An Empirical Study of Computation Equivalence as Determined by Decomposition Slice Equivalence

Keith Gallagher & David Binkley
Computer Science Department
Loyola College in Maryland
4501 N. Charles St. Baltimore, MD. 21210 USA
{kbg|binkley}@cs.loyola.edu

Abstract

In order to further understand and assess decomposition slicing we characterize and evaluate the size of reductions obtained by computing equivalent decomposition slices from the perspective of the comprehender, maintainer, tester and researcher. The analysis was performed on 68 **C** *language systems of sizes 100 to 50,000 lines. All decomposition slices were computed and compared for simple equality. From this data, we were able to determine with 95% confidence that the true mean percentage of equivalent decomposition slices is between 50.0% and 60.3%, with a* p-value $<$ *0.005.*

This has clear and significant impact for software testing, as any coverage method used for one of the variables used in an equivalence will apply to all variables in the class; for software comprehension as the number of items (variables) used for the understander is substantially reduced; for the software maintenance, as the number computational relationships is reduced; and for the researcher, in attempting to ascertain the underlying cause of this phenomena.

1. Introduction

Consider two of standard problems that confront a software engineer in an comprehension or maintenance task "How is the computation embodied in a variable that I am looking at related to the other computations (variables) of the program? And what will happen if I change it?" The solution to this class of problems is one of the myriad applications of program slicing.

Now suppose that the engineer examines another computation (variable) and poses the same questions, and gets the same answer by using program slicing. That is, the two program slices obtained by looking at evidently different computations are exactly the same. What is the underlying cause?

Two possible solutions arise. First, the computations are related by textual proximity and a straightforward comprehension task. For example, a couple of different computations are captured in one `for` statement.

In a more complicated scenario, the two computations have been combined by the original engineer in an unexpected way that is crucial to the solution at hand. Thus, there is conceptual relationship between the two evidently different computations such that one cannot be considered without reference to the other. This insight is important to the engineer. For now, changes to one subpart of the computation are known to effect the whole. When one computation is considered, an entire set of equivalent computations must also be considered.

Does this information assist the software engineer? Before we address that question, we must first determine if such information is worth the effort to compute, organize and present. First, we examine the size and percentage of such reductions. Using equivalent program slices as the reducing technique, we are able to show statistically significant reductions of 50%-60% in a collection of programs obtained from the net and from industry. Thus, we have a significant reduction in the *number of distinct slices* that a comprehender must tackle.

While program slicing is a technique for *reducing* the amount of information presented to a software engineer, in this instance, we use it to *combine* slice-equivalent computations. In light of our results, this means that a maintainer/comprehender need only consider approximately half of a program's computations to form a conceptual model.

1.1 Approach and Organization

Using the approach presented in the seminal paper of Basili, Selby, and Hutchens [1] (hereafter referred to as **BSH**), for experimentation in software engineering, we present a definition, plan, operation, and interpretation of the analysis. This paper is organized into 5 sections. Section 2 provides background and problem motivation; Section 3 describes the experimentation definition and plan of the empirical process; Section 4 presents the operation and interpretation of the analysis; and Section 5 concludes.

2 Background

A program slice, $\mathbf{SLICE}_{(v,n)}(p)$, of program p on variable, or set of variables, v, at statement n yields the portions of the program that contributed to the value of v just before statement n is executed [12]. The pair (v, n) is called a *slicing criterion*. Surveys of program slicing may be found in [2, 4, 11].

A decomposition slice [7] does not depend on statement numbers. It is the union of a collection of slices, which is still a program slice [12]. A decomposition slice captures all relevant computations involving a given variable and is defined as follows:

Definition 1 (Decomposition Slice) $\mathcal{DS}(v,p)$
Let
1. $Out(p, v)$ *be the set of statements in program* p *that output variable* v*,*
2. $last$ *be the last statement of* p*,*
3. $N = Out(p, v) \cup \{last\}$*.*

The statements in $\mathcal{DS}(v,p) = \bigcup_{n \in N} \mathbf{SLICE}_{(v,n)}(p)$ *form the decomposition slice on* v*.*

We take the decomposition slice for each variable in the program and form a graph,[1] using the partial ordering induced by proper subset inclusion. The graph of decomposition slices was originally intended to give software maintainers a method for visual impact analysis [6, 8]. The decomposition slice graph provides information to a software engineer about dependences that exist between variables in a system. It shows, for all variables, which decomposition slices are included in the decomposition slices of other variables. This information is useful to a software engineer in trying to gain an understanding of a system as it can be used to track the data-flow for a variable, and it can be used to identify the data that impacts on a particular variable (those variables on which it depends) and the impact of a variable (those variables which depend on it).

[1] The term "lattice" was used in [7].

2.1 Motivating Example

We produced the decomposition slice graph of a differencing program shown in Figure 1. It has 95 nodes and 364 edges. Every variable or programmer defined constant (`enum` or `typedef` value) generates a slice, as do unused global variables included in library header files. This caused a "fan out" at the bottom of the graph. These decomposition slices do not have any executable statements. There were 29 such "empty" slices. Removing them and the incident edges lowers the count to 66 nodes and 161 edges. This graph is not shown.

To further reduce the visual clutter and make the graph more readable for the software comprehender, we output only one node for each equivalent decomposition slice, using simple set equality. That is, those variables with identical statements constituting their decomposition slices were represented by one node. The reduced graph, shown in Figure 2, has 34 nodes and 43 edges.

Due to the vagaries of the layout algorithm, the reduced graph is rotated about the vertical axis with respect to the graph of Figure 1. The five upper leftmost nodes of Figure 1 are the five upper rightmost nodes of Figure 2. The three nodes to the upper right of Figure 1 are collapsed to the single node in the upper left of Figure 2. Following the edges from these nodes downward in Figure 1 leads to the "fan-out" in the lower center of the figure. This fan-out of 14 nodes is reduced to two nodes in Figure 2; and the node reduction induces a drastic reduction in the number of edges.

So in this small example we reduce a graph of 95 nodes and 364 edges to one of 34 nodes and 43 edges, a reduction of 62% in the node count by merely noting that some slices are the same. This, and other initial observations, was reported in another venue [5]. In this work, we attempt to ascertain whether or not this phenomena is a property of most software systems, or just an artifact of the examples.

3 Evaluation Definition and Plan

Following BSH, we define our experiment with six elements: motivation, object, purpose, perspective, domain, and scope. Table 1 summarizes: [To] understand and assess [the] reductions obtained by computing equivalent decomposition slices, [we] characterize and evaluate [for the] comprehender, maintainer, tester and researcher 68 systems [in the] **C** language.

For experiment planning BSH recommends:

> The design of an experiment couples the study scope with analytical methods and in-

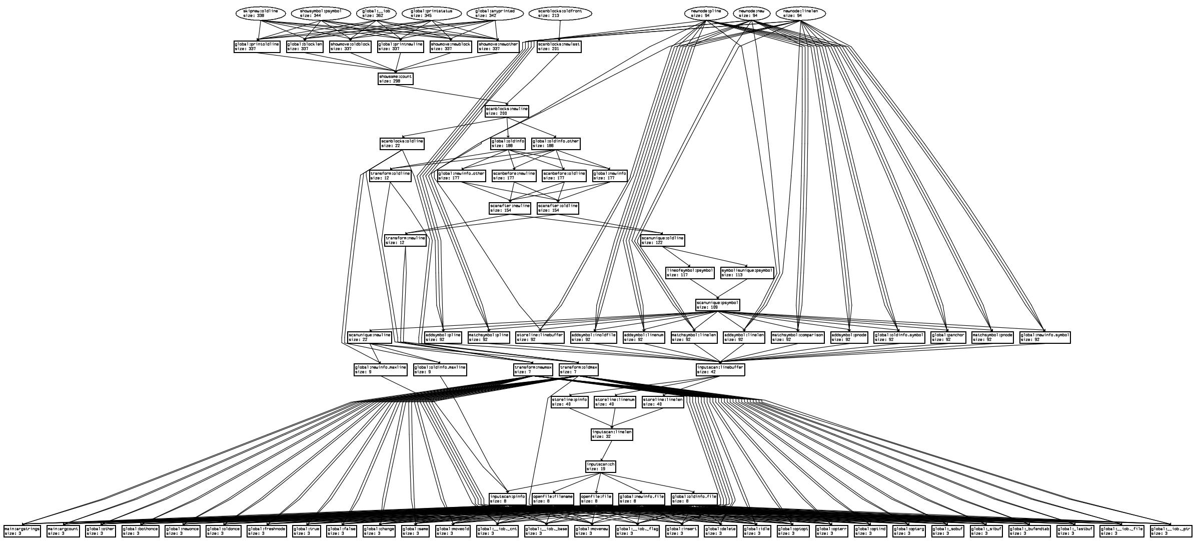

Figure 1. The decomposition slice graph of a differencing program.

Definition Element	*Activity*
Motivation	Understand and Assess
Object	Reductions Obtained by Computing Equivalent Decomposition Slices
Purpose	Characterize and Evaluate
Perspective	Comprehender, Maintainer, Tester and Researcher
Scope	61 Systems
Domain	**C** language

Table 1. Framework Definition

> dicates the domain samples to be examined. ... Different [experimental activities] require the examination of different criteria. ... The concrete manifestations of the [experimental] aspects examined are captured through measurement.

We are attempting to ascertain whether or not reduction by equivalent decomposition slices is a statistically significant property of **C** software systems. Once we have obtained the data, we will do a simple regression analysis to determine if significance exists. We will also need to verify that the sample population is normal. To do this, we selected a number of systems from the GNU/Linux utility set: file search utilities; calculators; editors; terminal management; language processors; debuggers; internet utilities; databases; and games. We are searching for the existence of a phenomena. Thus, our examination criteria will be indirect: statistical significance. The measurement will be quite simple: the number of decomposition slices and the number of equivalent ones.

4 Operation and Interpretation

For experiment operation and interpretation BHS recommends:

> The operation of the experiment consists of 1) preparation, 2) execution, and 3) analysis. ... The interpretation of the experiment consists of 1) interpretation context, 2) extrapolation, and 3) impact.

In this instance, the preparation included a pilot study [5]. Seven programs of size 350-4500 lines of code were analyzed using Unravel[10]. The results were promising: computing equivalent decomposition slices reduced the number of different slices by 51-81%. The sample and results, shown in Table 2 was not large enough, nor were the systems large enough to infer statistical significance.

For the collection of the data for this study, both Unravel and CodeSurfer[3] were used. The Unravel suite has a preprocessor which can be used to calculate all slices; a graph drawing back-end gives the full and reduced data. For CodeSurfer, the all slices are saved to a file; the same back-end computes the equivalences. It turns out that we did not need the graph to be drawn for the analysis; we just needed the node and equivalent node counts. This shortened the computation time considerably, as a computationally intensive

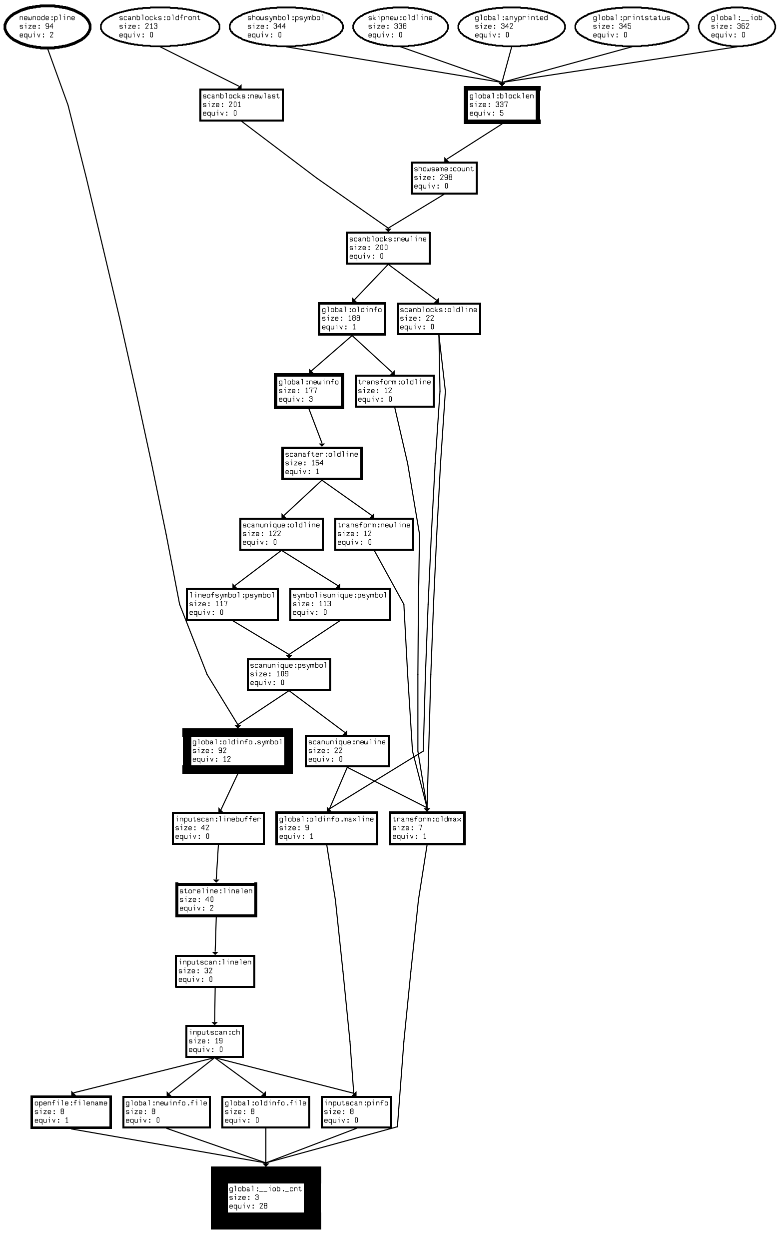

Figure 2. The reduced decomposition slice graph of Figure 1.

System	Original Slice Count	Reduced Slice Count	Reduced Percentage
dif.c	95	34	62%
lattice.c	168	83	51%
unravel.c	482	129	73%
analyzer.c	817	198	76%
parser.c	788	198	75%
P1.c	344	76	78%
P2.c	315	61	81%

Table 2. Pilot study reduction by equivalent slices

edge removal function could was avoided. Table3 show selected results.

The analysis was also straightforward; we placed the data in a spreadsheet and used the least-squares fit function over the pairs and produced some interesting results. A strong correlation exists between the number of reduced nodes and the number of original nodes, giving us an R^2 value of 70.9%. As this R^2 value is above 0.65, it indicates a strong relation. We test the null hypothesis, $H_0 : \beta = 0$, to determine if the number of reduced nodes is independent of the number of original nodes using a t-test. We receive a p-value < 0.005, therefore we reject the null hypothesis and conclude that the two variables are dependent. This conclusion, plus the high R^2 value, indicates that, by knowing the number of nodes in the original graph, we can predict with confidence the number of nodes in the corresponding reduced graph.

We would also like to characterize the percentage of node reduction over the total population of decomposition slice graphs. To do so, we first perform a Kolmogorov-Smirnov (KS) test to ensure the population is normal. The KS-test tries to determine if two datasets differ significantly and has the advantage of making no assumption about the distribution of data. Performing this test on the sample population, we receive a significance value of 0.189. Since this value is greater than 0.005, we can conclude that the sample population does not deviate from a normal population. We now conduct our t-test. From this data, we were able to determine with 95% confidence that the true mean percentage of node reduction is between 50.0% and 60.3%.

4.1 Interpretation and Application

The purpose of this study were two-fold: an evaluation of existing systems to determine if reduction by equivalent decomposition slices was more than a mere

System	Original Slice Count	Reduced Slice Count	Reduced Percentage
replace	936	622	33%
which	1230	863	29%
time-1.7	1093	910	16%
compress	1234	823	33%
wdiff	2852	1812	36%
termutils	3275	2315	29%
barcode	4419	2122	51%
indent	9430	3642	61%
bc	6494	3259	49%
copia	4705	2548	45%
gcc.cpp	8938	3942	55%
acct	9358	5827	37%
byacc	14185	7654	46%
gnubg-0.0	10119	5727	43%
flex2-4-7	15214	7984	47%
findutils	15710	7684	51%
ed	28560	7518	73%
EPWIC-1	13659	9944	27%
userv-0.95	15495	7361	52%
space	11338	10142	10%
flex2-5-4	20706	9764	52%
tile-forth	17579	5416	69%
prepro	11759	10602	9%
oracolo2	11829	10635	10%
diffutils	19513	10575	45%
gnuchess	21531	6906	67%
cadp	19488	12372	36%
ctags	48210	9100	81%
wpst	38997	14100	63%
ijpeg	29432	9224	68%
ftpd	30707	17308	43%
espresso	45968	16670	63%
go	41651	5118	87%

Table 3. Selected Data

artifact of a few sample systems; and characterization of these systems to determine the sized of such reductions. Both were successful: the reduction is more than an artifact and it is statistically significant.

There are a number of interesting applications of these observations. The first is that any test-coverage method used for one of the variables in an equivalence class will apply to all variables in the class. One of the problems faced by a tester attempting any coverage technique is that of redundant tests, those which do not increase coverage. If the tester attempts to increase coverage by targeting a variable that is in the equivalence class, no further coverage will be obtained. On the other hand, a coverage tester can get many variables covered through one test.

Second, for software comprehenders the *size* of the reduction hints at the difficulty of the task ahead. For instance, in Table 3, the `go` system (on the last line of the table) reduces by 87%. While there are still approximately 5000 slices to comprehend this is an improvement over the 41,000 originally obtained. Moreover, this analysis does not show the *relationships* that would be uncovered by a visual examination of the graph, which could further aid the comprehender. On the other side of this coin is the system `prepro` (11 lines from the bottom of Table 3) which has only a 9% reduction. This hints at difficulty of the task. It would seem that the graph is the only hope of finding meaningful relationships between the computations.

Third, software maintenance effort is reduced as the number of computations is reduced. One way to view the general idea of slicing is the attempt to find all the "pieces of the puzzle." This is why program slicing is a powerful maintenance tool. The equivalent decomposition slices give a straightforward abstraction mechanism that can be used in all phases of evolution. Alternatively, a small percentage reduction may be an indicator of difficulties ahead.

Fourth, this reduction poses interesting questions for the researcher, in attempting to ascertain the underlying cause(s) of this phenomena. The size and significance of the reductions may argue that we are currently programming with the wrong idioms. For instance, a simple *swap* operation need not evidence an intermediate variable. Linger, et al. [9] suggested a simple `a, b = b, a;` for a variable swap in 1979. We need simpler and more direct ways to embody computational definitions, that is not achievable with today's technologies. However, this is not an argument for object orientation. While objects give analysis and design insight, they are still *written* much like **C**. And slicing objects is not easy. This approach is also too fine-grained for today's gigantic, complicated and critical systems.[2] Perhaps these new idioms, whatever they are, will not even be sliceable.

While performing the program analysis, we discovered another apparently significant method to combine equivalent slices. This came about when examining some of the intermediate data. We first output all slices. Before we combined the slices into decomposition slices, we did the same equivalence reduction on the entire set that we did after we formed the decomposition slices. It appeared that we were getting significant reductions here also, but the data has not been carefully analyzed. Thus, it seems that combining by equivalent slices (not just decomposition slices) may be of interest. This discovery merits further investigation. A simple application of this would to provide the the engineer (maintainer, tester, comprehender, reuser, reengineer, etc.) with all the data (all the slices) to be formed into equivalence classes according to the task at hand, in an interactive process.

5. Conclusion

In the current computational milieu, many variables and objects are "glued" together to form computational entities. These entities create new programming idioms by the unique and clever ways in which they are assembled. It is precisely this activity that creates a system. When these entities are assembled, they become intertwined in ways that are eminently suited to discovery by program slicing. It is these coherent entities that of primary interest to the engineer. One way to reduce the information overload that the engineer sees is to use program slicing to combine and highlight the inter-connected entities. We have shown that combined decomposition slices, under simple equivalence gives a large and statistically significant reduction.

References

[1] V. Basili, R. Selby, and D. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12(7):733–743, July 1986.

[2] D. Binkley and K. Gallagher. A survey of program slicing. In M. Zelkowitz, editor, *Advances in Computers*. Academic Press, 1996.

[3] CodeSurfer. GrammaTech, Inc. http://www.grammatech.com/products/codesurfer.

[4] A. DeLucia. Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001.

[2] We have no suggestion as to what these new idioms might be. [AUTHORS' (kg) note: A lively discussion would ensue!]

[5] K. Gallagher and L. O'Brien. Analyzing programs via decomposition slicing. In *Proceedings of International Workshop on Empirical Studies of Software Maintenance, WESS*, 2001.

[6] K. B. Gallagher. Visual impact analysis. In *Proceedings of the Conference on Software Maintenance - 1996*, 1996.

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

[8] K. B. Gallagher and L. O'Brien. Reducing visualization complexity using decomposition slices. In *Proceedings of the 1997 Software Visualization Workshop, SoftVis97*, number ISBN 0725806303, Dec 1997.

[9] R. Linger, H. Mills, and B. Witt. *Structured Programming: Theory and Practice.* Addison-Wesley, Reading, Massachusetts, 1979.

[10] J. Lyle, D. Wallace, J. Graham, K. Gallagher, J. Poole, and D. Binkley. *A CASE tool to evaluate functional diversity in high integrity software.* U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995. http://hissa.ncsl.nist.gov/~ jimmy/unravel.html.

[11] F. Tip. A survey of programming slicing techniques. *Journal Of Programming Languages*, 13(3):121–189, 1995.

[12] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.