Analysis of Procedure Splitability

Tao Jiang
King's College
Strand, London
WC2R 2LS, UK.

Mark Harman
King's College
Strand, London
WC2R 2LS, UK.

Youssef Hassoun
King's College
Strand, London
WC2R 2LS, UK.

Keywords: Procedure Splitting, Program Analysis

Abstract

As software evolves there is a tendency for size to increase and structure to degrade, leading to problems for ongoing maintenance and reverse engineering. This paper introduces a greedy dependence-based procedure splitting algorithm that provides automated support for analysis and intervention where procedures show signs of poor structure and over large size. The paper reports on the algorithms, implementation and empirical evaluation of procedure splitability. The study reveals a surprising prevalence of splitable procedures and a strong correlation between procedure size and splitability.

1 Introduction

There is evidence that as programs evolve their structure deteriorates [12, 13]. One way that this program degradation manifests itself is that programs become 'bloated'. That is, systems have a tendency to accrue additional functionality and the code that goes with it. In this paper we explore this question of bloat at the procedure level. The primary question that the paper addresses is: is there a link between procedure size and splitability?

While there has been a lot of work on assessing procedure structure [1, 15] and some work on extracting a single subprocedure [11], this question has not previously been addressed. Clearly, more splitable procedures are candidates for re-engineering and so a positive answer to the question, though perhaps a further 'cloud' over the structural integrity of large-procedure systems, may contain a 'silver lining' of good news for the reverse and re-engineering community. That is, for the re-engineering community a positive answer to the paper's central question would suggest that re-engineering becomes increasingly effective as procedure size increases.

Previous work on this problem has focussed on cohesion improvement and procedure extraction. For example, tucking [11] extracts a part of a procedure to make a new subprocedure with the aim of improving cohesion, while procedure extraction, extracts marked code into a procedure, potentially improving the procedural abstraction of un-modularised code blocks [8, 10].

This paper adopts a similar approach to tucking, except that a procedure may be split into several subprocedures, rather than merely splitting in two, which tucking does. Our approach also uses a greedy optimization algorithm to find the optimal or near optimal points for a split, whereas tucking simply follows a set of pre-determined rules to locate split (or wedge) points.

Following an approach inspired by Beiman and Ott's slice based measurement of cohesion [1], we define measurements of splitability; the degree to which a procedure may be split without replication of any of the code in its body. The paper then uses an implementation of our greedy splitting algorithm to empirically investigate splitability and its correlation to procedure size.

The paper addresses three research questions RQ1-RQ3 to evaluate both the splitting technique and the splitability of code:

RQ1: How frequently are there splitable procedures in the programs studied?

RQ2: How much repeated code will need to be generated as a result of splitting. A perfect split requires no repetition. The more repetition is required, the less splitable the procedure is.

RQ3: What is the relationship between *splitability* and the size of a procedure?

The central finding of the paper lies in the answer to the last question. The paper demonstrates that there is, indeed, a statistically significant correlation between increasing procedure size and increasing splitablity. This is good news for re-engineers who may naturally consider large procedures as starting candidates for re-engineering activities.

To answer these questions, the paper presents three related empirical investigations. The investigations are performed on six different open source c programs (See Table 2). The primary contributions of the paper are:

1. An algorithm based on greedy application of dependence analysis is introduced to identify split points in procedures.

2. The empirical study shows that a surprisingly large proportion of the procedures considered are splitable.

3. The paper provides evidence for a strong statistical correlation between procedure size and splitability in the programs studied.

The next section, Section 2, defines splitable procedure and introduces the splitting algorithm. Section 3 presents the empirical study and answers the research questions posed in Section 1. Section 4 describes related work while Section 5 draws conclusions.

2 The Splitting Algorithm

Whether or not a procedure is apt to being split depends on the structure of the procedure. The possibility of splitting is increased if components of the procedure have little in common. The attribute of splitting (or splitability) is closely related to that of cohesion. Procedures are more inclined to split the lower their cohesion level is. This observation is inspired by Bieman and Ott [1] who use slicing as a mechanism to evaluate cohesion.

The CodeSurfer scripting language based on Scheme-STK [7] is used to process program code represented as a System Dependence Graph SDG [9] and to calculate the slices. Slicing is performed on every SDG node, and, therefore, for a procedure of n nodes, the search space for procedure splitting consists of 2^n subsets of slices. Next, the concept of splitable procedure is introduced:

Definition 1 $maximal(s)$
For the procedure P, let $Slices(P)$ be all the backward slices of P based on each SDG node as slicing criterion. A slice s which is not included in any other slices is called a maximal slice. Thus, s is $maximal(s) \iff \forall s', s' \neq s \Rightarrow s \nsubseteq s'$

Definition 2 $Splitable\ Procedure$
A procedure P is splitable if and only if there exist at least $s_1, s_2 \in Slices(P) \wedge s_1 \neq s_2 \wedge maximal(s_1) \wedge maximal(s_2)$

The purpose of applying the optimisation-based slicing approach advocated here is to identify split points automatically. This paper uses a greedy algorithm [16] to tackle this problem. The algorithm searches for a set of slices representing procedure components that make the overlap of these slices as small as possible.

The purpose of our splitting algorithm is to locate the best (or an acceptably good) solution among a number of possible solutions in the search space. The process of looking for a solution is equivalent to that of looking for a set of slices in the search space with minimal overlap among the slices. Clearly, enumeration will not be possible since the search space grows exponentially in n, the size of the program.

Though this is an exponentially large search space, the underlying optimisation problem is a set cover problem. Set cover problems submit to optimisation using greedy-based algorithms. That is, the greedy algorithm is known to produce solutions within a log of the global optimum for set cover [5, 20].

In employing the greedy algorithm presented in Figure 2, a slice is represented as a sequence of binary $0, 1$ digits in a two-dimensional matrix, where columns correspond to SDG nodes and rows the slices. Here, a "1" and "0" correspond to whether a SDG node belongs or does not belong to the slice, respectively. In the same figure, a $groups[N]$ represents a set of slices corresponding to a subprocedure.

The splitting algorithm proceeds as follows:

1. Construct all the static backward slices in a procedure

by slicing with respect to all the SDG nodes in the corresponding SDG.

2. Find sets of slices representing procedure components that make the overlap of these slices as small as possible. A greedy algorithm is used to construct such a minimal overlap set is described in Figure 2.

3. Recover slice statements from the corresponding SDG nodes by combining nodes that belong to a single statement. An example illustrating such combination is shown in Figure 1 where the SDG nodes of a statement and their dependency relationships are depicted.

4. Make the subprocedures obtained executable.

This paper focusses on steps 1 and 2 in order to investigate splitability.

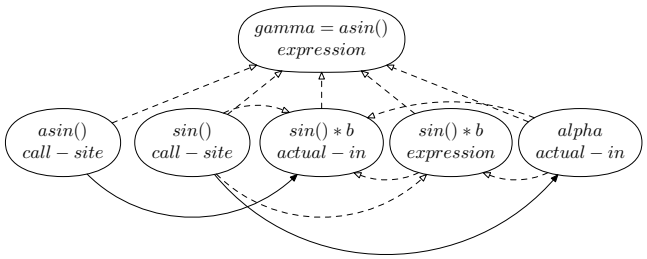

Figure 1: An example of recovering the statement $gamma = asin(sin(alpha)*b)$ from the corresponding six SDG nodes. Solid lines and dashed lines represent control and data dependency, respectively. The SDG node type (call-site, actual-in and expression) is shown under the SDG nodes.

One can always split a procedure into a set of slices where there are 'useless' slices that are totally included within others. Thus, to obtain useful and independent parts, $maximal(s)$ is defined and the greedy dependence-based splitting algorithm is introduced to decompose procedures according to Definition 2.

In the illustrative example presented in Figure 3, Procedure $Sum\_and\_Product$ includes two computations for Variables Sum and $Prod$. For simplicity, the example of Figure 3 only includes statements that correspond to single SDG nodes. In general, a statement may have several SDG nodes as explained in Figure 1. Table 1 shows all the backward slices for each SDG node of the procedure. The splitting algorithm is to find a set of maximal slices, each of

// **The greedy algorithm**
1 $slices[N]$; Array of Array of [0,1] sequence, slices[i] is a sequence of [0,1] and corresponds to node i as a slicing criterion.
2 $groups[N]$; Set of slices, a groups[i] represents Subprocedure i.
3 $count = 1$; Integer, represents the number of subprocedures.
4 $sort\ slices\ according\ to\ their\ sizes\ in\ descending\ order$;
5 $groups[1] = \{slices[1]\}$;
6 **for** $I := 2\ to\ N$ **do**
7 **if** $slices[I]\ \epsilon\ groups[count]$ **then** $continue$;
8 **else**
9 $count$++;
10 $groups[count] = slices[I]$;
11 **end**
12 **end**

Figure 2: Procedure splitting algorithm

```
Sum_and_Product ((1) int N, (2) int Sum,
    (3) int Prod){
(4) int I;
(5) Sum=0;
(6) Prod=1;
(7) for(I = 1; I <= N; I++){
(8)     Sum=Sum+I;
(9)     Prod=Prod*I;}}
```

Figure 3: An example for the splitting algorithm. The digits represent the SDG nodes of the procedure.

which represents an independent sub-procedure. In terms of Definition 1, $slice8$ and $slice9$ are two maximal slices and the rests are not, due to the fact that $slice2$ and $slice5$ are included in $slice8$, $slice3$ and $slice6$ are included in $slice9$ and $slice1$, $slice4$ and $slice7$ are included in both. In fact, $maximal(s)$ represents useful independent computation in a procedure.

In the procedure shown in Table 1 and the splitting algorithm in Figure 2, $slices[N]$ represents 9 slices of the procedure; $groups[N]$ represents a set of maximal slices whilst $count$ represents the number of maximal slices with initial value being 1. After sorting, the first maximal slice is $slice8$

assigned to $groups[1]$. The for loop is to find out more maximal slices if they exist. In this case, $slice9$ is another maximal slice.

Program slices	SDG nodes								
	1	2	3	4	5	6	7	8	9
slice1	1	0	0	0	0	0	0	0	0
slice2	0	1	0	0	0	0	0	0	0
slice3	0	0	1	0	0	0	0	0	0
slice4	0	0	0	1	0	0	0	0	0
slice5	0	1	0	0	1	0	0	0	0
slice6	0	0	1	0	0	1	0	0	0
slice7	1	0	0	1	0	0	1	0	0
slice8	1	1	0	1	1	0	1	1	0
slice9	1	0	1	1	0	1	1	0	1

Table 1: The backward slices of the splitable procedure in Figure 3. A 1 represents a SDG node that is included in the slice, while a 0 represents a SDG node that is not included in the slice.

3 Empirical Study

Our algorithm explores procedure structure automatically and provides candidates for splitting. In this section, procedure structure and relationships between procedure components of six open source systems are analysed. The structure and relationships are evaluated in terms of code overlap, splitability and the correlation between procedure size and splitability.

3.1 RQ1: How frequently are there splits?

It is not always the case that a procedure can be split into two or more subprocedures. This section explores the frequency of occurrence of splitable and non-splitable procedures.

Table 2 shows the size of programs with entry 'Loc' for the lines of code, 'Descriptions' for the functionality of programs, 'Vertices' for the number of SDG nodes corresponding to source code, 'Number of Procs' for the total number of procedures in the entire program, entry 'Split procs' represents how many procedures in the program can be split into two or more, 'Count' represents the number of procedures which can be split and 'Percentage' represents the ratio between 'Count' and 'Number of Procs'.

Table 2 provides information about how many procedures in the program can be split and the percentage of this to the total number of procedures. The results indicate that more than 20% of procedures (average 23.6%) in each of the six programs include independent procedure components contributing to the whole functionality.

Figure 4 shows the percentage of frequency distributions of procedures in terms of the number of subprocedures obtained by splitting. Each column in the figure represents the percentage of the number of procedures that can be split to the total number of procedures in the entire program. Here, the x-axis entries represent the number of subprocedures obtained as a result of splitting, where 1 represents non-splitable procedures. All figures show a maximum split level of 4. However, (b), (e) and (f) have a very few exceptional procedures that can be split into 9, 6 and 13 subprocedures respectively. Fewer than 1% of the procedures can be split into more than 4 subprocedures so, for clarity, they are not shown.

The results depicted in Figure 4 show that the majority of procedures falls into the first column indicating that most procedures are not splitable. As far as procedure cohesion is concerned, it is preferable that a procedure only completes one single task, thus making programs more efficient and maintainable [2, 3, 18]. However, in some cases, procedures can be split into two or more subprocedures. This suggests possible 'granularity of modularisation' issues; perhaps procedures should be split to aid on-going maintainability.

3.2 RQ2: Splitability of Procedures

This section analyses the dependency amongst subprocedures of each splitable procedure and explores the overlap and splitability distributions. To this end, code overlap and splitability measures are defined in terms of the subprocedures' sizes.

Since, in general, different procedure components are not completely independent, subprocedures might share some common SDG nodes whose computation contributes to all the subprocedures. In this case, there is extra repeated code generated and shared between different subprocedures. As can be seen from Figure 4, the majority of splitable procedures are split into two or three subprocedures, to be referred to as 2-way and 3-way splitable procedures, respectively. In what follows, we concentrate on this group of splitable procedures, disregarding those that could potentially be split into more than three subprocedures.

Depending on the size of the overlap or repeated code common to all subprocedures, different levels of splitability can be defined. Evaluating repeated code shared by all sub-

Programs	Description	Size		Number of Procs	Split Procs	
		Loc	Vertices		Count	Percentage
termutils-2.0	terminal control utilities	4,334	2,952	56	15	26.8
acct-6.3	accounting utilities package	6,178	4,305	88	24	27.3
space	ESA ADL interpreter	9,106	9,887	137	25	18.2
oracolo2	an array processor	14,326	8,776	135	25	18.5
byacc-1.9	LALR parser generator	6,420	8,046	178	50	28.1
a2ps-4.1	Text to postscript converter	20,407	17,226	248	55	22.2
Total		60,771	51,192	842	194	23.0
Average		10,129	8,532	141	33	23.6

Table 2: The set of programs studied.

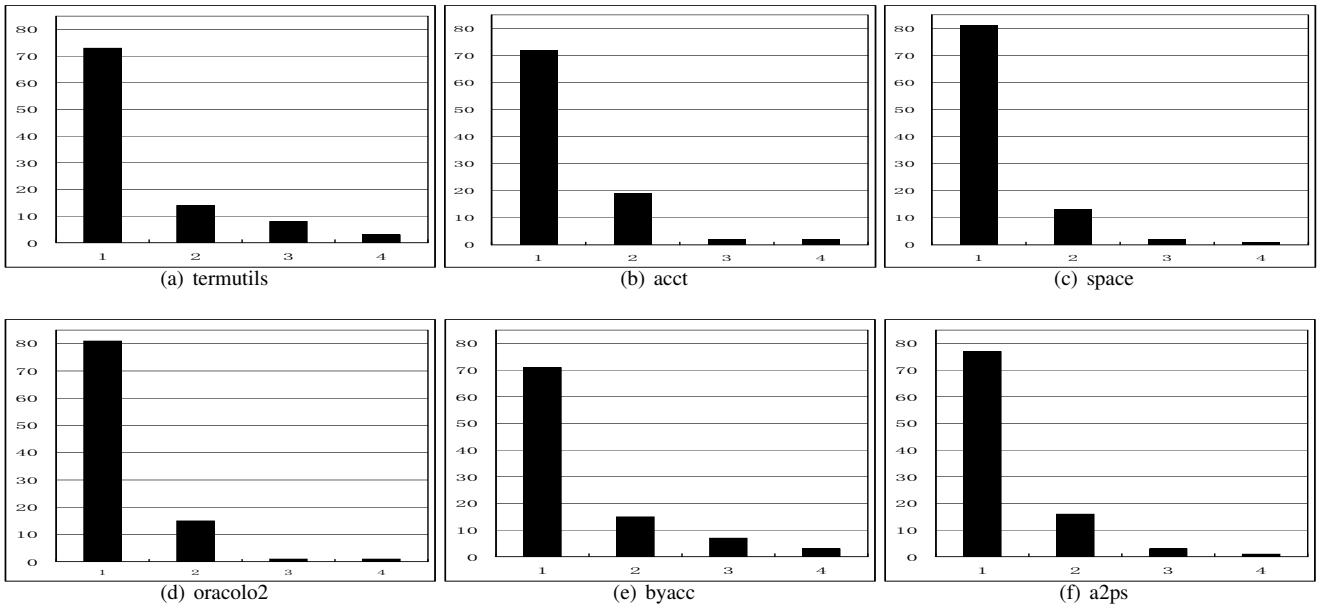

Figure 4: Distribution of splitable procedures. A column at horizontal position x of height y means that $y/N\%$ of procedures are split into x subprocedures, where N is the total number of procedures.

procedures suggests a measure of dependency between subprocedures. Splitability decreases as dependency between subprocedures increases, because too many repeated SDG nodes imply high inter-dependency relatedness between different procedure components. The splitability measure expresses how tightly different procedure components are related. High splitability indicates that procedures tend to compute more than a single and independent functionality and vice versa.

In order to define procedure splitability as a quantifiable measure, the following definitions are required. These definitions capture different metrics for assessing splitability. Let S_i and S_j denote two subprocedures of a splitable procedure, $Size(S_i)$ and $Size(S_j)$ denote the sizes, i.e., the number of SDG nodes of S_i and S_j, respectively. Further, let $Size(S_i \cap S_j)$ be the number of the SDG nodes shared by the two subprocedures S_i and S_j and $Max(Size(S_i), Size(S_j))$ be the greater value of the two sizes.

Definition 3 ***Overlap between two subprocedures of 2-way splitable procedures***
Overlap of 2-way splitable procedures, σ^{12}, is the ratio of the number of SDG nodes shared by the two subprocedures to the number of SDG nodes of the subprocedure of greater size,

$$\sigma^{12} = \frac{Size(S_1 \cap S_2)}{Max(Size(S_1), Size(S_2))} \qquad (I)$$

Definition 4 ***Min-Overlap amongst three subprocedures of 3-way splitable procedures***
Min-Overlap of 3-way splitable procedures, σ^{123}_{min}, is the ratio of the number of SDG nodes shared by the three subprocedures to the number of SDG nodes of the subprocedure of greatest size,

$$\sigma^{123}_{min} = \frac{Size(S_1 \cap S_2 \cap S_3)}{Max(Size(S_1), Size(S_2), Size(S_3))} \qquad (II)$$

Definition 5 ***Max-Overlap amongst three subprocedures of 3-way splitable procedures***
Max-Overlap of 3-way splitable procedures, σ^{123}_{max}, is the biggest Overlap of three pairs of subprocedures. If

$$\sigma^{12} = \frac{Size(S_1 \cap S_2)}{Max(Size(S_1), Size(S_2))} ,$$

$$\sigma^{13} = \frac{Size(S_1 \cap S_3)}{Max(Size(S_1), Size(S_3))} , \quad and$$

$$\sigma^{23} = \frac{Size(S_2 \cap S_3)}{Max(Size(S_2), Size(S_3))}$$

then

$$\sigma^{123}_{max} = Max(\sigma^{12}, \sigma^{13}, \sigma^{23}) \qquad (III)$$

Overlap represents the cohesive degree between subprocedures, that is, the more overlap between subprocedures is, the more cohesive a procedure is, and therefore, the less likely the procedure is splitable. Splitability is therefore measured as the opposite of overlap. Thus,

Definition 6 ***Splitability between two subprocedures of 2-way splitable procedures***
Splitability of a 2-way splitable procedure is defined as the opposite of Overlap of the procedure.

$$\mathcal{S} = 1 - \sigma^{12} \qquad (IV)$$

Definition 7 ***Max-Splitability amongst three subprocedures of 3-way splitable procedures***
Max-Splitability of a 3-way splitable procedure is defined as the opposite of Min-Overlap of the procedure.

$$\mathcal{S}_{max} = 1 - \sigma^{123}_{min} \qquad (V)$$

Definition 8 ***Min-Splitability amongst three subprocedures of 3-way splitable procedures***
Min-Splitability of a 3-way splitable procedure is defined as the opposite of Max-Overlap of the procedures.

$$\mathcal{S}_{min} = 1 - \sigma^{123}_{max} \qquad (VI)$$

Figure 5 shows overlap frequency distribution of splitable procedures that are split into two subprocedures. The x-axis represents the range of the overlap as defined in (I). The y-axis represents the percentage of the number of the splitable procedures which lie in the corresponding overlap range to the number of all the 2-way splitable procedures of each program.

The overlap distribution expresses dependency between two different procedure components. Locations closer to the origin indicate low inter-dependency implying high splitability. In this region, procedures can be split without generating a large amount of repeated code, i.e., their components share few common SDG nodes.

For locations away from the origin, splitability of procedures decreases since increasing overlap between two subprocedures indicates high inter-dependency between subprocedures. That is, the different components of a procedure share quite a few common SDG nodes. In this case, splitting generates a large amount of repeated code.

Turning to 3-way splitable procedures, there are two empirical investigations of procedure overlap corresponding to definitions (II) and (III). Figure 6 shows the overlap distribution for 3-way splitable procedures in terms of the overlap measure defined in (II). For low overlap ranges, splitable procedures consist of multi-functionality, where at least two subprocedures are relatively independent and do not share many common SDG nodes. The distributions of the six programs considered in this study exhibit different behaviour.

Figure 7 shows the overlap distribution for 3-way splitable procedures corresponding to maximum overlap σ^{123}_{max} as defined in (III). This figure shows a different behaviour to that corresponding to minimum overlap of Figure 6. For

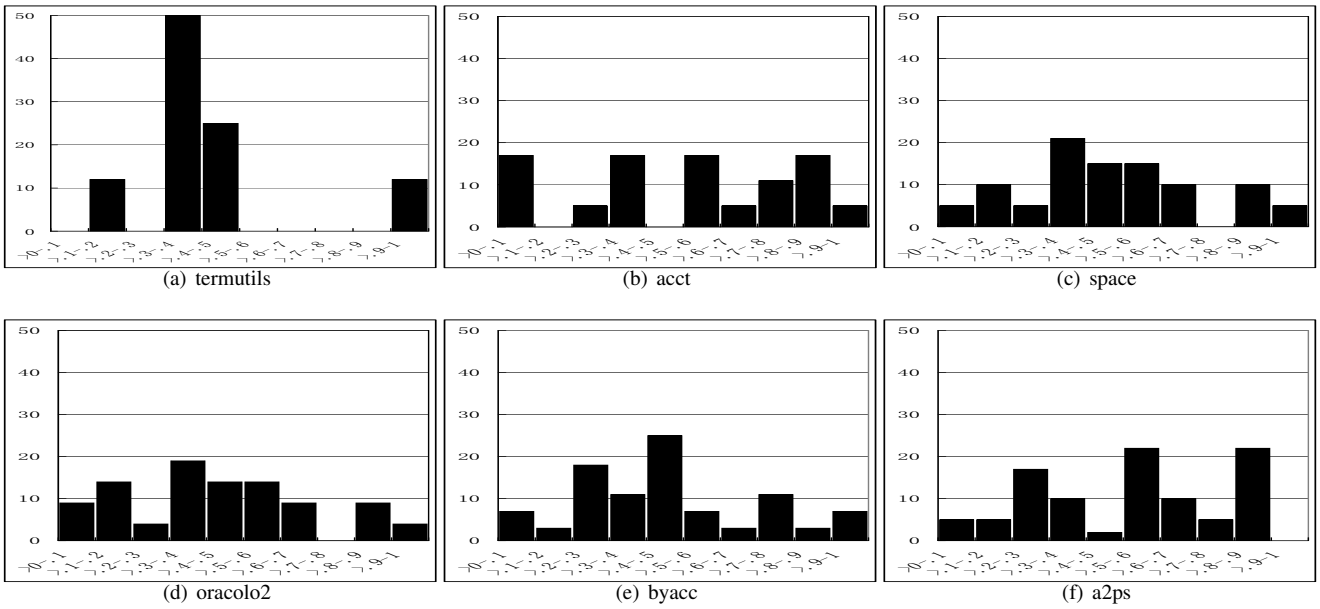

Figure 5: Overlap frequency distribution of splitable procedures that can be split into two subprocedures. The x-axis measures overlap using σ^{12} defined in (I). The y-axis represents the percentage of 2-way splitable procedures that lie in the corresponding overlap range.

low overlap ranges, splitable procedures consist of multi-functionality where all the three subprocedures are relatively more independent of each other and all do not share many common SDG nodes.

3.3 RQ3: Exploration of correlation between procedure size and splitability

This section explores the correlation between procedure size defined in terms of SDG nodes and splitability as defined in (IV), (V) and (VI). That is, how the size of a procedure is related to its splitability. Both, 2-way splitable and 3-way splitable procedures are considered. Intuitively, procedures with relatively small size should turn out to be barely splitable; with increasing procedure size, the procedures would be expected to be more likely to split into more subprocedures.

The Spearman correlation coefficient (ρ) is used in this paper to investigate the relationship between a procedure's size and its splitability.

Table 3 shows the first correlation between procedure size and splitability in terms of splitability definition (IV), where procedures are split into two subprocedures. For all the six programs considered in our empirical study, except `a2ps`, ρ lies between 0.7 and 0.9 indicating a strong correlation between procedure size and splitability. For `a2ps`, the ρ-value approaches 1 indicating a stronger correlation. This kind of strong or very strong correlation indicates that as procedure size increases, procedure's splitability increases. Our results show that, with increasing procedure size, there is a trend to 2-way splitability.

Table 4 shows the second correlation between procedure size and splitability in terms of splitability as defined in (V) where procedures are split into three subprocedures. The ρ-values for `termutils` and `a2ps` lie between 0.0 and 0.2, which denotes very weak to negligible correlation. The ρ-value for `space` is between 0.2 and 0.4, which indicates weak or low correlation. In this case, there is no significant correlation between procedure size and splitability. However, the ρ-value for `byacc` lies between 0.9 and 1 indicating very strong correlation. For `acct` and `oracolo2`, since the number of ranks is too small, the ρ-value is not statistically significant.

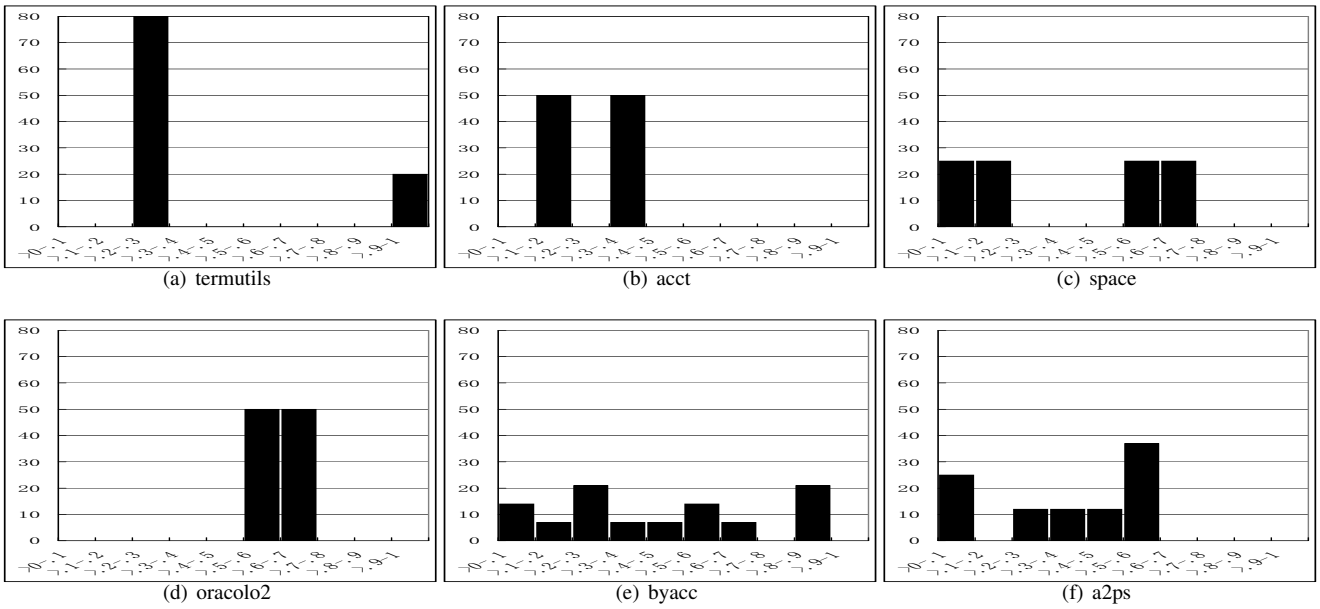

Figure 6: Overlap frequency distribution of splitable procedures that can be split into three subprocedures. The x-axis measures overlap using σ^{123}_{min} defined in (II). The y-axis represents the percentage of 3-way splitable procedures that lie in the corresponding overlap range.

Programs	termutils	acct	space	oracolo2	byacc	a2ps
Spearman coefficient (Correlation)	0.81	0.73	0.82	0.73	0.83	0.92
Number of 2-way Splitable procedures (Ranks)	8	17	19	21	27	40

Table 3: Correlation between procedure size and splitability of procedures where splitable procedures can be split into two subprocedures in terms of $\mathcal{S}$ defined in (IV).

Spearman Correlation data corresponding to Definition (VI) shows similar results to those of Table 4, i.e, there is no significant strong correlation between procedure size and splitability, except for `byacc`.

In summary, these results indicate that there is no consistent correlation between procedure size and procedure's splitability into three subprocedures, though there is for splitability into two subprocedures.

4 Related Work

Several authors have addressed the question of measuring procedure cohesion [1, 18, 19]. The previous work uses slicing based approaches to analyse and quantify the measurement of cohesion of procedures in terms of output variables. However, no proposal has been made to automate the process of finding procedure components as a first step for measuring cohesion.

Ott and Thuss [18] quantify cohesion defined by Constantine and Yourdon [3] using program slicing. Thus, the

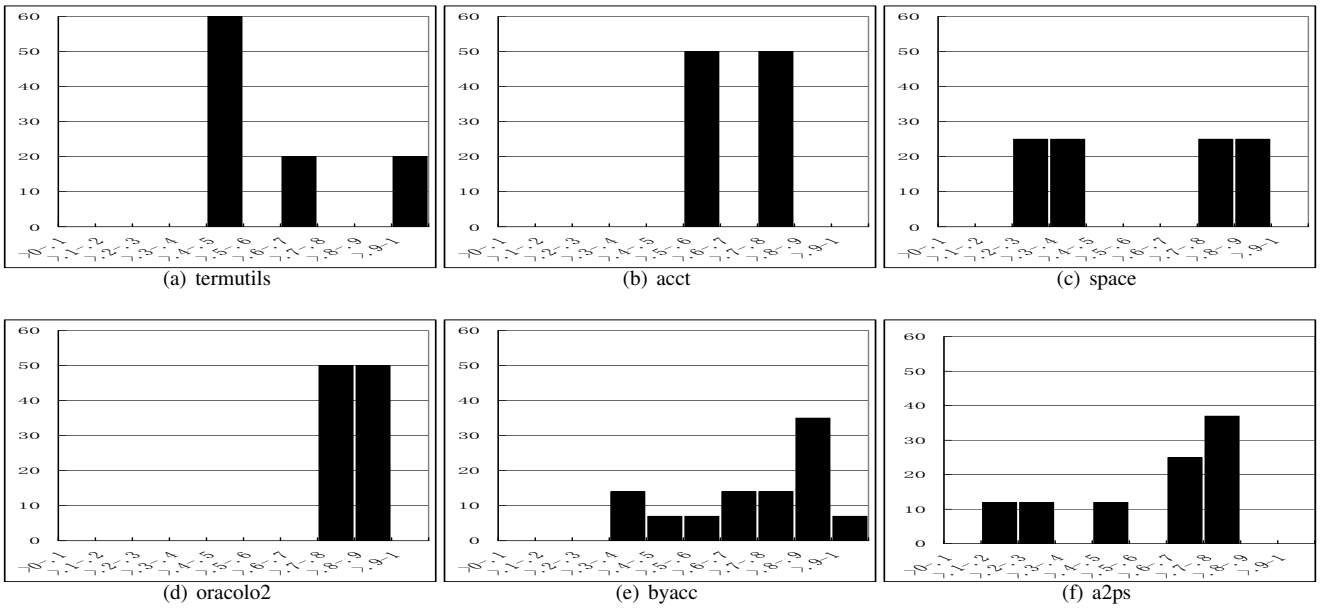

Figure 7: Overlap frequency distribution of splitable procedures that can be split into three subprocedures. The x-axis measures overlap using σ^{123}_{max} defined in (III). The y-axis represents the percentage of 3-way procedures that lie in the corresponding overlap range.

Programs	termutils	acct	space	oracolo2	byacc	a2ps
Spearman Coefficient (Correlation)	0.05	1.0	0.40	1.0	0.93	0.06
Number of 3-way Splitable procedures (Ranks)	5	2	4	2	14	8

Table 4: Correlation between procedure size and splitability of procedures where splitable procedures can be split into three subprocedures in terms of $\mathcal{S}_{max}$ defined in (V).

quantitative measurement is specified to evaluate quality of programs. However, in this work, no methodology for automating procedure splitting to explore procedure cohesion has been suggested. Moveover, no empirical study for evaluating single procedure cohesion on existing open sources samples has been performed. The work reported here helps to analyse the structure of procedures by dividing each procedure into subprocedures moving this previous work from passive measurement to active improvement.

There are other techniques that extract a single procedure component as a new separate procedure to re-factor the original procedure. For example, Lakhotia [11] combines program slicing and transformations to 'tuck' a code fragment in a procedure as a new subprocedure and replace the subprocedure code through a procedure call. As a result, the cohesion level is improved for the whole procedure without changing semantics.

However, 'tucking' does not consider the whole structure of procedure; it only extracts some concerned code fragment based upon the slice of an interesting statement. Another example, is the "Extract Method" refactoring transformation [6, 17] which aims at improving the internal structure of the

program. The approach followed in this paper differs from refactoring in exploring the program structure through an empirical study.

Similarly, there are other procedure refactoring techniques such as those that use program slicing [4, 14] and transformation techniques [8, 10] for procedure extraction. These techniques all focus on single procedure extraction for reuse, rather than splitting procedures. As such, these approaches allow for the construction of subprocedures, though cannot explore the entire structure of procedure and obtain all possible subprocedures. Our approach employs a splitting algorithm which allows for the splitting of procedure into its components. As has been shown, there may be many such subprocedures not merely one. The results reported are also the first to statistically explore the relationship between procedure size and procedure splitability.

5 Conclusions

The paper analyses procedure splitability. To this end, an empirical study is performed using six real world open source programs. The empirical investigation targeted, firstly, evaluating program structure in terms of splitable procedure distribution, secondly, defining procedure splitability for 2-way and 3-way splitable procedures, and analysing the splitability distribution for these two classes of splitable procedures, and thirdly exploring the correlation between procedure size and procedure splitability. The results show that there is a strong correlation between procedure size and procedure splitability in the case of 2-way splitable procedures. However, the results indicate no strong correlation in the case of 3-way splitable procedures for five out of six programs.

References

[1] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[2] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, 1976.

[3] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice Hall, 1979.

[4] R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 93–101, 2004.

[5] U. Feige and P. Tetali. Approximating min sum set cover. In *Algorithmica*, volume 40, pages 219–234. Springer New York, Sept. 2004.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] Grammatech Inc. The codesurfer slicing system, 2002. url:www.grammatech.com.

[8] M. Harman, D. Binkley, R. Singh, and R. Hierons. Amorphous procedure extraction. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 85–94, Sept. 2004.

[9] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. Proceedings in *ACM SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[10] R. Komondoor and S. Horwitz. Effective automatic procedure extraction. In 11^{th} *IEEE International Workshop on Program Comprehension*, pages 33–43, Los Alamitos, California, USA, May 2003.

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

[12] M. M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[13] M. M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, Jan. / Feb. 1998.

[14] K. Maruyama. Automated method-extraction refactoring by using block-based slicing. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 31–40, New York, NY, USA, 2001. ACM Press.

[15] T. Meyers and D. W. Binkley. Slice-based cohesion metrics and software intervention. In 11^{th} *IEEE Working Conference on Reverse Engineering*, pages 256–266, Los Alamitos, California, USA, Nov. 2004.

[16] H. Naeimi and A. DeHon. A greedy algorithm for tolerating defective crosspoints in NanoPLA design. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 49–56, 2004.

[17] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[18] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM Conference on Software Engineering*, pages 198–204, May 1989.

[19] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Los Alamitos, California, USA, May 1993.

[20] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 1998.