy, in which such decomposition will be difficult. The ability to identify causes of clusters may allow such navigation tools to either avoid

them or to treat them as special cases. Furthermore, the ability to break some clusters will improve the applicability of these approaches.

The paper makes the following primary contributions:

(1) It introduces an algorithm for variable substitution that allows the dependence due to a particular global variable to be ignored. This is used to assess and measure the effects of the global variable on dependence.
(2) The paper presents quantitative results that assess the effect on dependence of 849 global variables in 21 programs. The study reveals that more than half the programs considered have individual global variables that have a significant impact on overall program dependence.
(3) The paper presents further qualitative results of the effect these high dependence globals have on large dependence clusters. In some cases, a single global was found to be the sole cause of a cluster, establishing evidence for a link between the use of globals and the presence of large dependence clusters. The study also investigates the categories into which these effects fall and the source code constructs that cause them.
(4) Finally, the paper presents a case study in which sets of global variables collectively combine to cause dependence clusters.

The rest of the paper is organised as follows: Section 2 presents background material on dependence clusters. Section 3 introduces the algorithm for eliminating dependence due to a global variable and uses this to measure the effect of each global has on overall program dependence. Sections 4 and 5 present the five research questions addressed by the experiments and the experimental design. Sections 6 and 7 present the results of the quantitative and qualitative studies of global variable dependence effects, while Section 8 presents the case study of multi-global variable dependence effects. Sections 9 and 10 consider threats to validity and related work, and finally, Section 11 summarises the paper.

2 Dependence Clusters

A *dependence cluster* is a maximal set of program points all of which are mutually inter–dependent. Being maximal, a dependence cluster is not contained within any other dependence cluster. One complexity in identifying dependence clusters is the non-transitive nature of dependence for certain language constructs such as threads [34] and procedures [31]. This means that dependence clusters are not simply strongly connected components within a dependence graph. Fortunately, context-sensitive interprocedural dependence analysis captures the necessary calling-context information [31].

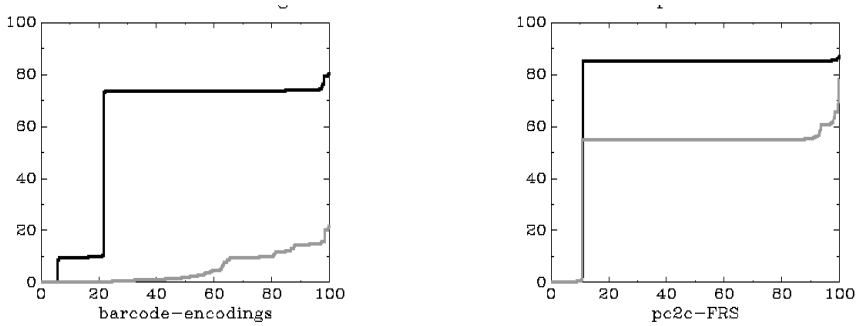


Fig. 1. Two kinds of reduced dependence. In the left chart, the dependence cluster is broken. In the right chart, the cluster size is reduced, but the cluster remains unbroken.

To visualize dependence clusters the *Monotone Slice-size Graph* (MSG) was introduced [9]. A slice is a sub-program that captures a semantically meaningful sub-computation from a program. Slices can be efficiently computed using the System Dependence Graph (SDG) [31]. An MSG is a graph of all a program's slice sizes plotted in monotonically increasing order on the x-axis. The y-axis measures slice size.

For ease of comparison the MSGs shown in this paper use the percentage of the entire program on the y-axis. Thus, an MSG plots a landscape of monotonically increasing slice sizes, in which dependence clusters correspond to sheer–drop cliff faces followed by a long flat plateau (*e.g.*, see the black line in Figure 1).

To illustrate cluster identification using an MSG, Figure 1 shows four example MSGs. As explained below, each graph includes two MSGs: an original MSG in black and a post reduction MSG in grey. For example, the MSG in black in the left chart shows (reading along the horizontal axis) that approximately the first 20% of slices are very small (containing about 10% of the program), after which the MSG reveals a sharp increase to almost 80% of the program. Most of the remaining slices are all of essentially the same size. This is visual evidence of a dependence cluster in the program. The MSG for a program devoid of clusters is shown in grey under this line.

3 Assessing the Impact of Global Variables on Dependence

The impact of global variable g is measured in terms of the area under the MSG. That is, the difference in the area for the MSG constructed from the

program with and without the dependence due to g will be deemed to denote the impact of g on overall program dependence.

Of course, reducing the quantity of overall program dependence may not mean the breaking of a dependence cluster for which MSG area reduction is a necessary, but not sufficient condition. This is illustrated in Figure 1 where the two charts show the two kinds of reduction. Both original (black) MSGs include a considerable cluster. The (lower) grey MSG in the left chart shows the result of ignoring the dependence associated with the global variable encodings; this clearly breaks the cluster. In the chart on the right the (lower) grey MSG shows only a reduction in the size of the slices making up the cluster. In this example the cluster itself is still present.

In order to measure the impact of ignoring the dependences associated with a given global variable, the approach adopted in this paper transforms the program to remove all such dependence. However, although this may be simple in principle, it turns out that the program analysis task of 'ignoring dependence' is not entirely trivial in practice.

The goal is to ignore all dependence that can be caused by definitions and references of the global variable. To achieve this, it is *insufficient* simply to mark as untraversable dependences due a given global variable. The reason for this is that such an edge-ignoring approach will not take into account the secondary effects that a global variable has. For example, upon computations that support the identification of dependence, such as dependencies due to pointers and the summary edge computation used by CodeSurfer [27] to build an SDG.

To illustrate, consider ignoring the dependence caused by global variable $g2$ in the code fragment and partial SDG shown in Figure 2. Because p may point to $g1$ or $g2$, the assignment *p = 23 does not definitely kill the assignment $g1 = 42$. However, in the absence of $g2$, *p = 23 kills the assignment $g1 = 42$ because p only points to $g1$. This obviates the need for the data dependence edge from $g1 = 42$ to $local = g1$.

As this example shows, simple edge marking is insufficient. To overcome this, the paper introduces a simple transformation to syntactically delete the global variable from the AST. Of course, such a transformation is not meaning preserving. It is not intended to be. It merely serves the purpose of ignoring all dependence due to the global variable under consideration. The resulting SDG is used to analyse the impact of the global by comparison to the original SDG (with all dependence due to the global included). In this way, the global variable's impact upon dependence can be fully assessed.

To study the impact of ignoring the dependences due to a global variable, it is pragmatic to observe two principles:

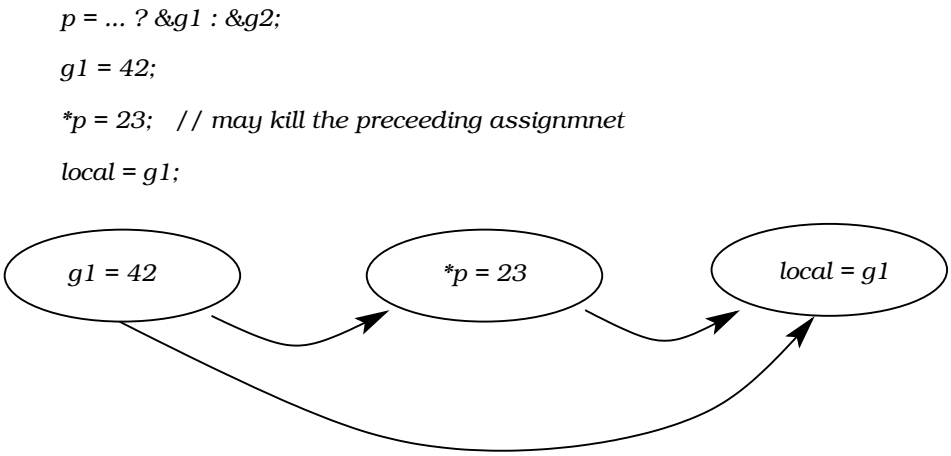


Fig. 2. Illustration were marking edges is insufficient.

(1) The size of the program should not be changed more than necessary.
(2) The effects of dependences not due to the global should remain unaffected

Taking these two concerns into account, the transformation rules replace each occurrence of a global in such a way that dependence structure is otherwise unchanged while having minimum impact on program size. For example, consider the assignment a=g+b where only the dependencies associated with global variable g is to be ignored. A minimum impact replacement, assuming that g is an int variable, would be to replace g with a constant of the appropriate type (*e.g.*, 0). While this would clearly change the meaning of the program (unless g happens to always have the value 0), the revised statement a=0+b preserves the dependence of a on b and also the size of the program (in this case right down to the character level); its sole effect is to removal the dependence of the statement on global variable g.

In the general case, any type-appropriate constant will suffice. In the formalization of the replacement, the name 'TAC' is used to denote this *type-appropriate constant*. For example, the constant ((int *)0) is used for a global variable of type int *.

To further illustrate the transformation, consider the replacement of the expression "g++". This is a two step process. The first replaces g++ with g and then second replaces g with TAC. The resulting expression removes dependencies associated with g, but does not change the number of SDG vertices used to represent the code; thus, in terms of the analysis presented in the next section, this change does not impact the program's size.

A careful analysis of the grammar for C allows program size reduction to be kept to a minimum. For example, C treats the array expression a[i] internally as the pointer expression *(a+i), which is equivalent to *(i+a) and hence i[a]. This allows occurrences of a global array g to be replaced with a constant

and thus removes the dependence associated with the global array. For example, global_array[i++] is transformed into TAC[i++] where the resulting program omits dependencies associated with global_array, but maintains the dependencies associated with i.

The formal rules for when a type-appropriate constant is suitable are given by the function *DL* (which identifies expressions that denote a named location). When this function returns bottom ($\perp$) then the use of TAC is permitted. Otherwise, the expression involving the global, must be deleted from the program. In the transformation rules, this latter case leads to the global being rewritten to the empty string, λ. In most cases, such a replacement does not impact the program's (SDG's) size. Its effect is primarily on the source represented by a given SDG vertex.

The remainder of the section formalizes the global variable dependence-elimination transformation. The algorithm, which operates on C programs, is presented as a set of transformation rules. Modification for other similar languages (*e.g.*, C++) is straightforward.

Transformation rules are written as $\dfrac{\textit{constraint set}\quad \textit{input source}}{\textit{output source}}$

and make use of the following notation

- TAC denotes a *type appropriate constant*
- λ denotes the empty string
- $[x|y]$ denotes the selection of x or y
- var(*declarator*) denotes the variable declared
  For example, var(a[]) and var(*a) are both a.
- g denotes the global targeted by the transformation

The rules transform declarations and expressions that involve g. In general, C language declarations involve storage class and type specifiers followed by a list of *declarators*. The transformation, shown in the upper left of Figure 3 removes declarators that resolve to g (*e.g.*, g, g[10], or $**g$). While syntactically valid, if this leaves an otherwise empty declaration, the tool removes the entire declaration.

For most expression occurrences of g it is sufficient to replace g with a TAC using the rule at the upper right of Figure 3. However, there are four special cases involving *lvalues* that require special treatment because an lvalue denotes a modifiable memory location [44]. Lvalues are defined by the following production from the C grammar [44]:

lvalue → identifier
→ * expression
→ primary '[' expression ']'
→ primary -> identifier
→ lvalue . identifier
→ (lvalue)

Only lvalues that denote (part of the) the location represented by g require special treatment. Those that simply involve g in denoting some other location do not. For example, no special treatment is needed for g[i], which is replaced by TAC[i] to correctly retain the use of i. As noted above, this is legal in C and thus a convenient method for achieving the minimum impact removal of the dependence induced by the global. In the four special cases, the function DL identifies those lvalues that *denote a location* directly associated with an identifier:

fun	DL	identifier	=	identifier
|	DL	* expression	=	$\perp$
|	DL	primary '[' expression ']'	=	$\perp$
|	DL	primary -> identifier	=	$\perp$
|	DL	lvalue . identifier	=	DL(lvalue)
|	DL	(lvalue)	=	DL(lvalue)

Thus, DL(a[i]++) = $\perp$ because there is no identifier associated with the incremented memory location, while DL(p.x) = p because part of the location referred to by p is modified.

The four special cases are shown as the bottom four rules in Figure 3; they occur when (some part of) g's address is taken, when it is modified by an assignment operator, and when an increment or decrement operator is involved. In the first case, NULL is used rather than TAC. For the second, the assignment portion of the assignment expression is removed. That is, "g += i++" is transformed into "i++". For the final two cases, the increment or decrement operator is simply removed.

It is interesting to note that, for occurrences of g such as *g = 2, the general rule is sufficient because g itself is not being modified. In practice this leaves an anonymous location being updated (through TAC). Such locations do not cause dependencies in the SDG, and thus the transformation retains only the effect of the right-hand side of the assignment expression. There are other similar cases when the transformation removes parts of statements that have no effect on the program's dependencies. For example, the transformation of "g++" leaves "TAC". Further examples are shown in Figure 4.

$$\frac{\{\ \mathsf{var}(declarator) = g\ \}}{\begin{array}{c} declarator \\ \hline \lambda \end{array}} \qquad \frac{\{\}}{\begin{array}{c} g \\ \hline TAC \end{array}}$$

General Cases

$$\frac{\{\ DL(lvalue) = g\ \}}{\begin{array}{c} \mathsf{\&}\ lvalue \\ \hline \mathsf{NULL} \end{array}} \qquad \frac{\{\ DL(lvalue) = g\ \}}{\begin{array}{c} lvalue\ assign\_op\ expression \\ \hline expression \end{array}}$$

$$\frac{\{\ DL(lvalue) = g\ \}}{\begin{array}{c} [\ --\ |\ ++\ ]\ lvalue \\ \hline lvalue \end{array}} \qquad \frac{\{\ DL(lvalue) = g\ \}}{\begin{array}{c} lvalue\ \ [\ --\ |\ ++\ ] \\ \hline lvalue \end{array}}$$

Special Cases

Fig. 3. Transformations for removing variable g.

Original Code	Transformed Code	Original Code	Transformed Code
int g	λ	int a,g	int a
g *= a++	a++	g.x = a+b	a+b
*g	*TAC	$g \rightarrow$ x	*(TAC).x
g[i]	TAC[i]	A[g]	A[TAC]
&g	NULL	&A[g]	&A[TAC]

Fig. 4. Transformation examples showing the replacement of g with TAC.

4 Research Questions

The empirical analysis addresses three sets of research questions, concerning the qualitative and quantitative effects of single globals and the combined effects of sets of globals (multiple globals). The study reports results on the quantitative effects that global variables have upon program dependence in general and the specific qualitative effect that high dependence globals have upon the program's large dependence clusters.

The majority of the results concern the effects of single globals because these

potentially produce the most valuable information to the software engineer. That is, if it is possible to identify a single global variable that can be deemed to be responsible for a dependence cluster, then the engineer has a chance to consider the meaning of this variable and ways in which it might be possible to account for, ameliorate or otherwise reduce the impact of the dependence cluster. Clearly, where clusters have multiple causes, such amelioration may also be possible, but it is likely that dealing with single causes will be preferable.

Research Question RQ1.1:
Overall Quantitative Effect of Single Globals
Over all programs, what proportion of dependence is due to global variables and how many globals have a significant impact (as defined in Section 5.2) to total program dependence?

Research Question RQ1.2:
Quantitative Effect of Single Globals on Each Program
For each program considered separately, how many globals have a significant impact (again using the statistical tests described in Section 5.2) on dependence and what is the magnitude of their effect? (This question make more sense knowing the results for RQ1.1.)

Research Question RQ2.1:
Qualitative Assessment of Effect of Single Global on Dependence Clusters
What effects do global variables have upon dependence clusters?

Research Question RQ2.2:
Qualitative Assessment of the Causes in Source Code
What source code patterns are found that correspond to the effects of global variables on dependence clusters?

The four research questions above concern the effects of *single* global variables. Research Question RQ3 concerns the effects of multiple globals on dependence clusters.

Research Question RQ3:
Qualitative and Quantitative Effect of Multiple Globals on Each Program
What effects can be found where multiple global variables participate collectively in causing dependence clusters?

5 Experimental Design

This section briefly describes the programs studied and the tests used to assess whether a global variable has a significant impact upon dependence.

5.1 Programs Studied and their MSGs

The 21 programs used as subjects in the study are described in Figure 5. They were taken from online repositories such as directory.fsf.org and planet-source-code.com except for space which is code created for the European Space Agency. For each program, the figure includes the name of the program, the number of slices taken when analyzing it, its size in lines of code as counted by the unix utility wc and the number of non-blank non-comment lines of code as counted by sloc [52], and finally a brief description. These programs cover a variety of application domains such as programming utilities, terminal software, booking systems, simulations, games, interpreters, and code transformation tools.

5.2 Statistical Tests Applied

The statistical tests used in this paper follow the well established approach to measuring the significance of effects adopted in work on control limits theory [42]. Two statistical tests are applied to determine those global variables that have significant impact on overall program dependence. The first is based on the notion of outliers, while the second is based on variance. A global variable is taken to have a significant impact upon program dependence if it is determined to be significant by both tests. The reason for considering both tests is to reinforce each individual test. Furthermore, tests based upon outliers are more robust against non-normality in the data distribution. Finally, note that most common statistical tests, such as a t-test, are not applicable in this case. Many such tests assume that the data come from normal population and compare population means but are not designed to test if a given value is expected to lie outside a population.

Using the outlier approach [1], a point is significant if it lies three times the interquartile range below (or above) the mean. Thus, using this approach, a global variable has a *significant impact* upon program dependence if it causes a reduction in the area under the MSG that is more than three times the

Program	Slice Count	wc	sloc	Description
address_book	1478	884	701	Diary management
barcode	5234	4192	2806	Barcode encoder
bc	4515	12113	7721	Calculator
compress	1201	1294	793	File compression
ctags	16716	15222	11387	C tag generator
ed	12261	9274	6202	Line editor
fass	1862	1155	990	8086 Assembler
file_server	1229	792	632	HTTP server
indent	5263	6697	4773	Pretty printer
interpreter	1880	1553	1185	Expression evaluator
lottery	2020	1382	1251	Lottery Player
nascar	1992	1096	722	Nascar Racing
pc2c	3507	1246	945	Convert Pascal to C
protest	1201	756	512	String matching
replace	803	565	513	Rexp replacement
space	9367	11987	6180	Array preprocessor
sudoku	545	710	420	Sudoku solver
sudoku1	584	915	376	Sudoku solver
time	877	2243	1308	Time pretty printer
wdiff	1484	3378	1986	Word by word diff
which	1414	2982	1807	File look utility
Total	79123	81636	54148	

Fig. 5. The 21 programs studied

interquartile range below the mean. In the variance approach [1], a point more than two standard deviations below (or in general above) the mean is significantly different, while a point three or more standard deviations below the mean is taken as *highly significantly different*. Thus, using this approach, a global variable has a *significant impact* upon program dependence if it causes a reduction in the area under the MSG that is two standard deviations below the mean. Furthermore, it has a *highly significant impact* if it causes a reduction in the area under the MSG that is three standard deviations below the mean.

In the data analysis the interquartile range and the standard deviation are computed using two different samples. The first includes all the data from all the programs collectively; a global variable is considered to be significant iff it causes a significant impact on the quantity of dependence, relative to the mean of all 849 variables considered in the whole study. The second approach considers each program in isolation; thus, a global variable is considered to be significant iff it causes a significant effect on dependence, relative to the mean

of all global variables in that program.

It turns out that, for all programs, the variance approach is a stricter test for significance than the outlier approach: if a result is significant according to the variance approach, then it is also significant with respect to the outlier approach (though not necessarily vice versa). In the experiment while both tests were applied and the resulting sets of global variables intersected, the result was always the same as if only the more strict variance approach was used.

6 Impact on Dependence Levels

The quantitative study is concerned with addressing Research Questions RQ1.1 and RQ1.2. It provides results of the assessment of the impact of each global variable (of 849 in total) on the dependence in the programs in which these globals reside.

6.1 RQ1.1: Overall effects of Single Global Causes

For each of the programs, the MSG was first computed using the unmodified program. Then, it was re-computed 849 times. Each re-computation ignored the dependence due to one of the global variables. Dependence effects were then assessed by comparing the area under the MSGs.

Over all globals in the study, the average area remaining after a global variable's dependences are ignored was 97.4% with a standard deviation of 8.6%. Figure 6 shows a histogram of results for the reduction due to globals. Clearly, most globals cause little reduction: 800 of the 849 global variables no more than a ten percent reduction. However, more importantly, as can be seen in the right hand detailed histogram, there do exist *some* global variables that have a considerable impact on the quantity of dependence in the programs in which they reside.

Figure 7 shows the 30 of the 849 global variables that have a significant impact, with those having a highly-significant impact shown in black. The y-axis shows the remaining dependence for each global variable shown on the x-axis. This represents 3.5% of all the global variables. Therefore, the answer to RQ1.1 is that there are a few individual global variables that have a significant effect on the quantity of program dependence, but most globals have little effect.

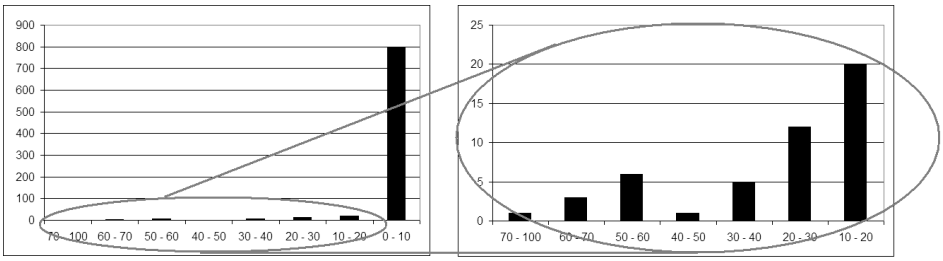


Fig. 6. Histogram showing the counts (on the y-axis) of globals leading to different ranges of dependence reduction. As the left-hand histogram shows, most global variables have little effect; they fall into the 0-10% range. The right-hand histogram zooms in on the remaining data which shows reductions from 10-100%.

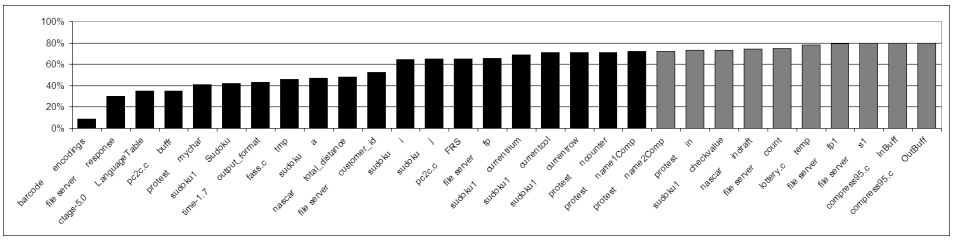


Fig. 7. Impact of ignoring dependencies for each global variable. The chart measures the *remaining dependence* on the y-axis for the 3.5% of the global variables that have a significant impact. Highlighted in black are those that have a highly significant impact.

6.2 RQ1.2: Per Program Effects of Single Global Causes

The answer to Research Question RQ1.1 suggests that there are only a few global variables that have a significant effect on dependence. A natural question to ask is whether these are concentrated in a few programs that are in some way 'special cases', or whether there are many programs that contain a global that causes a significant effect on dependence. This is the question raised by Research Question RQ1.2.

Over all 21 programs, twelve programs include at least one of the variables identified in the previous section as causing an overall significant dependence reduction. Figure 8 shows the reductions for the top four variables from each of these twelve programs (the break down of the reduction kinds are discussed in the next section). It is noticeable from this figure that the second and subsequent global variables have considerably lower impact than that of the first global variable. These results provide an intriguing answer to Research Question RQ1.2: though there may be few significant global variables overall, more than half the programs studied have only one of them.

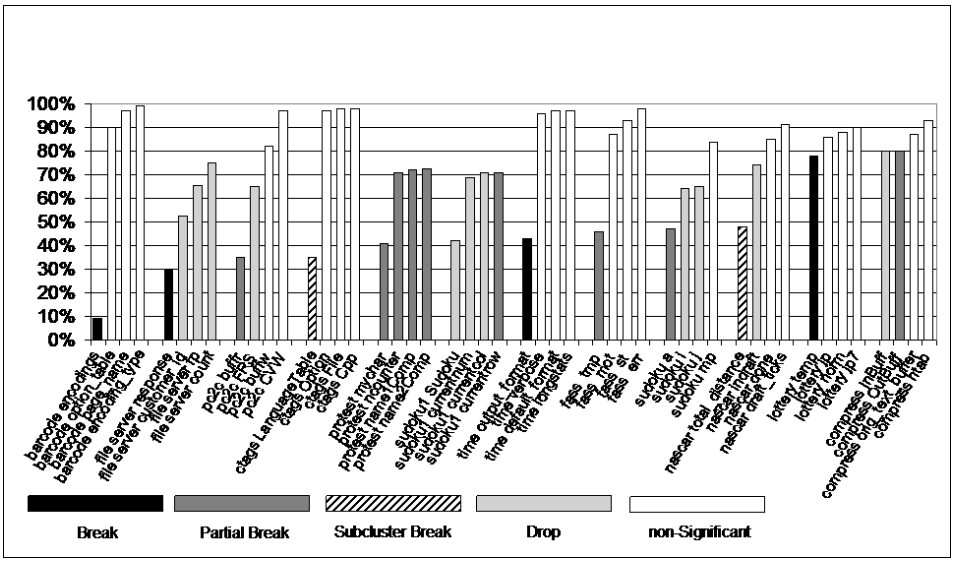


Fig. 8. Impact of ignoring dependencies for each global variable. The chart shows the four reduction causing global variables for programs where at least one significant global variable exists.

Section 5.2 describes two statistical tests that are applied to determine if a global variable causes a significant difference. These test can be applied to the entire pool of global variables (as done above) or to the globals of each programs independently. This second version asks not whether a variable, g is significant among all variables studied, but whether g is significant among only those globals that reside in the same program as g. The results serve to strengthen confidence in the answer to RQ1.2.

Figure 9 shows a comparison of the two approaches to assessing significance. Figure 10 shows the visual relationship between the two approaches to testing for significance. As can be seen from Figure 9, apart from one case (the program time), *all* those programs that contain global variables that are significant among all globals, also contain globals that are significant in the program in which they reside. This lends additional evidence to support the belief that many programs may contain globals with a significant effect on dependence.

7 Impact on Dependence Clusters

The previous section considered the quantitative impact of global variables on dependence in general. This section focuses on the specific impact of global variables on the dependence clusters of the programs studied.

			Significant Impact		
				Counts	
Program	Mean	Standard	Cutoff	Prog	All
over all	97.4%	8.6%	80.2%	26	29
ed	99.8%	0.3%	99.1%	1	0
bc	99.7%	0.5%	98.7%	0	0
which	99.7%	0.5%	98.6%	0	0
indent	99.5%	1.6%	96.4%	5	0
address	98.5%	2.4%	93.8%	1	0
wdiff	98.6%	2.5%	93.6%	0	0
interpreter	98.5%	2.7%	93.2%	1	0
lottery	96.8%	4.6%	87.5%	2	1
compress	97.4%	4.7%	87.9%	3	2
ctags	98.9%	7.7%	83.4%	1	1
protest	94.3%	11.3%	71.6%	2	5
time	96.8%	11.3%	74.3%	0	1
barcode	98.2%	11.6%	74.9%	1	1
nascar	93.7%	11.8%	70.1%	1	2
sudoku1	91.5%	12.9%	65.6%	1	5
sudoku	91.5%	13.0%	65.4%	3	3
pc2c	94.6%	14.9%	64.8%	1	2
fass	92.5%	15.8%	60.9%	1	1
file_server	89.1%	17.1%	54.9%	2	5
replace	-	-	-	0	0
space	99.8%	-	-	0	0

Fig. 9. For each program, the mean reduction leading to the number of globals causing a significant reduction (penultimate column). Each line of the table includes the mean area when ignoring the dependence of each global, the standard deviation of the mean, the cutoff for a significant impact, and the two counts. The final column is the number of significant globals using the mean over the entire collection of programs, which is shown in the first line. The program replace has no globals and the program space has only one.

7.1 RQ2.1: Overall Categorization of the Effects of Single Global Causes

From Section 2, a reduction in area under the MSG is a necessary, but not sufficient condition for the breaking of a dependence cluster. Research Question RQ2.1 asks if the reduction in area under the MSG is accompanied by a corresponding breaking of clusters. This is a more subjective determination.

To address this question the 849 MSGs were examined. Four patterns emerge, which will be denoted: *break, partial break, sub-cluster break,* and *drop*. Representative examples of the four are shown in Figure 11 where each graph shows two MSGs: the original MSG in black and the MSG after ignoring the

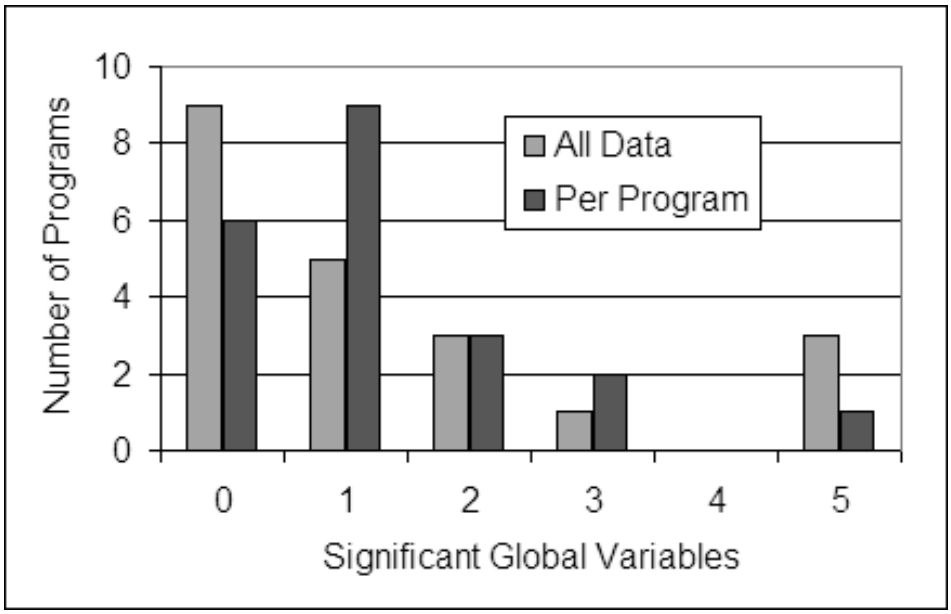
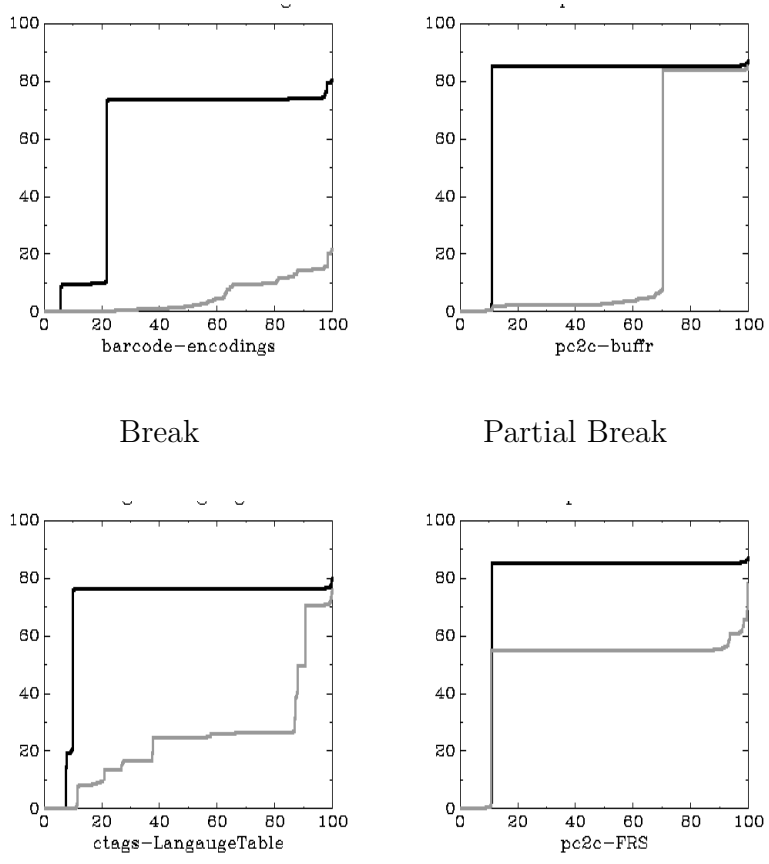


Fig. 10. Number of programs with various numbers of global variables having a significant impact.

dependence of the given global variable in grey. For example, the MSGs for barcode shown at the far left illustrate the *break* case where a large dependence cluster disappears. Next to this is a *partial break*. Here the MSG (when ignoring buffr of pc2c) shows a clear but partial breaking of the dependence clusters of pc2c. In this case approximately three fifths of the cluster disappears. Next to this, is a *sub-cluster break* where the one large cluster in the MSG for ctags fractures into a collection of smaller clusters (evidenced by the stair step pattern in the MSG). The final MSG illustrates the *drop* case where no cluster breaking occurs; the size of the slices making up the cluster are simply reduced, though the cluster remains. Thus, the characteristic cliff-face and plateau is still present but with reduced height.

As shown in Section 6 using the data over all programs, 29 global variables are significant, while using the per program data, 26 global variables are identified. Those global variables identified by the per-program data that are not by the all-program data are all *drops* of small magnitude.

Figure 12 summarizes the categorization of the effect of each significant global. Numerically, the 16 global variables significant in both approaches produced 3 breaks, 6 partial breaks, 2 sub-cluster breaks, and 5 drops. Thus, in total, ignoring the dependence associated with just over half of the significant global variables and 1.3% of all globals (11 of the 849), led to the breaking of clusters. Figure 8 includes the categorization for each program using the data over all programs.



Break Partial Break

Sub-cluster Break Drop

Fig. 11. Examples from the classification of MSGs

	Break	Partial Break	Sub-cluster Break	Drop
All Data	4	10	2	13
Per Program	3	6	2	15
Both	3	6	2	5

Fig. 12. Effect of global variables with a significant impact on dependence clusters.

Over half the global variables that are considered significant (either per program or overall) play an important role in the formation of a dependence cluster. Furthermore, as can be seen from Figure 8, all of the programs that contain a global variable that has a significant effect on dependence also contain a global variable with a non–trivial role to play in the formation of large dependence clusters.

7.2 RQ2.2: Source Code Features that Appear to Cause Clusters

To answer Research Question RQ2.2 the source code of each program with one or more significant global variable was inspected in an attempt to identify source code patterns that cause the use of globals to lead to the presence of dependence clusters. Four patterns emerge. These will be denoted: *central data structure*, *top-up*, *lazy programmer*, and *library*. Each is now defined and illustrated using case study examples from the code studied.

Examples of a *central data structure* include the *board* in a game, the *memory registers* used by interpreter, the *current instruction* processed by the assembler fass, and *current state* of the parser within the pretty printer indent. In most cases, ignoring the dependence associated with such a global variable simply causes a drop, but not a breaking of dependence clusters. This is primarily because dependencies involving other variables continue to tie the disparate parts of the cluster together.

However, in some cases, the central data structure is all that binds the cluster; ignoring its dependencies breaks it. For example, in fass, the current instruction is held in a central data structure. Ignoring this global variable's dependencies disconnects the code for processing each kind of instruction.

The second pattern, *top up*, is caused by a variable that adds an often small increment to a large collection of slices. The most common cause of this pattern is an input buffer where the reading of the input is part of most slices and gets 'cut off' from each of these slices when ignoring the input buffer's dependencies.

In four of the twelve occurrences of this pattern, the input was subsequently used in sufficient decision logic to also lead to some evidence of cluster breaking. With three of the four, this applies to only a small number of slices. The fourth is buffr from pc2c. As shown in the upper right chart of Figure 11, approximately three fifths of the large cluster is broken up by ignoring the (decision based) dependencies of buffr.

The third pattern, *lazy programmer*, occurs when a single global variable is used in place of a collection of locals. Often this pattern is obvious from the global variable's name (*e.g.*, temp or xxindex). This pattern causes needless dependence connections between functions; thereby, linking together otherwise unrelated parts of the program.

An example of this pattern is the global FRS (Function Return String) from pc2c. Its dependencies' impact on the MSG (a drop) can be seen in the lower right of Figure 11. Other examples include (loop) counters and (input) file pointers.

The final pattern, *library*, occurs in three of the programs. It is the only one that was always associated with at least partial breaking of dependence clusters in this study. This pattern is similar to the *central data structure* pattern except that the data structure is read only. Instances include
struct encoding encodings[]; from barcode,
static parserDefinition** LanguageTable; from ctags, and
char *output_format; from time.

For example, the array encodings holds a pointer to each of the kinds of barcode that the barcode program is able to encode. Similarly, ctags can produce tags for a variety of languages. The selection is achieved by indexing into the global array LangaugeTable. Finally, output_format is used by the output function of time. The format is iterated over; thus, tying together the code for all the various output formats.

Library variables cause dependence clusters primarily because static analysis tools cannot determine that the particular element chosen will not change. For example, from the static analysis point of view, it is possible for barcode to switch encoding functions in the middle of an encoding. This serves to link together the different encoders into one large cluster. The (external) insight that only one encoder is ever used for any given encoding (execution of the program) would allow a comprehension tool, for example, to break the cluster. However, such a tool would require sophisticated domain knowledge, placing this beyond the abilities of current dependence analysis technology.

8 Multiple Global Causes

This section considers the effects of ignoring the dependencies associated with combinations of globals. Several of the programs shown in Figure 8 include such sets of multiple significant global variables. Program pc2c is used as an illustrative case study. Similar patterns exist in compress, file_server, protest, and sudoku. The case study considers the two significant global variables: FRS and buffr and the 'almost significant' global variable buffw. The dependence reductions achieved by ignoring various combinations are shown on the y-axis of the chart at the top left of Figure 13.

Next to the percent reduction bar chart, the top row of Figure 13 shows pc2c's original MSG and the MSG resulting from ignoring the dependence of all three variables. The interesting thing to note in the final MSG is the complete breaking of the clusters. The second row shows the MSGs for buffr, FRS, and their combination, which is interesting because it shows more than simply the combined effect. That is, in addition to the 'break' and 'drop', it also shows evidence of a sub-cluster break. The final three charts show the

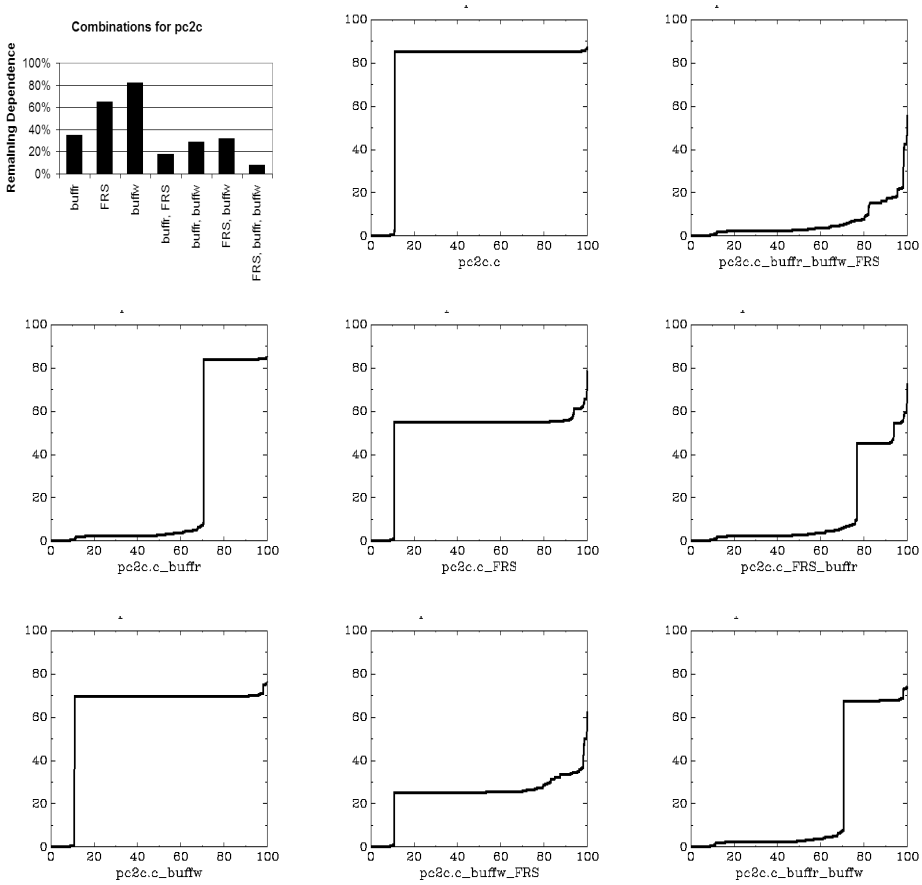


Fig. 13. MSGs for selected global variable of pc2c showing the impact of ignoring dependences associated with combinations of global variables.

MSG produced when ignoring dependence associated with buffw, buffw and FRS, and buffr and buffw.

9 Threats to Validity

This section considers threats to the external, internal, and construct validity of the results presented. The main external threat arises from the possibility that the selected programs are not representative of programs in general, with the result that the findings of the experiments do not apply to 'typical' programs. The programs studied include a wide variety of different tasks including, applications, utilities, games, and system code. There is, therefore, reasonable cause for confidence in the results obtained and the conclusions drawn from them. However, all of the programs studied were C or C++ programs. Therefore, it would be premature to infer that the results necessarily apply to other programming languages.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. In this experiment, the primary threat comes from the potential for faults in the tools used to gather the data. A mature and widely used slicing tool (CodeSurfer [27]) was used to mitigate the concern.

Construct validity assesses the degree to which the variables used in the study accurately measure the concepts they purport to measure. Note that in the presence of human judgments, construct validity is a more serious concern. In this study the only measurement is of slice size, which can be done accurately.

10 Related Work

This paper considers the role global variables play in source-level dependence in general and dependence clusters in particular. Though global variables have long been regarded as a potential causes of problems [53], there has been no previous work that has provided a quantitative assessment of their impact on dependence. Previous work on dependence clusters in software engineering [22,38,40] has focused on higher levels of abstraction, such as models or functions. Previous work on source-level dependence clusters has been primarily carried out in support of compiler analysis where semantics preservation is a key requirement [23,33,43].

Dependence analysis has been shown to be effective at reducing the computational effort required to automate the test-data generation process [29]. Using dependence analysis, it is possible to reduce both the amount of code to be tested and the size of the input domain. However, the presence of large dependence clusters will mean that no such reduction can be achieved when testing any part of the program that lies in a cluster. Identifying and busting these clusters can therefore be thought of as a step towards improving testability.

In software maintenance, dependence analysis is used to protect a software maintenance engineer against the potentially unforeseen side effects of a maintenance change. This can be achieved by measuring the impact of the proposed change [19] or by attempting to identify portions of code for which a change can be safely performed, free from side effects [25,50]. Unfortunately, all statements in a dependence cluster transitively impact all other statements in the cluster. Therefore, the ripple effect for these statements will be large and any attempt to perform a modification will be challenging.

The transformation based approach to assessment of dependence due to globals and the effects on dependence clusters is similar to Krinke's barrier slicing [35] and the 'wedge' transformation of Lakhotia and Deprez [36]. In barrier

slicing, barriers are used to prevent consideration of any dependence past the barrier. In tucking, a wedge is inserted into the code to 'cap off' dependence before the wedge so that the code may be split out and folded into smaller, ideally more cohesive, sub units.

The work reported in this paper is part of a research agenda, currently being pursued by two of the present authors (Harman and Binkley) and their colleagues and collaborators. For this agenda, dependence analysis is advocated as a way to provide tools and techniques for empirical assessment of dependence structures. The present paper is an invited submission to a special issue of JSS and it was largely through this work that the invitation to submit the present paper arose. The authors have been encouraged by the editor to include a brief overview of this previous work in this section. To achieve this, the following paragraphs set out the results so far established in this on-going 'empirical dependence analysis' research agenda.

Binkley and Harman [14] presented the first study which aimed to answer the question: 'How big is a typical program slice?'. Though slicing had been first proposed some 24 years previously by Mark Weiser [51], there had not been any subsequent attempt to systematically slice a large code base using every possible slicing criterion, thereby providing baseline data on slice size. This paper aimed to do just that. The paper constructed slices for 43 programs containing just over one million lines of code. To date, this remains the largest empirical study of slicing in the literature. It also considered the impact that calling context and structure field expansion has on slice size. In order to construct the large number of slices required, several novel slice efficiency techniques were introduced [15,18]. The paper was later extended [12] to provide a larger study that also considered the effects of dead code, pointer analysis, and slice granularity on the size of slices produced.

Subsequently, Binkley and Harman noticed that, though forward and backward slicing are dual notions of dependence, there is an interesting difference in the distribution of size of slices. That is, because of the duality of forward and backward dependence, the *average* size of a set of forward slices of procedure or program p will be identical to the average size of the backward slices of p. However the distributions of these slices are very different; the forward slices tend to be smaller. This was demonstrated empirically, where it was shown that the difference in slice–size distribution is entirely due to the effects of control dependence in structured languages [8]. This realization lends to forward slicing a hitherto unrealized importance, making it all the more surprising that forward slicing has been largely overlooked in the literature, by comparison with its much more widely–studied counterpart: backward slicing.

Though forward slices have been demonstrated to be smaller than backward slices, the sad fact remains that all static slices, forward or backward, tend to

be rather large. That is, a programmer faced with a million line program, is unlikely to be consoled by a 300,000 line slice; though the slice may be smaller than the original program, the threshold at which comprehension support becomes realistic remains some way off. In order to address this slice size problem, a new form of dependence analysis called *Key Statement Analysis* was introduced [28] and recently empirically studied [6]. In Key Statement Analysis, dependence computation is used to target those few statements upon which the program's dependence revolves. The empirical findings demonstrate that key statement analysis can be used to identify the few statements in a program that capture most of the impact of the whole program's dependence.

A separate study presented the effects of formal parameters and global variables on levels of predicate dependence [13,16]. Predicate dependence was considered a subject worthy of study because the predicates of a program capture its essential logical intention, the comprehension of which underpins so many software engineering activities. The primary finding of this work was the observation that, as the number of formal parameters available to a predicate increases, the proportion upon which it depends tends to decrease. This was noticed in many of the programs studied and it was a trend that was borne out by statistical analysis. No such trend was observed for backward slicing. This result may indicate that as functions increase the number of formal parameters available, they tend to become less cohesive.

Subsequent work [32,48], exploited this observation regarding cohesion to develop Search Based Software Engineering techniques for automating the process of slicing procedures, guided by fitness functions that capture dependence interactions.

Previous work has also considered other potential harmful effects of dependence structures that can be uncovered using static analysis. Chief among these 'dependence anti patterns' [7] are dependence clusters [9]. This work provided a definition of several forms of anti pattern and dependence–based techniques for locating them. The empirical results indicated how these techniques found examples of anti patterns in open source and production industrial code. Other work illustrated the way in which the normally static nature of these forms of dependence analysis could be brought to life in animations, that provide a 'fourth dimension' to dependence visualization [10].

Previous work has also presented empirical results on the relationship between high level program concepts (such as credit, undercarriage and holiday entitlement) and low level dependence at the statement level [5,26]. This work revealed that code which is conceptually similar also has a tighter, more cohesive dependence structure.

In 2004 Binkley and Harman provided a detailed survey of empirical results

on program slicing [17], to which the reader is referred for a more detailed account of related work and results concerning program slicing and program dependence.

11 Summary and Future Work

This paper is concerned with the effect of global variables on program dependence. It introduces a technique for measuring the effect of a global variable on the quantity of dependence present in a program and uses this to study the effects of 849 global variables from 21 programs. The results show that, while most global variables have essentially no impact on program dependence, there are a few that have a large and significant effect. Moreover, though there may be few such global variables, many programs (more than half those studied) have at least one such significant variable.

The paper also studies the way in which dependencies due to some global variables hold together large dependence clusters. The results show that globals can be the sole cause of such clusters. The paper presents a categorization of these effects and examines the source code patterns behind the clusters.

The empirical results presented in the paper are findings from a study of C and C++ programs. Future work will consider other programs and programming paradigms to assess the degree to which these results generalize. It will also investigate the opportunities for dependence cluster-breaking refactoring suggested by the finding that global variables may be the sole cause of some dependence clusters.

Future work will consider the relationships between the dependence clusters of a program, techniques for helping the programmer to break them into smaller, more manageable clusters, empirical assessment of their effects on program comprehension and other potential causes of dependence clusters.

References

[1] P. Armitage, G. Berry, Statistical methods in medical research, Macmillan, London, 1994.

[2] F. Balmas, Using dependence graphs as a support to document programs, in: 2^{nd} IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, 2002, pp. 145–154.

[3] J. Barnes, High Integrity Software: The SPARK Approach to Safety and Security, Addison Wesley, New York, NY, 2003.

[4] J. Beck, D. Eichmann, Program and interface slicing for reverse engineering, in: IEEE/ACM 15^{th} Conference on Software Engineering (ICSE'93), IEEE Computer Society Press, Los Alamitos, California, USA, 1993, pp. 509–518.

[5] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, An empirical study of the relationship between the concepts expressed in source code and dependence, Journal of Systems and SoftwareTo appear.

[6] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, Evaluating key statements analysis, in: 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08), IEEE Computer Society, Beijing, China, 2008, pp. 121–130.

[7] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, J. Wegener, Dependence anti patterns, in: 4^{th} International ERCIM Workshop on Software Evolution and Evolvability (Evol'08), L'Aquila, Italy, 2008, pp. 25–34.

[8] D. Binkley, M. Harman, Forward slices are smaller than backward slices, in: 5^{th} IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, 2005, pp. 15–24.

[9] D. Binkley, M. Harman, Locating dependence clusters and dependence pollution, in: 21^{st} IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, 2005, pp. 177–186.

[10] D. Binkley, M. Harman, J. Krinke, Animated visualisation of static analysis: Characterising, explaining and exploiting the approximate nature of static analysis, in: 6^{th} International Workshop on Source Code Analysis and Manipulation (SCAM 06), Philadelphia, Pennsylvania, USA, 2006, pp. 43–52.

[11] D. W. Binkley, Semantics guided regression test cost reduction, IEEE Transactions on Software Engineering 23 (8) (1997) 498–516.

[12] D. W. Binkley, N. Gold, M. Harman, An empirical study of static program slice size, ACM Transactions on Software Engineering and Methodology 16 (2) (2007) 1–32.

[13] D. W. Binkley, M. Harman, An empirical study of predicate dependence levels and trends, in: 25^{th} IEEE International Conference and Software Engineering (ICSE 2003), IEEE Computer Society Press, Los Alamitos, California, USA, 2003, pp. 330–339.

[14] D. W. Binkley, M. Harman, A large-scale empirical study of forward and backward static slice size and context sensitivity, in: IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, 2003, pp. 44–53.

[15] D. W. Binkley, M. Harman, Results from a large–scale study of performance optimization techniques for source code analyses based on graph reachability algorithms, in: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), IEEE Computer Society Press, Los Alamitos, California, USA, 2003, pp. 203–212.

[16] D. W. Binkley, M. Harman, Analysis and visualization of predicate dependence on formal parameters and global variables, IEEE Transactions on Software Engineering 30 (11) (2004) 715–735.

[17] D. W. Binkley, M. Harman, A survey of empirical results on program slicing, Advances in Computers 62 (2004) 105–178.

[18] D. W. Binkley, M. Harman, J. Krinke, Empirical study of optimization techniques for massive slicing, ACM Transactions on Programming Languages and Systems 30 (2007) 3:1–3:33.

[19] S. E. Black, Computing ripple effect for software maintenance, Journal of Software Maintenance and Evolution: Research and Practice 13 (2001) 263–279.

[20] A. Cimitile, A. De Lucia, M. Munro, A specification driven slicing process for identifying reusable functions, Software Maintenance: Research and Practice 8 (1996) 145–178.

[21] Y. Deng, S. Kothari, Y. Namara, Program slice browser, in: 9^{th} IEEE International Workshop on Program Comprenhesion, IEEE Computer Society Press, Los Alamitos, California, USA, 2001, pp. 50–59.

[22] T. Eisenbarth, R. Koschke, D. Simon, Locating features in source code, IEEE Transactions on Software Engineering 29 (3), special issue on ICSM 2001.

[23] C. N. Fischer, R. J. LeBlanc, Crafting a Compiler, Benjamin/Cummings Series in Computer Science, Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[24] D. L. Fisher, Global variables versus local variables, Software – Practice and Experience 13 (5) (1983) 467–469.

[25] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (8) (1991) 751–761.

[26] N. Gold, M. Harman, Z. Li, K. Mahdavi, An empirical study of executable concept slice size, in: 13^{th} Working Conference on Reverse Engineering (WCRE 06), Benevento, Italy, 2006, pp. 103–114.

[27] Grammatech Inc., The codesurfer slicing system (2002).
URL `www.grammatech.com`

[28] M. Harman, N. Gold, R. M. Hierons, D. W. Binkley, Code extraction algorithms which unify slicing and concept assignment, in: IEEE Working Conference on Reverse Engineering (WCRE 2002), IEEE Computer Society Press, Los Alamitos, California, USA, 2002, pp. 11 – 21.

[29] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, J. Wegener, The impact of input domain reduction on search-based test data generation, in: ACM Symposium on the Foundations of Software Engineering (FSE '07), Association for Computer Machinery, Dubrovnik, Croatia, 2007, pp. 155–164.

[30] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.

[31] S. Horwitz, T. Reps, D. W. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–61.

[32] T. Jiang, M. Harman, Y. Hassoun, Analysis of procedure splitability, in: 15^{th} Working Conference on Reverse Engineering (WCRE'08), Antwerp, Belgium, 2008, pp. 247–256.

[33] N. Jones, S. Muchnick (eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[34] J. Krinke, Static slicing of threaded programs, in: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), 1998, pp. 35–42.

[35] J. Krinke, Barrier slicing and chopping, in: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), IEEE Computer Society Press, Los Alamitos, California, USA, 2003, pp. 81–87.

[36] A. Lakhotia, J.-C. Deprez, Restructuring programs by tucking statements into functions, Information and Software Technology Special Issue on Program Slicing 40 (11 and 12) (1998) 677–689.

[37] A. Lakhotia, P. Singh, Challenges in getting formal with viruses, Virus Bulletin.

[38] K. Mahdavi, M. Harman, R. M. Hierons, A multiple hill climbing approach to software module clustering, in: IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, 2003, pp. 315–324.

[39] L. F. Marshall, J. Webber, Gotos considered harmful and other programmers taboos, in: A. Blackwell, E. Bilotta (eds.), 12^{th} Psychology of Programmers Interest Group Annual Workshop (PPIG 12), 2000, pp. 171–180.

[40] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Transactions on Software Engineering 32 (3) (2006) 193–208.

[41] A. Podgurski, L. Clarke, A formal model of program dependences and its implications for software testing, debugging, and maintenance, IEEE Transactions on Software Engineering 16 (9) (1990) 965–79.

[42] D. Reid, N. R. Sanders, Operations Management: An Integrated Approach, 3rd ed., Wiley, 2007.

[43] T. W. Reps, Solving demand versions of interprocedural analysis problems, in: P. Fritzon (ed.), Compiler Construction, 5th International Conference, vol. 786 of Lecture Notes in Computer Science, Springer, Edinburgh, U.K., 1994, pp. 389–403.

[44] D. Ritchie, The c reference manual (1975).
URL `cm.bell-labs.com/cm/cs/who/dmr/cman.pdf`

[45] P. Sestoft, Replacing function parameters by global variables, in: Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London, IFIP and ACM, ACM Press and Addison-Wesley, 1989, pp. 39–53.

[46] B. Stroustrup, The C++ Programming Language, 3rd ed., Addison-Wesley, 2000.

[47] R. E. Sward, A. T. Chamillard, Re-engineering global variables in Ada, ACM SIGADA Ada Letters 24 (4) (2004) 29–34.

[48] J. Tao, N. Gold, M. Harman, Z. Li, Locating dependence structures using search based slicing, Information and Software TechnologyTo appear.

[49] P. Tennberg, Refactoring global objects in multithreaded applications, C/C++ Users Journal 20 (5) (2002) 20–24.

[50] P. Tonella, Using a concept lattice of decomposition slices for program understanding and impact analysis, IEEE Transactions on Software Engineering 29 (6) (2003) 495–509.

[51] M. Weiser, Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method, Ph.D. thesis, University of Michigan, Ann Arbor, MI (1979).

[52] D. A. Wheeler, SLOC count user's guide, http://www.dwheeler.com/sloccount/sloccount.html (2005).

[53] W. Wulf, M. Shaw, Global variables considered harmful, ACM SIGPLAN Notices 8 (2) (1973) 28–34.

[54] L. Yu, S. R. Schach, K. Chen, A. J. Offutt, Categorization of common coupling and its application to the maintainability of the linux kernel, IEEE Transactions on Software Eng 30 (10) (2004) 694–706.