Coherent Clusters in Source Code

Syed Islam, Jens Krinke, David Binkley*, Mark Harman

University College London
*Loyola University Maryland**

Abstract

This paper presents the results of a large scale empirical study of *coherent dependence clusters*. All statements in a coherent dependence cluster depend upon the same set of statements and affect the same set of statements; a coherent cluster's statements have 'coherent' shared backward and forward dependence. We introduce an approximation to efficiently locate coherent clusters and show that it has a *minimum* precision of 97.76%. Our empirical study also finds that, despite their tight coherence constraints, coherent dependence clusters are in abundance: 23 of the 30 programs studied have coherent clusters that contain at least 10% of the whole program. Studying patterns of clustering in these programs reveals that most programs contain multiple significant coherent clusters. A series of case studies reveals that these major clusters map to logical functionality and program structure. For example, we show that for the program acct, the top five coherent clusters all map to specific, yet otherwise non-obvious, functionality. Cluster visualization can also reveal subtle deficiencies in program structure and identify potential candidates for refactoring efforts. Finally a study of inter-cluster dependence is used to highlight how coherent clusters built are connected to each other, revealing higher-level structures, which can be used in reverse engineering.

Keywords: Dependence analysis, program comprehension, program slicing, clustering, re-engineering, structural defect, dependence pollution, inter-cluster dependence

1. Introduction

Program dependence analysis is a foundation for many activities in software engineering such as testing, comprehension, and impact analysis [8]. For example, it is essential to understand the relationships between different parts of a system when making changes and the impacts of these changes [22]. This has led to both static [44, 16] and blended (static and dynamic) [36, 37] dependence analyses of the relationships between dependence and impact.

One important property of dependence is the way in which it may cluster. This occurs when a set of statements all depend upon one another, forming a

dependence cluster. Within such a cluster, any change to an element potentially affects every other member of the cluster. If such a dependence cluster is very large, then this mutual dependence clearly has implications on the cost of maintaining the code.

In previous work [10], we introduced the study of dependence clusters in terms of program slicing and demonstrated that large dependence clusters were (perhaps surprisingly) common, both in production (closed source) code and in open source code [26]. Our findings over a large corpus of C code was that 89% of the programs studied contained at least one dependence cluster composed of 10% or more of the program's statements. The average size of the programs studied was 20KLoC, so these clusters of more than 10% denoted significant portions of code. We also found evidence of super-large clusters: 40% of the programs had a dependence cluster that consumed over half of the program.

More recently, our finding that large clusters are widespread in C systems has been replicated for other languages and systems, both in open source and in proprietary code [1, 7, 40]. Large dependence clusters were also found in Java systems [7, 38, 40] and in legacy Cobol systems [25].

Recently, there has been interesting work on the relationship between faults, program size, and dependence clusters [15], and between impact analysis and dependence clusters [1, 26]. Large dependence clusters can be thought of as dependence 'anti-patterns' because the high impact of changes may lead to problems for on-going software maintenance and evolution [1, 9, 38]. As a result, refactoring has been proposed as a technique for splitting larger clusters of dependence into smaller clusters [10, 14].

Dependence cluster analysis is complicated by the fact that inter-procedural program dependence is non-transitive, which means that the statements in a traditional dependence cluster, though they all depend on each other, may not each depend on the same set of statements, nor need they necessarily affect the same set of statements external to the cluster.

This paper introduces and empirically studies[1] *coherent dependence clusters*. In a coherent dependence cluster all statements share identical intra-cluster and extra-cluster dependence. A coherent dependence cluster is thus more constrained than a general dependence cluster. A coherent dependence cluster retains the essential property that all statements within the cluster are mutually dependent, but adds the constraint that all incoming dependence must be identical and all outgoing dependence must also be identical. That is, all statements within a coherent cluster depend upon the same set of statements outside the cluster and all statements within a coherent cluster affect the same set of statements outside the cluster.

This means that, when studying a coherent cluster, we need to understand only a single external dependence context in order to understand the behavior of the entire cluster. For a dependence cluster that fails to meet the external constraint, statements of the cluster may have a different external dependence

[1]Preliminary results were presented at PASTE [29].

context. This is possible because inter-procedural dependence is non-transitive.

It might be thought that very few sets of statements would meet these additional coherence constraints, or that, where such sets of statements do meet the constraints, there would be relatively few statements in the coherent cluster so-formed. Our empirical findings provide evidence that this is not the case: coherent dependence clusters are common and they can be very large. This finding provides a new way to investigate the dependence structure of a program and the way in which it clusters. This paper presents empirical results that highlight the existence and applications of coherent dependence clusters.

The primary contributions of the paper are as follows:

1. Empirical analysis of thirty programs assesses the frequency and size of coherent dependence clusters. The results demonstrate that large coherent clusters are common validating their further study.
2. Two further empirical validations consider the impact of data-flow analysis precision and the precision of an approximation used to compute coherent clusters.
3. A series of four case studies shows how coherent clusters identify logical program structures.
4. A study of inter-cluster dependence highlights how coherent clusters form the building blocks of larger dependence structures that can support, as an example, reverse engineering.

The remainder of this paper is organized as follows: Section 2 provides background on coherent clusters and their visualization. Section 3 provides details on the subject programs, the validation of the slice approximation used, and the experimental setup. This is followed by quantitative and qualitative studies into the existence and impact of coherent dependence clusters and the inter-cluster dependence study. Section 4 considers related work and finally, Section 5 summarizes the work presented.

2. Background

This section provides background on dependence clusters. It first presents a sequence of definitions that culminate in the definition for a coherent dependence cluster. Then, it reviews existing dependence cluster visualizations including the cluster visualization tool *decluvi*. Previous work [10, 26] has used the term *dependence cluster* for a particular kind of cluster, termed a *mutually-dependent cluster* herein to emphasize that such clusters consider only mutual dependence internal to the cluster. This distinction allows the definition to be extended to incorporate external dependence.

2.1. Dependence Clusters

Informally, *mutually-dependent clusters* are maximal sets of program statements that mutually depend upon one another [26]. They are formalized in terms of mutually dependent sets in the following definition.

Definition 1 (Mutually-Dependent Set and Cluster [26])
A *mutually-dependent set (MDS)* is a set of statements, S, such that
$\forall x, y \in S : x$ depends on y.
A *mutually-dependent cluster* is a maximal MDS; thus, it is an MDS not properly contained within another MDS.

The definition of an MDS is parameterized by an underlying *depends-on* relation. Ideally, such a relation would precisely capture the impact, influence, and dependence between statements. Unfortunately, such a relation is not computable. A well known approximation is based on Weiser's *program slice* [42]: a slice is the set of program statements that affect the values computed at a particular statement of interest (referred to as a slicing criterion). While its computation is undecidable, a minimal (or precise) slice includes exactly those program elements that affect the criterion and thus can be used to define an MDS in which t depends on s iff s is in the minimal slice taken with respect to slicing criterion t.

The slice-based definition is useful because algorithms to compute approximations to minimal slices can be used to define and compute approximations to mutually-dependent clusters. One such algorithm computes a slice as the solution to a reachability problem over a program's *System Dependence Graph* (SDG) [27]. An SDG is comprised of vertices, which essentially represent the statements of the program and two kinds of edges: data dependence edges and control dependence edges. A data dependence connects a definition of a variable with each use of the variable reached by the definition [21]. Control dependence connects a predicate p to a vertex v when p has at least two control-flow-graph successors, one of which can lead to the exit vertex without encountering v and the other always leads eventually to v [21]. Thus p controls the possible future execution of v. For structured code, control dependence reflects the nesting structure of the program. When slicing an SDG, a slicing criterion is a vertex from the SDG.

A naïve definition of a dependence cluster would be based on the transitive closure of the dependence relation and thus would define a cluster to be a strongly connected component. Unfortunately, for certain language features, dependence is non-transitive. Examples of such features include procedures [27] and threads [31]. Thus, in the presence of these features, strongly connected components overstate the size and number of dependence clusters. Fortunately, context-sensitive slicing captures the necessary context information [10, 27, 32, 13, 33].

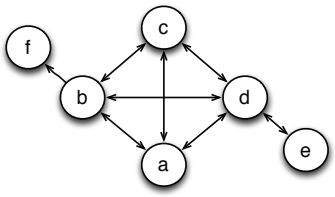
Two kinds of SDG slices are used in this paper: backward slices and forward slices [27, 35]. The backward slice taken with respect to vertex v, denoted $\mathsf{BSlice}(v)$, is the set of vertices reaching v via a path of control and data dependence edges where this path respects context. The forward slice, taken with respect to vertex v, denoted $\mathsf{FSlice}(v)$, is the set of vertices reachable from v via a path of control and data dependence edges where this path respects context.

The program P shown in Figure 1 illustrates the non-transitivity of slice inclusion. The program has six assignment statements (assigning the variables a,

backward slice on assignment to						P
a	b	c	d	e	f	
						1:
|	|	|	|			2: f1(x) {
|	|	|	|			3: **a** = f2(x, 1) + f3(x);
|	|	|	|			4: return f2(a, 2) + f4(a);
						5: }
						6:
|	|	|	|		|	7: f2(x, y) {
|	|	|	|		|	8: **b** = x + y;
|	|	|	|		|	9: return b;
						10: }
						11:
|	|	|	|			12: f3(x) {
|	|	|	|			13: if (x>0) {
|	|	|	|			14: **c** = f2(x, 3) + f1(x);
|	|	|	|			15: return c;
						16: }
|	|	|	|			17: return 0;
						18: }
						19:
|	|	|	|	|		20: f4(x) {
|	|	|	|	|		21: **d** = x;
|	|	|	|	|		22: return d;
						23: }
						24:
			|	|		25: f5(x) {
			|	|		26: **e** = f4(5);
			|			27: return f4(e);
						28: }
						29:
	|				|	30: f6(x){
					|	31: **f** = f2(42, 4);
						32: return f;
						33: }
						34:

Figure 1: Dependence intransitivity and clusters

b, c, d, e and f) whose dependencies are shown in columns 1–6 as backward slice inclusion. Backward slice inclusion contains statements that affect the slicing criterion through data and control dependence. The dependence relationship between these statements is also extracted and shown in Figure 2 using a directed graph where the nodes of the graph represent the assignment statements



Slice Criterion	Backward Slice	Forward Slice
a	{a, b, c, d}	{a, b, c, d}
b	{a, b, c, d}	{a, b, c, d, f}
c	{a, b, c, d}	{a, b, c, d}
d	{a, b, c, d, e}	{a, b, c, d, e}
e	{d, e}	{d, e}
f	{b, f}	{f}

Figure 2: Backward slice inclusion relationship for Figure 1

and the edges represent the backward slice inclusion relationship from Figure 1. The table on the right of Figure 2 also gives the forward slice inclusions for the statements. All other statements in P, which do not define a variable, are ignored. In the diagram, x depends on y (y ∈ BSlice(x)) is represented by y → x. The diagram shows two instances of dependence intransitivity in P. Although b depends on nodes a, c, and d, node f, which depends on b, does not depend on a, c, or d. Similarly, d depends on e but a, b, and c, which depend on d do not depend on e.

2.2. Slice-based Clusters

A *slice-based cluster* is a maximal set of vertices included in each others slice. The following definition essentially instantiates Definition 1 using BSlice. Because $x \in \mathsf{BSlice}(y) \Leftrightarrow y \in \mathsf{FSlice}(x)$ the dual of this definition using FSlice is equivalent. Where such a duality does not hold, both definitions are given. When it is important to differentiate between the two, the terms *backward* and *forward* will be added to the definition's name as is done in this section.

Definition 2 (Backward-Slice MDS and Cluster [26])
A *backward-slice MDS* is a set of SDG vertices, V, such that
$\forall x, y \in V : x \in \mathsf{BSlice}(y)$.
A *backward-slice cluster* is a backward-slice MDS contained within no other backward-slice MDS.

In the example shown in Figure 2, the vertices representing the assignments to a, b, c and d are all in each others backward slices and hence satisfy the definition of a backward-slice cluster. These vertices also satisfy the definition of a forward-slice cluster as they are also in each others forward slices.

As dependence is not transitive, a statement can be in multiple slice-based clusters. For example, in Figure 2 the statements d and e are mutually dependent upon each other and thus satisfy the definition of a slice-based cluster. Statement d is also mutually dependent on statements a, b, c, thus the set {a, b, c, d} also satisfies the definition of a slice-based cluster. It can be shown that the clustering problem reduces to the NP-Hard *maximum clique* problem [23] making Definition 2 prohibitively expensive to implement.

2.3. Same-Slice Clusters

An alternative definition uses the *same-slice* relation in place of slice inclusion [10]. This relation replaces the need to check if two vertices are in each others slice with checking if two vertices have the *same* slice. The result is captured in the following definitions for *same-slice cluster*. The first uses backward slices and the second forward slices.

Definition 3 (Same-Slice MDS and Cluster [26])
A *same-backward-slice MDS* is a set of SDG vertices, V, such that
$\forall x, y \in V : \mathsf{BSlice}(x) = \mathsf{BSlice}(y)$.
A *same-backward-slice cluster* is a same-backward-slice MDS contained within no other same-backward-slice MDS.

A *same-forward-slice MDS* is a set of SDG vertices, V, such that
$\forall x, y \in V : \mathsf{FSlice}(x) = \mathsf{FSlice}(y)$.
A *same-forward-slice cluster* is a same-forward-slice MDS contained within no other same-forward-slice MDS.

Because $x \in \mathsf{BSlice}(x)$ and $x \in \mathsf{FSlice}(x)$, two vertices that have the same slice will always be in each other's slice. If slice inclusion were transitive, a backward-slice MDS (Definition 2) would be identical to a same-backward-slice MDS (Definition 3). However, as illustrated by the examples in Figure 1, slice inclusion is not transitive; thus, the relation is one of containment where every same-backward-slice MDS is also a backward-slice MDS but not necessarily a maximal one.

For example, in Figure 2 the set of vertices {a, b, c} form a same-backward-slice cluster because each vertex of the set yields the same backward slice. Whereas the set of vertices {a, c} form a same-forward-slice cluster as they have the same forward slice. Although vertex d is mutually dependent with all vertices of either set, it doesn't form the same-slice cluster with either set because it has additional dependence relationship with vertex e.

Although the introduction of same-slice clusters was motivated by the need for efficiency, the definition inadvertently introduced an *external* requirement on the cluster. Comparing the definitions for slice-based clusters (Definition 2) and same-slice clusters (Definition 3), a slice-based cluster includes only the *internal* requirement that the vertices of a cluster depend upon one another. However, a same-backward-slice cluster (inadvertently) adds to this internal requirement the *external* requirement that all vertices in the cluster are affected by the same vertices external to the cluster. Symmetrically, a same-forward-slice cluster adds the *external* requirement that all vertices in the cluster affect the same vertices external to the cluster.

2.4. Coherent Dependence Clusters

This subsection first formalizes the notion of *coherent dependence clusters* and then presents a slice-based instantiation of the definition. Coherent clusters

are dependence clusters that include not only an internal dependence requirement (each statement of a cluster depends on all the other statements of the cluster) but also an external dependence requirement. The external dependence requirement includes both that each statement of a cluster depends on the same statements external to the cluster and also that it influences the same set of statements external to the cluster. In other words, a coherent cluster is a set of statements that are mutually dependent and share identical extra-cluster dependence. Coherent clusters are defined in terms of the coherent MDS:

Definition 4 (Coherent MDS and Cluster [29])
A *coherent MDS* is a set of SDG vertices V, such that
$\forall x, y \in V$: x depends on a implies y depends on a and a depends on x implies a depends on y.
A *coherent cluster* is a coherent MDS contained within no other coherent MDS.

The slice-based instantiation of coherent cluster employs both backward *and* forward slices. The combination has the advantage that the entire cluster is both affected by the same set of vertices (as in the case of same-backward-slice clusters) and also affects the same set of vertices (as in the case of same-forward-slice clusters). The slice-based instantiation yields *coherent-slice clusters*:

Definition 5 (Coherent-Slice MDS and Cluster [29])
A *coherent-slice MDS* is a set of SDG vertices, V, such that
$\forall x, y \in V : \mathsf{BSlice}(x) = \mathsf{BSlice}(y) \wedge \mathsf{FSlice}(x) = \mathsf{FSlice}(y)$
A *coherent-slice cluster* is a coherent-slice MDS contained within no other coherent-slice MDS.

At first glance the use of both backward and forward slices might seem redundant because $x \in \mathsf{BSlice}(y) \Leftrightarrow y \in \mathsf{FSlice}(x)$. This is true up to a point: for the internal requirement of a coherent-slice cluster, the use of either BSlice or FSlice would suffice. However, the two are not redundant when it comes to the external requirements of a coherent-slice cluster. With a mutually-dependent cluster (Definition 1), it is possible for two vertices within the cluster to influence or be affected by different vertices *external* to the cluster. Neither is allowed with a coherent-slice cluster. To ensure both external effects are captured, both backward and forward slices are required.

In Figure 2 the set of vertices {a, c} form a coherent cluster as both these vertices have exactly the same backward and forward slices. That is, they share the identical intra- and extra- cluster dependencies. Coherent clusters are therefore a stricter from of same-slice clusters, all coherent clusters are also same-slice MDS but not necessarily maximal. It is worth noting that same-slice clusters partially share extra-cluster dependency. For example, each of the vertices in the same-backward-slice cluster {a, b, c} is dependent on the same set of external statements, but do not influence the same set of external statements.

2.5. Hashed based Coherent Slice Clusters

The computation of coherent-slice clusters (Definition 5) grows prohibitively expensive even for mid-sized programs where tens of gigabytes of memory are required to store the set of all possible backward and forward slices. The computation is cubic in time and quadratic in space. An approximation is employed to reduce the computation time and memory requirement. This approximation replaces comparison of slices with comparison of hash values, where hash values are used to summarize slice content. The result is the following approximation to coherent-slice clusters in which H denotes a hash function.

Definition 6 (Hash-Based Coherent-Slice MDS and Cluster [29])
A *hash-based coherent-slice MDS* is a set of SDG vertices, V, such that

$\forall x, y \in V : \mathsf{H}(\mathsf{BSlice}(x)) = \mathsf{H}(\mathsf{BSlice}(y)) \wedge \mathsf{H}(\mathsf{FSlice}(x)) = \mathsf{H}(\mathsf{FSlice}(y))$

A *hash-based coherent-slice cluster* is a hash-based coherent-slice MDS contained within no other hash-based coherent-slice MDS.

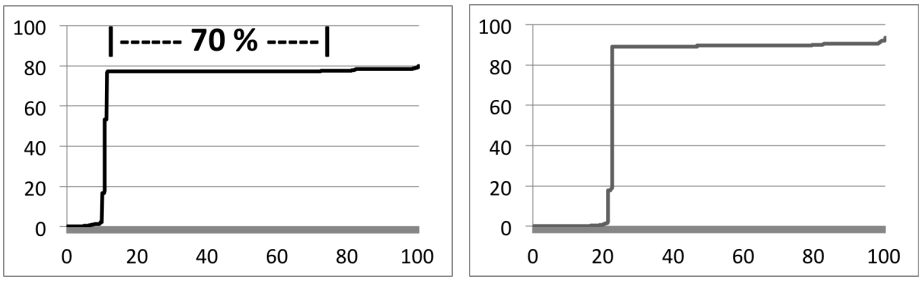
The precision of this approximation is empirically evaluated in Section 3.3. From here on, the paper considers only hash-based coherent-slice clusters unless explicitly stated otherwise. Thus, for ease of reading, hash-based coherent-slice cluster is referred to simply as *coherent cluster*.

2.6. Graph Based Cluster Visualization

This section describes two graph-based visualizations for dependence clusters. The first visualization, the *Monotone Slice-size Graph* (MSG) [10], plots a landscape of monotonically increasing slice sizes where the y-axis shows the size of each slice, as a percentage of the entire program, and the x-axis shows each slice, in monotonically increasing order of slice size. In an MSG, a dependence cluster appears as a sheer-drop cliff face followed by a plateau. The visualization assists with the inherently subjective task of deciding whether a cluster is large (how long is the plateau at the top of the cliff face relative to the surrounding landscape?) and whether it denotes a discontinuity in the dependence profile (how steep is the cliff face relative to the surrounding landscape?). An MSG drawn using backward slice sizes is referred to as a backward-slice MSG (B-MSG), and an MSG drawn using forward slice sizes is referred to as a forward-slice MSG (F-MSG).

As an example, the open source calculator `bc` contains 9,438 lines of code represented by 7,538 SDG vertices. The B-MSG for `bc`, shown in Figure 3a, contains a large plateau that spans almost 70% of the MSG. Under the assumption that same slice size implies the same slice, this indicates a large same-slice cluster. However, "zooming" in reveals that the cluster is actually composed of several smaller clusters made from slices of very similar size. The tolerance implicit in the visual resolution used to plot the MSG obscures this detail.

The second visualization, the *Slice/Cluster Size Graph* (SCG), alleviates this issue by combining both slice and cluster sizes. It plots three landscapes, one of increasing slice sizes, one of the corresponding same-slice cluster sizes, and the third of the corresponding coherent cluster sizes. In the SCG, vertices are



(a) B-MSG (b) F-MSG

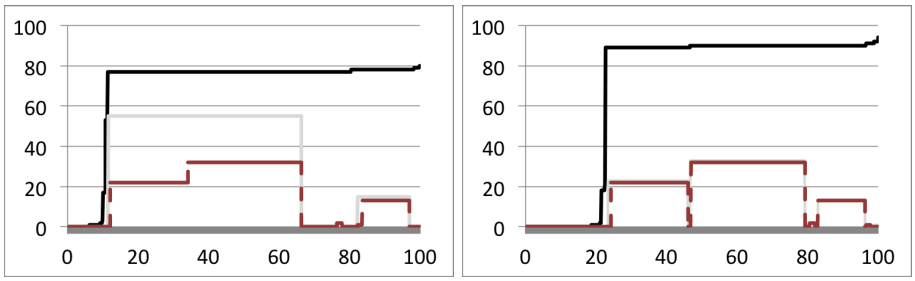
Figure 3: MSGs for the program bc.

ordered along the x-axis first according to their slice size, second according to their same-slice cluster size, and third according to the coherent cluster size. Three values are plotted on the y-axis: slice sizes form the first landscape, and cluster sizes form the second and third. Thus, SCGs not only show the sizes of the slices and the clusters, they also show the relation between them and thus bring to light interesting links. Two variants of the SCG are considered: the backward-slice SCG (B-SCG) is built from the sizes of backward slices, same-backward-slice clusters, and coherent clusters, while the forward-slice SCG (F-SCG) is built from the sizes of forward slices, same-forward-slice clusters, and coherent clusters. Note that both backward and forward SCGs use the same coherent cluster sizes.

The B-SCG and F-SCG for the program bc are shown in Figure 4. In both graphs the slice size landscape is plotted using a solid black line, the same-slice cluster size landscape using a gray line, and the coherent cluster size landscape using a (red) broken line. The B-SCG (Figure 4a) shows that bc contains two large same-backward-slice clusters consisting of almost 55% and almost 15% of the program. Surprisingly, the larger same-backward-slice cluster is composed of smaller slices than the smaller same-backward-slice cluster; thus, the smaller cluster has a bigger impact (slice size) than the larger cluster. In addition, the presence of three coherent clusters spanning approximately 15%, 20% and 30% of the program's statements can also be seen.

2.7. Cluster Visualization Tool

Cluster visualizations such as the SCG can provide an engineer a quick high-level overview of how difficult a program will be to work with [9]. High-level abstraction can cope with a tremendous amount of code (millions of lines) and reveal the high-level structure of a program. This overview can help an engineer form a mental model of a program's structure and consequently aid in tasks such as comprehension, maintenance, and reverse engineering. However, the high-level nature of the abstraction implies less detail. Furthermore, programmers are most comfortable in the spatial structure in which they read and write (i.e., that of source code). To accommodate the need for multiple levels of



(a) B-SCG (b) F-SCG

Figure 4: SCGs for the program bc

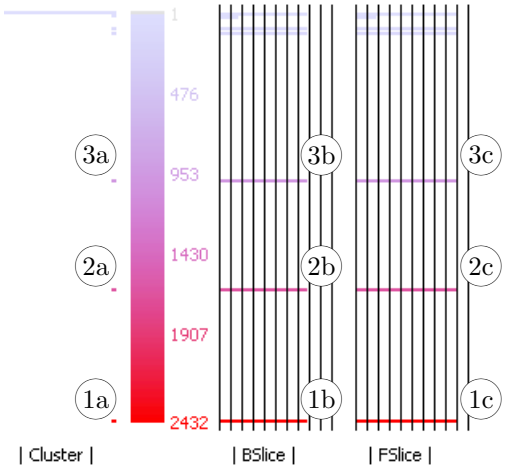


Figure 5: Heat Map View for bc

abstraction, the cluster visualization tool *decluvi* [28] provides four views: a *Heat Map* view and three different source-code views. The latter three include the *System View*, the *File View*, and the *Source View*, which allow a program's clusters to be viewed at increasing levels of detail. A common coloring scheme is used in all four views to help tie the different views together.

The *Heat Map* view aids an engineer in creating a mental model of the overall system. This overview can be traced to the source code using the other three views. The Heat Map provides a starting point that displays an overview of all the clusters using color to distinguish clusters of varying sizes. The view also displays additional statistics such as the size of the backward and forward slices for each coherent cluster and the number of clusters of each size. Figure 5 shows the Heat Map for bc, which is annotated for the purpose of this discussion. The three labels 1a, 1b, and 1c highlight statistics for the largest cluster (Cluster 1) of the program, whereas 2a, 2b, and 2c highlight statistics of the 2^{nd} largest cluster (Cluster 2) and the 3's the 3^{rd} largest cluster (Cluster 3). Starting from

the left of the Heat Map, using one pixel per cluster, horizontal lines (limited to 100 pixels) show the number of clusters that exist for each cluster size. This helps identify cases where there are multiple clusters of the same size. For example, the single dot next to the labels 1a, 2a and 3a depict that there is one cluster for each of the three largest sizes. A single occurrence is common for large clusters, but not for small clusters as illustrated by the long line at the top left of the Heat Map, which indicates multiple (uninteresting) clusters of size one.

To the right of the cluster counts is the actual Heat Map (color spectrum) showing cluster sizes from small to large reading from top to bottom using colors varying from blue to red. In gray scale this appear as shades of gray, with lighter shades (corresponding to blue) representing smaller clusters and darker shades (corresponding to red) representing larger clusters. Red is used for larger clusters as they are more likely to encompass complex functionality, making them more important "hot topics".

A numeric scale on the right of the Heat Map shows the cluster size (measured in SDG vertices). For program bc, the scale runs from 1–2432, depicting the sizes of the smallest cluster, displayed using light blue (light gray), and the largest cluster, displayed in bright red (dark gray).

Finally, on the right of the number scale, two slice size statistics are displayed: |BSlice| and |FSlice|, which show the sizes of the backward and forward slices for the vertices that form a coherent cluster. The sizes are shown as a percentage of the SDG's vertex count, with the separation of the vertical bars representing 10% increments. For example, Cluster 1's BSlice (1b) and FSlice (1c) include approximately 80% and 90% of the program's SDG vertices, respectively.

Turning to *decluvi*'s three source-code views, the *System View* is at the highest level of abstraction. Each file containing executable source code is abstracted into a column. For bc this yields the nine columns seen in Figure 6. The name of the file appears at the top of each column, color coded to reflect the size of the largest cluster found within the file. The vertical length of a column represents the length of the corresponding source file. To keep the view compact, each line of pixels in a column summarizes multiple source lines. For moderate sized systems, such as the case studies considered herein, each pixel line represents about eight source code lines. The color of each line reflects the largest cluster found among the summarized source lines, with light gray denoting source code that does not include any executable code. Finally, the numbers at the bottom of each column indicate the presence of the top 10 clusters in the file, where 1 denotes the largest cluster and 10 is the 10^{th} largest cluster. Although there may be other smaller clusters in a file, numbers are used to depict only the ten largest clusters because they are most likely to be of interest. In the case studies considered in Section 3, only the five largest coherent clusters are ever found to be interesting.

The *File View*, illustrated in Figure 7, is at a lower level of abstraction than the System View. It essentially zooms in on a single column of the System View. In this view, each pixel line corresponds to one line of source code. The pixel

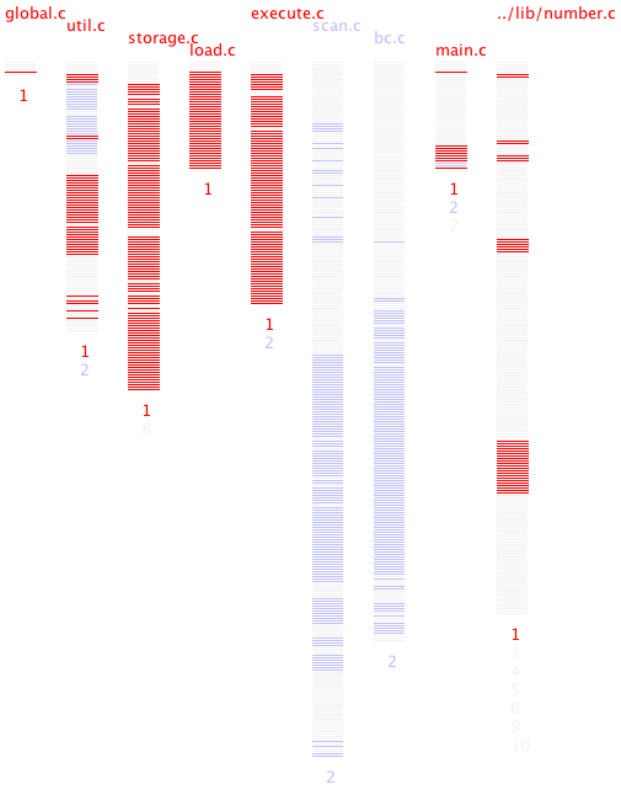


Figure 6: System View for the Program bc showing each file using one column and each line of pixels summarizing *eight* source lines. Blue color (medium gray in black & white) represent lines whose vertices are part of smaller size clusters than those in red color (dark gray), while lines not containing any executable lines are always shown in light gray.

Figure 7: File View for the file util.c of Program bc. Each line of pixels correspond to one source code line. Blue (medium gray in black & white) represents lines with vertices belonging to the 2^{nd} largest cluster and red (dark gray) represents lines with vertices belonging to the largest cluster. The rectangle marks function init_gen, part of both clusters.

lines are indented to mimic the indentation of the source lines and the number of pixels used to draw each line corresponds to the number of characters in the represented source code line. This makes it easier to relate this view to actual source code. The color of a pixel line depicts the size of the largest coherent cluster formed by the SDG vertices from the corresponding source code line. Figure 7 shows the File View of bc's file util.c, filtered to show only the two largest coherent clusters, while smaller clusters and non-executable lines are shown in light gray.

While the first two views aid in locating parts of the system involved in one or more clusters, the *Source View* allows a programmer to see the actual source code lines that makes up each cluster. This can be useful in addressing questions such as *Why is a cluster formed? What binds a cluster together?* and *Is there dependence pollution [10, 26]?* The *Source View*, illustrated in Figure 8, is a

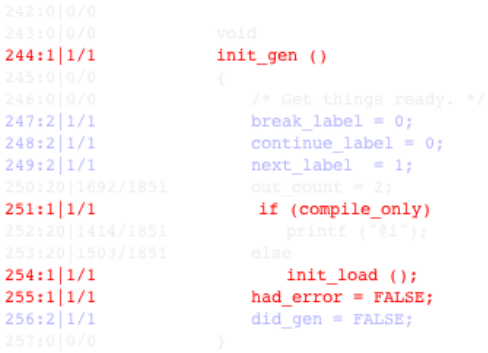


Figure 8: Source View showing function init_gen in file util.c of Program bc. The *decluvi* options are set to filter out all but the two largest clusters thus blue (medium gray in black & white) represents lines from the 2^{nd} largest cluster and red (dark gray) lines from the largest cluster. All other lines including those with no executable code are shown in light gray.

concrete view that maps the clusters onto actual source code lines. The lines are displayed in the same spatial context in which they were written, line color depicts the size of the largest cluster to which the SDG vertices representing the line belong. Figure 8 shows Lines 241–257 of bc's file util.c, which has again been filtered to show only the two largest coherent clusters. The lines of code whose corresponding SDG vertices are part of the largest cluster are shown in red (dark gray) and those lines whose SDG vertices are part of the second largest cluster are shown in blue (medium gray). Other lines that do not include any executable code or whose SDG vertices are not part of the two largest clusters are shown in light gray. On the left of each line is a *line tag* with the format $a:b|c/d$, which represents the line number (a), the cluster number (b), and an identification c/d for the c^{th} of d clusters having a given size. For example, in Figure 8, Lines 250 and 253 are both part of a 20^{th} largest cluster (clusters with same size have the same rank) as indicated by the value of b; however they belong to different clusters as indicated by the differing values of c in their line tags.

Decluvi has features such as *filtering* and *relative coloring*. These features help to isolate and focus on a set of clusters of interest. Filtering allows a range of cluster sizes of interest to be defined. Only clusters whose size falls within the filtered range are shown using the Heat Map colors. Those outside the specified range along with non-executable lines of code are shown in light gray where in grayscale they appear in the lightest shade of gray. The filtering system incorporates a feature to *hide* non-executable lines of code as well as clusters whose size falls outside the specified range. In addition, relative coloring allows the Heat Map colors to be automatically adjusted to fit within a defined cluster size range. Relative coloring along with filtering overcomes the problem where clusters of similar sizes are represented using similar colors, making them

indistinguishable.

3. Empirical Evaluation

This section presents the empirical evaluation into the existence and impact of coherent dependence clusters. The section first discusses the experimental setup and the subject programs included in the study. It then presents two validation studies, the first on the use of CodeSurfer's pointer analysis, and, the second on the use of hashing to summarize slices for efficient cluster identification. The section then quantitatively considers the existence of coherent dependence clusters and identifies patterns of clustering within the programs. This is followed by a series of four case studies, where qualitative analysis, aided by *decluvi*, is used to highlight how knowledge of clusters can aid a software engineer. Finally, inter-cluster dependence and threats to validity are considered. More formally, the empirical evaluation addresses the following research questions:

RQ1 Does increased pointer analysis precision result in smaller coherent clusters?

RQ2 Does hashing provide a sufficient summary of a slice to allow comparing hash values to replace comparing slices?

RQ3 Do large coherent clusters exist in production source code?

RQ4 Which patterns of clustering can be identified?

RQ5 Can analysis of coherent clusters reveal structures within a program?

RQ6 Does dependence between coherent clusters induce larger dependence structures?

The first two research questions (*RQ1* and *RQ2*) provide empirical verification for the results subsequently presented. *RQ1* establishes the impact of the data flow analysis quality used by the slicing tool. Whereas *RQ2* is the foundation for the remaining empirical study because it establishes that the hash function for approximating slice content is sufficiently precise. If the static slices produced by the slicer are overly conservative or if the slice approximation is not sufficiently precise, then the results presented will not be reliable. Fortunately, the results provide confidence that the slice precision and hashing accuracy are sufficient.

Whereas *RQ1* and *RQ2* focus on the veracity of our approach, *RQ3* investigates the validity of the study; if large coherent clusters are not prevalent, then they would not be worthy of further study. We place very specific and demanding constraints on a set of vertices for it to be deemed a coherent cluster. If such clusters are not common then their study would be merely an academic exercise. Our findings reveal that, despite the tight constraints inherent in the definition of a coherent dependence cluster, they are, indeed, very common.

Program	C Files	LoC	SLoC	ELoC	SDG vertex count	Total Slices	Largest Coherent Cluster Size	Description
a2ps	79	46,620	22,117	18,799	224,413	97,170	8%	ASCII to Postscript
acct	7	2,600	1,558	642	7,618	2,834	11%	Process monitoring
acm	114	32,231	21,715	15,022	159,830	63,014	43%	Flight simulator
anubis	35	18,049	11,994	6,947	112,282	34,618	13%	SMTP messenger
archimedes	1	787	575	454	20,136	2,176	4%	Semiconductor device simulator
barcode	13	3,968	2,685	2,177	16,721	9,602	58%	Barcode generator
bc	9	9,438	5,450	4,535	36,981	15,076	32%	Calculator
byacc	12	6,373	5,312	4,688	45,338	16,590	7%	Parser generator
cflow	25	12,542	7,121	5,762	68,782	24,638	8%	Control flow analyzer
combine	14	8,202	6,624	5,279	49,288	29,118	15%	File combinator
copia	1	1,168	1,111	1,070	42,435	6,654	48%	ESA signal processing code
cppi	13	6,261	1,950	2,554	17,771	10,280	13%	C preprocessor formatter
ctags	33	14,663	11,345	7,383	152,825	31,860	48%	C tagging
diction	5	2,218	1,613	427	5,919	2,444	16%	Grammar checker
diffutils	23	8,801	6,035	3,638	30,023	16,122	44%	File differencing
ed	8	2,860	2,261	1,788	35,475	11,376	55%	Line text editor
enscript	22	14,182	10,681	9,135	67,405	33,780	19%	File converter
findutils	59	24,102	13,940	9,431	102,910	41,462	22%	Line text editor
flex	21	23,173	12,792	13,537	89,806	37,748	16%	Lexical Analyzer
garpd	1	669	509	300	5,452	1,496	14%	Address resolved
gcal	30	62,345	46,827	37,497	860,476	286,000	62%	Calendar program
gnuedma	1	643	463	306	5,223	1,488	44%	Development environment
gnushogi	16	16,301	11,664	7,175	64,482	31,298	40%	Japanese chess
indent	8	6,978	5,090	4,285	24,109	7,543	52%	Text formatter
less	33	22,661	15,207	9,759	451,870	33,558	35%	Text reader
spell	1	741	539	391	6,232	1,740	20%	Spell checker
time	6	2,030	1,229	433	4,946	3,352	4%	CPU resource measure
userv	2	1,378	1,112	1,022	15,418	5,362	9%	Access control
wdiff	4	1,652	1,108	694	10,077	2,722	6%	Diff front end
which	6	3,003	1,996	753	8,830	3,804	35%	Unix utility
Average	20	11,888	7,754	5,863	91,436	28,831	27%	

Table 1: Subject programs

These results motivate the remaining research questions. Having demonstrated that our technique is suitable for finding coherent clusters and that such clusters are sufficiently widespread to be worthy of study, we investigate specific coherent clusters in detail. *RQ4* asks whether there are common patterns of clustering in the programs studied and *RQ5* asks whether these clusters reveal aspects of the underlying logical structure of programs. Finally, *RQ6* looks explicitly at inter-cluster dependency relationships and considers areas of software engineering where they may be of interest.

3.1. Experimental Subjects and Setup

The study considers the 30 C programs shown in Table 1, which provides a brief description of each program alongside six measures: number of files containing executable C code, LoC – lines of code (as counted by the Unix utility wc), SLoC – the non-comment non-blank lines of code (as counted by the utility sloccount [43]), ELoC – the number of source code lines that *CodeSurfer* [4] considers to contain executable code, the number of SDG vertices, the number of slices produced, and finally the size (as a percentage of the program's SDG vertex count) of the largest coherent cluster. The values under ELoC are smaller

than the other source code measures because they reflect SLoC for the particular preprocessed version of the program considered by CodeSurfer.

The data and visualizations presented in this paper are generated from slices taken with respect to *source-code representing* SDG vertices. This excludes pseudo vertices introduced into the SDG, to represent, for example, global variables, which are modeled as additional pseudo parameters by CodeSurfer. Thus in Table 1 total slices is smaller than the SDG vertex count. Cluster sizes are also measured in terms of source-code representing SDG vertices, which is more consistent than using lines of code as it is not influenced by blank lines, comments, statements spanning multiple lines, multiple statements on one line, or compound statements.

The slices along with the mapping between the SDG vertices and the actual source code is extracted from the mature and widely used slicing tool CodeSurfer (version 2.1). The cluster visualizations were generated by *decluvi* using data extracted from CodeSurfer. The *decluvi* system along with scheme scripts for data acquisition and pre-compiled datasets for several open-source programs can be downloaded from `http://www.cs.ucl.ac.uk/staff/s.islam/decluvi.html`

3.2. CodeSurfer Pointer Analysis

Recall that the definition of a coherent dependence cluster is based on an underlying *depends-on* relation, which is approximated using program slicing. Pointer analysis plays a key role in the precision of slicing. This section presents a study on the effect of various levels of pointer-analysis precision on the size of the coherent clusters. It addresses research question *RQ1* by considering whether more sophisticated pointer analysis results in more precise slices and hence smaller clusters. There is no automatic way to determine whether the slices are correct and precise, we use slice size as a measure of conciseness and thus precision.

CodeSurfer provides three levels of points-to analysis precision (Low, Medium, and High) that provide increasingly precise points-to information at the expense of additional memory and analysis time [24]. The Low setting uses minimal pointer analysis that assumes every pointer may point to every object that has its address taken (variable or function). At the Medium and High settings, CodeSurfer performs extensive pointer analysis using the algorithm proposed by Fahndrich et al. [20], which implements a variant of Andersen's points-to algorithm [3] (this includes parameter aliasing). At the Medium setting, fields of a structure are not distinguished while the High level distinguishes structure fields. The High setting should produce the most precise slices but requires more memory and time during SDG construction, which puts a functional limit on the size and complexity of the programs that can be handled by CodeSurfer [24].

The study compares slice and cluster size for CodeSurfer's three precision options (Low, Medium, High) to study the impact of points-to precision. The results of the study are shown in Table 2. Column 1 lists the programs, while columns 2–4, 5–7, 8–10, and 11–13 present the average slice size, maximum slice size, average cluster size, and maximum cluster size, respectively, for each each of the three precision settings. The results for average slice size deviation and

Program	Average Slice Size			Max Slice Size			Average Cluster Size			MaxCluster Size		
	L	M	H	L	M	H	L	M	H	L	M	H
a2ps	25223	23085	20897	45231	44139	43987	2249	1705	711	10728	9295	4002
acct	763	700	621	1357	1357	1357	79	66	40	272	236	162
acm	19083	17997	16509	29403	28620	28359	3566	3408	4197	9356	9179	10809
anubis	11120	10806	9085	16548	16347	16034	939	917	650	2708	2612	2278
archimedes	113	113	113	962	962	962	3	3	3	39	39	39
barcode	3523	3052	2820	4621	4621	4621	1316	1870	1605	2463	2970	2793
bc	5278	5245	5238	7059	7059	7059	1185	1188	1223	2381	2384	2432
byacc	3087	2936	2886	9036	9036	9036	110	110	103	583	583	567
cflow	7314	5998	5674	11856	11650	11626	865	565	246	3060	2191	1097
combine	3512	3347	3316	13448	13448	13448	578	572	533	2252	2252	2161
copia	1844	1591	1591	3273	3273	3273	1566	1331	1331	1861	1607	1607
cppi	1509	1352	1337	4158	4158	4158	196	139	139	825	663	663
ctags	12681	11663	11158	15483	15475	15475	7917	4199	3955	11080	7905	7642
diction	421	392	387	1194	1194	1194	46	37	37	217	196	196
diffutils	5049	4546	4472	7777	7777	7777	3048	1795	1755	4963	3596	3518
ed	4203	3909	3908	5591	5591	5591	2099	1952	1952	3281	3146	3146
enscript	7023	6729	6654	16130	16130	16130	543	554	539	3140	3242	3243
findutils	7020	6767	5239	11075	11050	11050	1969	1927	1306	4489	4429	2936
flex	9038	8737	8630	17257	17257	17257	622	657	647	3064	3064	3064
garpd	284	242	224	628	628	628	32	31	29	103	103	103
gcal	132860	123438	123427	142739	142289	142289	40885	40614	40614	93541	88532	88532
gnuedma	385	369	368	730	730	368	178	176	174	333	331	330
gnushogi	9569	9248	9141	14726	14726	14726	1577	2857	2820	3787	6225	6179
indent	4104	4058	4045	5704	5704	5704	2036	2032	1985	3402	3399	3365
less	13592	13416	13392	16063	16063	16063	4573	3074	3035	7945	5809	5796
spell	359	293	291	845	845	845	58	31	48	199	128	174
time	201	161	158	730	730	730	4	3	3	35	33	33
userv	1324	972	964	2721	2662	2662	69	32	53	268	154	240
wdiff	687	582	561	2687	2687	2687	33	21	19	184	158	158
which	1080	1076	1070	1744	1744	1744	413	413	410	798	798	793

Table 2: CodeSurfer pointer analysis settings

largest cluster size deviation are visualized in Figures 9 and 10. The graphs use the High setting as the base line and show the percentage deviation when using the Low and Medium settings.

Figure 9 shows the average slice size deviation when using the lower two settings compared to the highest. On average, the Low setting produces slices that are 14% larger than the High setting. Program userv has the largest deviation of 37% when using the Low setting. For example, in userv the minimal pointer analysis fails to recognize that the function pointer oip can never point to functions sighandler_alrm and sighandler_child and includes them as called functions at call sites using *oip, increasing slice size significantly. In all 30 programs, the Low setting yields larger slices compared to the High setting.

The Medium setting always yields smaller slices when compared to the Low setting. For eight programs, the Medium setting produces the same average slice size as the High setting. For the remaining programs the Medium setting produces slices that are on average 4% larger than when using the High setting. The difference in slice size occurs because the Medium setting does not differentiate between structure fields, which the High setting does. The largest deviation is seen in findutils at 34%. With the Medium setting, the structure

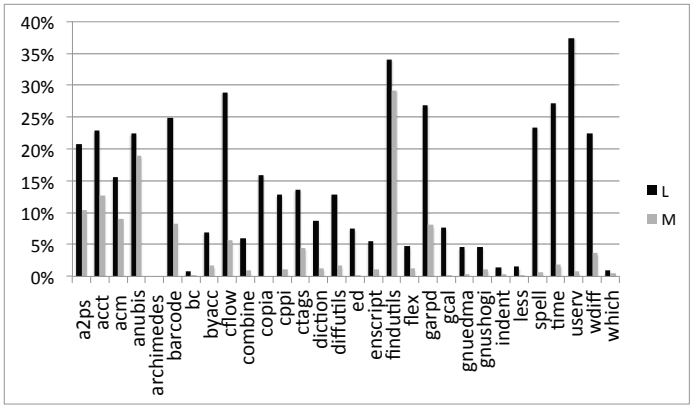


Figure 9: Percentage deviation of average slice size for Low and Medium CodeSurfer pointer analysis settings

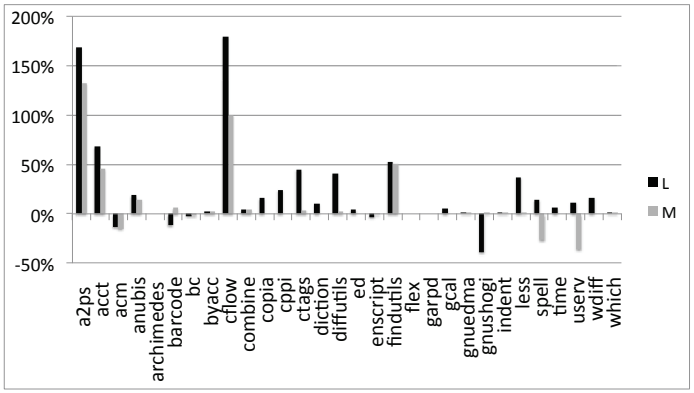
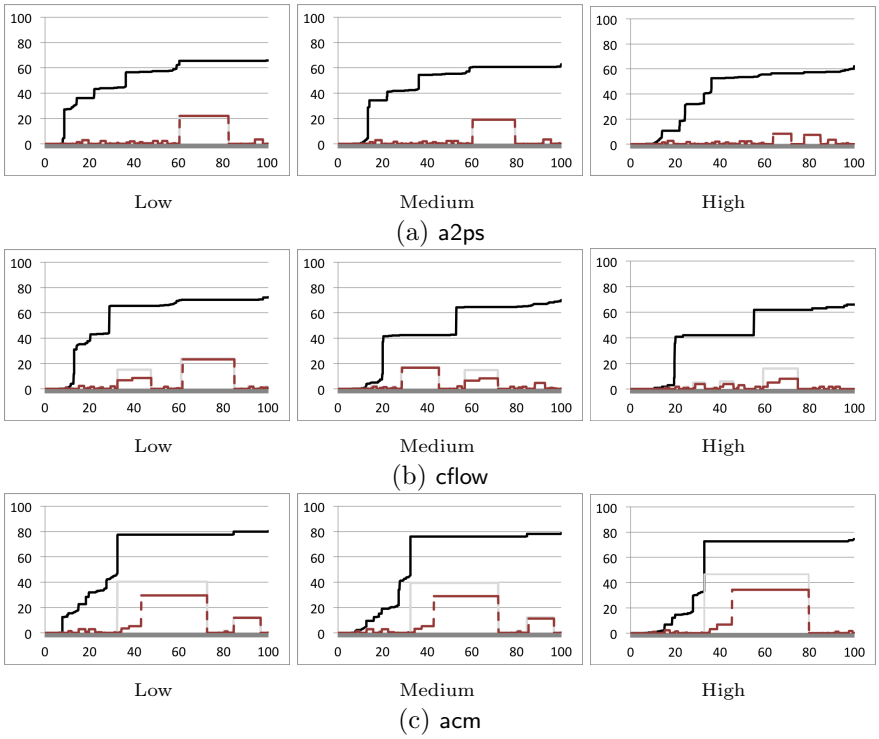


Figure 10: Percentage deviation of largest cluster size for Low and Medium CodeSurfer pointer analysis settings

fields (options, regex_map, stat_buf and state) of findutils are lumped together as if each structure were a scalar variable, resulting in larger, less precise, slices.

Figure 10 visualizes the deviation of the largest coherent cluster size when using the lower two settings compared to the highest. The graph shows that the size of the largest coherent clusters found when using the lower settings is larger in most of the programs. On average there is a 22% increase in the size of the largest coherent cluster when using the Low setting and a 10% increase when using the Medium setting. In a2ps and cflow the size of the largest cluster increases over 100% when using the Medium setting and over 150% when using the Low setting. The increase in slice size is expected to result in larger clusters due to the loss of precision.

The B-SCGs for a2ps for the three settings is shown in Figure 11a. In the



(a) a2ps

(b) cflow

(c) acm

Figure 11: SCGs for Low, Medium and High pointer settings of CodeSurfer

graphs it is seen that the slice sizes get smaller and have increased steps in the (black) landscape indicating that they become more precise. The red landscape shows that there is a large coherent cluster detected when using the Low setting running from approx. 60% to 85% on the x-axis. This cluster drops in size when using the Medium setting. At the High setting this coherent clusters breaks up into multiple smaller clusters. In this case, a drop in the cluster size also leads to breaking of the cluster in to multiple smaller clusters.

In the SCGs for cflow (Figure 11b) a similar drop in the slice size and cluster size is observed. However, unlike a2ps the large coherent cluster does not split into smaller clusters but only drops in size. The largest cluster when using the Low setting runs from 60% to 85% on the x-axis. This cluster reduces in size and shifts position running 30% to 45% x-axis when using the Medium setting. The cluster further drops in size down to 5% running 25% to 30% on the x-axis when using the High setting. In this case the largest cluster has a significant drop in size but doesn't split into multiple smaller clusters.

```
f6(x) {
  f = *p(42, 4);
  return f;
}
```

Figure 12: replacement coherent cluster example

Surprisingly, Figure 10 also shows seven programs where the largest coherent cluster size actually increases when using the highest pointer analysis setting on CodeSurfer. Figure 11c shows the B-SCGs for acm which falls in this category. This counter-intuitive result is seen only when the more precise analysis determines that certain functions cannot be called and thus excludes them from the slice. Although in all such instances slices get smaller, the clusters may grow if the smaller slices match other slices already forming a cluster.

For example, consider replacing function f6 in Figure 1 with the code shown in Figure 12, where f depends on a function call to a function referenced through the function pointer p. Assume that the highest precision pointer analysis determines that p does not point to f2 and therefore there is no call to f2 or any other function from f6. The higher precision analysis would therefore determine that the forward slices and backward slices of a, b and c are equal, hence grouping these three vertices in a coherent cluster. Whereas the lower precision is unable to determine that p cannot point to f2, the backward slice on f will conservatively include b. This will lead the higher precision analysis to determine that the set of vertices {a, b, c} are one coherent cluster whereas the lower precision analysis include only set of vertices {a, c} in the same coherent cluster.

As an answer to *RQ1*, we find that in 85% of cases the Medium and Low settings result in larger coherent clusters when compared to the High setting. For the remaining cases we have identified valid scenarios where more precise pointer analysis can result in larger coherent clusters. The results also confirm that a more precise pointer analysis leads to more precise (smaller) slices. Be-

cause it gives the most precise slices and most accurate clusters, the remainder of the paper uses the highest CodeSurfer pointer analysis setting.

3.3. Validity of the Hash Function

This section addresses research question *RQ2: Does hashing provide a sufficient summary of a slice to allow comparing hash values to replace comparing slices?* The section validates the use of comparing slice hash values in lieu of comparing actual slice content. The use of hash values to represent slices reduce both the memory requirement and run-time, as it is no longer necessary to store or compare entire slices. The hash function, denoted H in Definition 6, determines a hash value for a slice based on the unique vertex ids assigned by CodeSurfer. Validation of this approach is needed to confirm that the hash values provide a sufficiently accurate summary of slices to support the correct partitioning of SDG vertices into coherent clusters. Ideally, the hash function would produce a unique hash value for each distinct slice. The validation study aims to find the number of unique slices for which the hash function successfully produces an unique hash value.

For the validation study we chose 16 programs from the set of 30 subject programs. The largest programs were not included in the validation study to make the study-time manageable. Results are based on both the backward and forward slices for every vertex of these 16 programs. To present the notion of precision we introduce the following formalization. Let V be the set of all source-code representing SDG vertices for a given program P and US denote the number of *unique slices*: $US = |\{\mathsf{BSlice}(x) : x \in V\}| + |\{\mathsf{FSlice}(x) : x \in V\}|$. Note that if all vertices have the same backward slice then $\{\mathsf{BSlice}(x) : x \in V\}$ is a singleton set. Finally, let UH be the number of *unique hash-values*, $UH = |\{\mathsf{H}(\mathsf{BSlice}(x)) : x \in V\}| + |\{\mathsf{H}(\mathsf{FSlice}(x)) : x \in V\}|$.

The accuracy of hash function H is given as Hashed Slice Precision, $HSP = UH/US$. A precision of 1.00 ($US = UH$) means the hash function is 100% accurate (i.e., it produces a unique hash value for every distinct slice) whereas a precision of $1/US$ means that the hash function produces the same hash value for every slice leaving $UH = 1$.

Table 3 summarizes the results. The first column shows each program. The second and the third columns report the values of US and UH respectively. The fourth column reports HSP, the precision attained using hash values to compare slices. Considering all 78,587 unique slices the hash function produced unique hash values for 74,575 of them, resulting in an average precision of 94.97%. In other words, the hash function fails to produce unique hash values for just over 5% of the slices. Considering the precision of individual programs, five of the programs have a precision greater than 97%, while the lowest precision, for findutils, is 92.37%. This is, however, a significant improvement over previous use of slice size as the hash value, which is only 78.3% accurate [10].

Coherent cluster identification uses two hash values for each vertex (one for the backward slice and other for the forward slice) and the slice sizes. Slice size matching filters out some instances where the hash values happen to be the same by coincidence but the slices are different. The likelihood of both

Program	Unique Slices (US)	Unique Hash values (UH)	Hashed Slice Precision (HSP)	Cluster Count (CC)	Hash Cluster Count (HCC)	Hash Precision Clusters (HCP)
acct	1,558	1,521	97.63%	811	811	100.00%
barcode	2,966	2,792	94.13%	1,504	1,504	100.00%
bc	3,787	3,671	96.94%	1,955	1,942	99.34%
byacc	10,659	10,111	94.86%	5,377	5,377	100.00%
cflow	16,584	15,749	94.97%	8,457	8,452	99.94%
copia	3,496	3,398	97.20%	1,785	1,784	99.94%
ctags	8,739	8,573	98.10%	4,471	4,470	99.98%
diffutils	5,811	5,415	93.19%	2,980	2,978	99.93%
ed	2,719	2,581	94.92%	1,392	1,390	99.86%
findutils	9,455	8,734	92.37%	4,816	4,802	99.71%
garpd	808	769	95.17%	413	411	99.52%
indent	3,639	3,491	95.93%	1,871	1,868	99.84%
time	1,453	1,363	93.81%	760	758	99.74%
userv	3,510	3,275	93.30%	1,827	1,786	97.76%
wdiff	2,190	2,148	98.08%	1,131	1,131	100.00%
which	1,213	1,184	97.61%	619	619	100.00%
Sum	78,587	74,575	–	40,169	40,083	–
Average	4,912	4,661	94.97%	2,511	2,505	99.72%

Table 3: Hash function validation

hash values matching those from another vertex with different slices is less than that of a single collision. Extending US and UH to clusters, Columns 5 and 6 (Table 3) report CC, the number of coherent clusters in a program and HCC, the number of coherent clusters found using hashing. The final column shows the precision attained using hashing to identify clusters, $HCP = HCC/CC$. The results show that of the 40,169 coherent clusters, 40,083 are uniquely identified using hashing, which yields a precision of 99.72%. Five of the programs show total agreement, furthermore for each program HCP is over 99%, except for userv, which has the lowest precision of 97.76%. This can be attributed to the large percentage (96%) of single vertex clusters in userv. The hash values for slices taken with respect to these single-vertex clusters have a higher potential for collision leading to a reduction in overall precision. In summary, this study provides an affirmative answer to *RQ2*. The hash-based approximation is sufficiently accurate. Comparing hash values can replace the need to compare actual slices.

3.4. Do large coherent clusters occur in practice?

Having demonstrated that hash function H can be used to effectively approximate slice contents, this section considers the validation research question, *RQ3: Do large coherent clusters exist in production source code?* The question is first answered quantitatively using the size of the largest coherent cluster in each program and then through visual analysis of the SCGs.

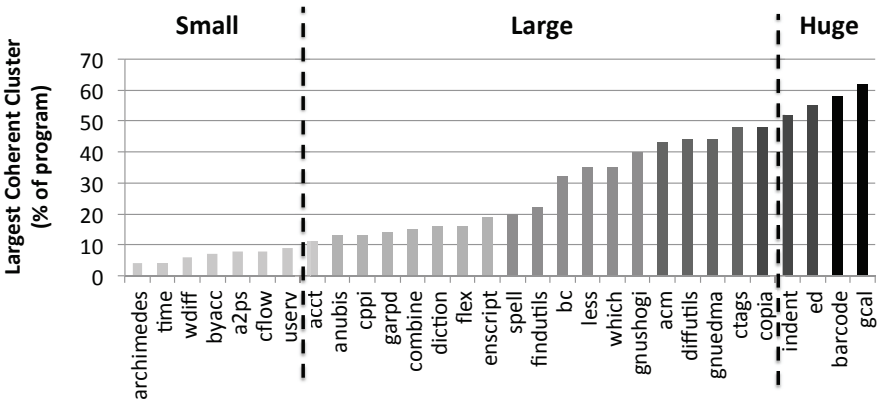


Figure 13: Size of largest coherent cluster

To assess if a program includes a *large* coherent cluster, requires making a judgement concerning what threshold constitutes large. Following prior empirical work [10, 26, 28, 29], a threshold of 10% is used. In other words, a program is said to contain a large coherent cluster if 10% of the program's SDG vertices produce the same backward slice as well as the same forward slice.

Figure 13 shows the size of the largest coherent cluster found in each of the 30 subject programs. The programs are divided into 3 groups based on the size of the largest cluster present in the program.

Small: *Small* consists of seven programs none of which have a coherent cluster constituting over 10% of the program vertices. These programs are archimedes, time, wdiff, byacc, a2ps, cflow and userv. Although it may be interesting to study why large clusters are not present in these programs, this paper focuses on studying the existence and implications of large coherent clusters.

Large: This group consists of programs that have at least one cluster with size 10% or larger. As there are programs containing much larger coherent clusters, a program is placed in this group if it has a large cluster between the size 10% and 50%. Over two-thirds of the programs studied fall in this category.

The program at the bottom of this group (acct) has a coherent cluster of size 11% and the largest program in this group (copia) has a coherent cluster of size 48%. We present both these programs as case studies and discuss their clustering in detail in Sections 3.6.1 and 3.6.4, respectively. The program bc which has multiple large clusters with the largest of size 32% falls in the middle of this group and is also presented as a case study in Section 3.6.3.

Huge: The final group consists of programs that have a large coherent cluster

whose size is over 50%. Out of the 30 programs 4 fall in this group. These programs are indent, ed, barcode and gcal. From this group, we present indent as a case study in Section 3.6.2.

In summary all but 7 of the 30 subject programs contain a large coherent cluster. Therefore, over 75% of the subject programs contain a coherent cluster of size 10% or more. Furthermore, half the programs contain a coherent cluster of at least 20% in size. It is also interesting to note that although this grouping is based only on the largest cluster, many of the programs contain multiple large coherent clusters. For example, ed, ctags, nano, less, bc, findutils, flex and garpd all have multiple large coherent clusters. It is also interesting to note that there is no correlation between a program's size (measured in SLoC) and the size of its largest coherent cluster. For example, in Table 1 two programs of very different sizes, cflow and userv, have similar largest-cluster sizes of 8% and 9%, respectively. Whereas programs acct and ed, of similar size, have very different largest coherent clusters of sizes 11% and 55%.

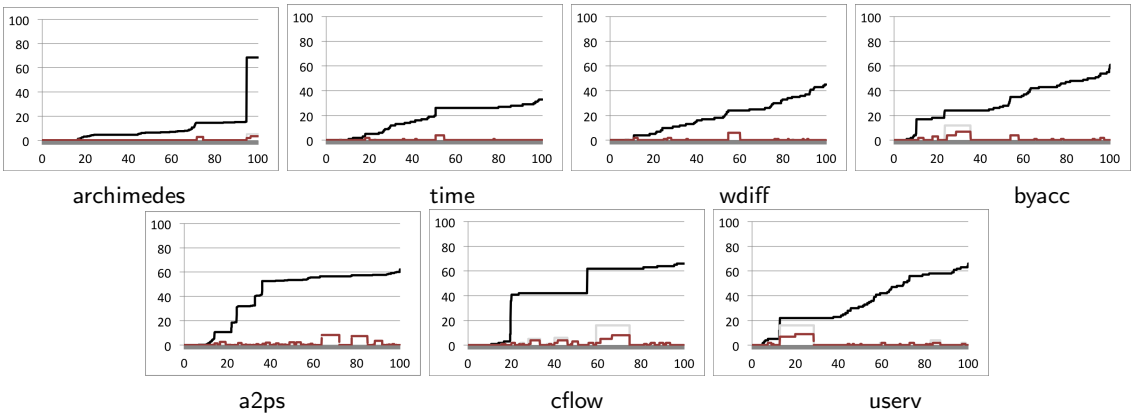


Figure 14: Programs with *small* coherent clusters

Therefore as an affirmative answer to *RQ3*, the study finds that 23 of the 30 programs studied have a large coherent cluster. Some programs also have a huge cluster covering over 50% of the program vertices. Furthermore, the choice of 10% as a threshold for classifying a cluster as large is a relatively conservative choice. Thus, the results presented in this section can be thought of as a lower bound to the existence question.

3.5. Patterns of clustering

This section presents a visual study of SCGs for the three program groups and addresses *RQ4* by identifying patterns of clustering common among the groups. Figures 14–16 show graphs for the three categories. The graphs in the figures are laid out in ascending order based on the largest coherent cluster present in the program and thus follow the same order as seen in Figure 13.

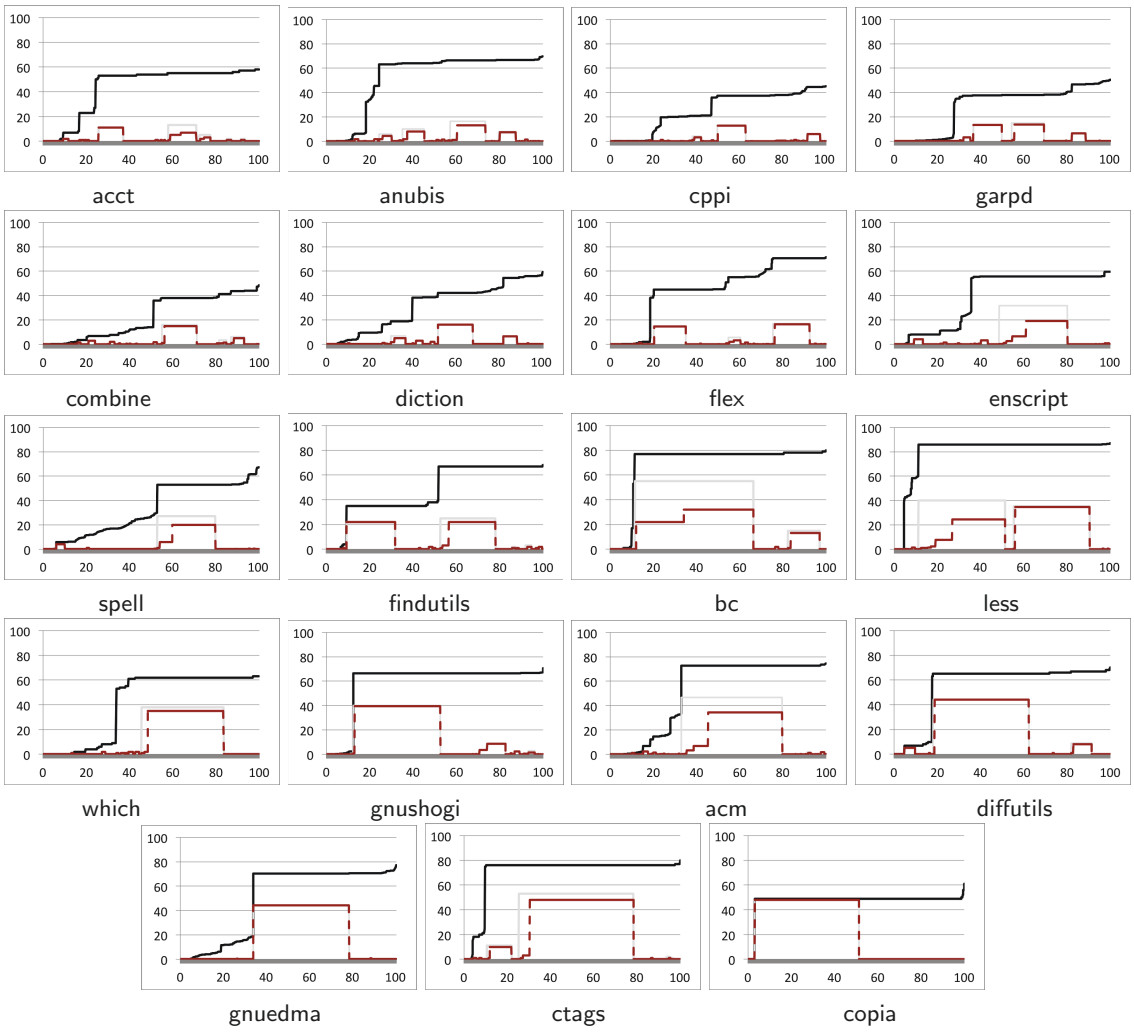


Figure 15: Programs with *large* coherent clusters

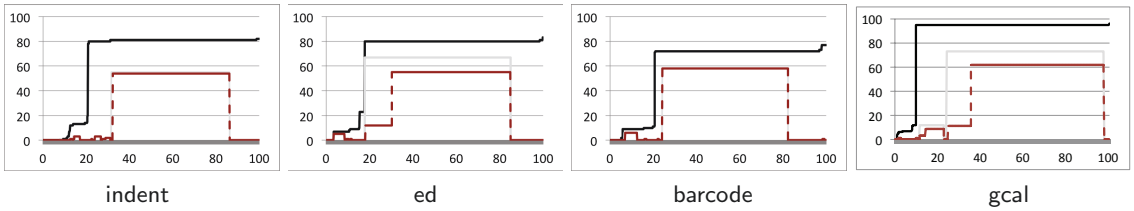


Figure 16: Programs with *huge* coherent clusters

Figure 14 shows SCGs for the seven programs of the *small* group. In the SCGs of the first three programs (archimedes, time and wdiff) only a small coherent cluster is visible in the red landscape. In the remaining four programs, the red landscape shows the presence of multiple small coherent clusters. It is very likely that, similar to the results of the case studies presented later, these clusters also depict logical constructs within each program.

Figure 15 shows SCGs of the 19 programs that have at least one large, but not huge, coherent cluster. That is, each program has at least one coherent cluster covering 10% to 50% of the program. Most of the programs have multiple coherent clusters as is visible on the red landscape. Some of these have only one large cluster satisfying the definition of *large*, such as acct. The clustering of acct is discussed in further detail in Section 3.6.1. Most of the remaining programs are seen to have multiple large clusters such as bc, which is also discussed in further detail in Section 3.6.3. The presence of multiple large coherent cluster hints that the program consists of multiple functional components. In three of the programs (which, gnuedma and copia) the landscape is completely dominated by a single large coherent cluster. In which and gnuedma this cluster covers around 40% of the program vertices whereas in copia the cluster covers 50%. The presence of a single large dominating cluster points to a centralized functionality or structure being present in the program. Copia is presented as a case study in Section 3.6.4 where its clustering is discussed in further detail.

Finally, SCGs for the four programs that contain *huge* coherent clusters (covering over 50%) are found in Figure 16. In all four landscapes there is a very large dominating cluster with other smaller clusters also being visible. This pattern supports the conjecture that the program has one central structure or functionality which consists of most of the program elements, but also has additional logical constructs that work in support of the central idea. Indent is one program that falls in this category and is discussed in further detail in Section 3.6.2.

As an answer to *RQ4*, the study finds that most programs contain multiple coherent clusters. Furthermore, the visual study reveals that a third of the programs have multiple large coherent clusters. Only three programs copia, gnuedma, and which show the presence of only a single (overwhelming) cluster.

Having shown that coherent clusters are prevalent in programs and that most programs have multiple significant clusters, it is our conjecture that these clusters represent high-level functional components of programs and hence represent the systems logical structure. The following sections present a series of four case studies where such mappings are discussed in details.

3.6. Coherent Cluster and program decomposition

This section presents four case studies using acct, indent, bc and copia. The case studies form a major contribution of the paper and collectively address research question *RQ5: Can analysis of coherent clusters reveal structures within a program?* The programs have been chosen to represent the *large* and *huge* groups identified in the previous section. Three programs are taken from the *large* group as it consists of the majority of the programs and one from the *huge*

group. Each of the three programs from the *large* group were chosen because it exhibits specific patterns. Acct has multiple coherent clusters visible in its profile and has the smallest large cluster in the group, bc has multiple large coherent clusters, and copia has only a single large coherent cluster dominating the entire landscape.

3.6.1. Case Study: acct

The first of the series of case studies is acct, an open-source program used for monitoring and printing statistics about users and processes. The program acct is one of the smaller programs with 2,600 LoC and 1,558 SLoC from which CodeSurfer produced 2,834 slices. The program has seven C files, two of which, getopt.c and getopt1.c, contain only conditionally included functions. These functions provide support for command-line argument processing and are included if needed library code is missing.

Table 4 shows the statistics for the five largest clusters of acct. Column 1 gives the cluster number, where 1 is the largest and 5 is the 5^{th} largest cluster measured using the number of vertices. Columns 2 and 3 show the size of the cluster as a percentage of the program's vertices and actual vertex count, as well as the line count. Columns 4 and 5 show the number of files and functions where the cluster is found. The cluster sizes range from 11.4% to 2.4%. These five clusters can be readily identified in the Heat-Map visualization (not shown) of *decluvi*. The rest of the clusters are very small (less than 2% or 30 vertices) in size and are thus of little interest.

Cluster	Cluster Size		Files	Functions
	%	vertices/lines	spanned	spanned
1	11.4%	162/88	4	6
2	7.2%	102/56	1	2
3	4.9%	69/30	3	4
4	2.8%	40/23	2	3
5	2.4%	34/25	1	1

Table 4: acct's top five clusters

The B-SCG for acct (row one of Figure 15) shows the existence of these five coherent clusters along with other same-slice clusters. *Splitting* of the same-slice cluster is evident in the SCG. Splitting occurs when the vertices of a same-slice cluster become part of different coherent clusters. This happens when vertices have either the same backward slice or the same forward slice but not both. In acct's B-SCG the vertices of the largest same-backward-slice cluster spanning the x-axis from 60% to 75% are not part of the same coherent cluster. This is because the vertices do not share the same forward slice. This splitting effect is common among the programs studied.

Decluvi visualization (not shown) of acct reveals that the largest cluster spans four files (file_rd.c, common.c, ac.c, and utmp_rd.c), the 2^{nd} largest cluster spans only a single file (hashtab.c), the 3^{rd} largest cluster spans three files (file_rd.c,

ac.c, and hashtab.c), the 4^{th} largest cluster spans two files (ac.c and hashtab.c), while the 5^{th} largest cluster includes parts of ac.c only.

The largest cluster of acct is spread over six functions, log_in, log_out, file_open, file_reader_get_entry, bad_utmp_record and utmp_get_entry. These functions are responsible for *putting accounting records into the hash table* used by the program, *accessing user-defined files*, and *reading entries* from the file. Thus, the purpose of the code in this cluster is to track user login and logout events.

The second largest cluster is spread over two functions hashtab_create and hashtab_resize. These functions are responsible for *creating fresh hash tables* and *resizing existing hash tables* when the number of entries becomes too large. The purpose of the code in this cluster is the memory management in support of the program's main data structure.

The third largest cluster is spread over four functions: hashtab_set_value, log_everyone_out, update_user_time, and hashtab_create. These functions are responsible for *setting values of an entry*, *updating all the statistics* for users, and *resetting the tables*. The purpose of the code from this cluster is the modification of the user accounting data.

The fourth cluster is spread over three functions: hashtab_delete, do_statistics, and hashtab_find. These functions are responsible for *removing entries* from the hash table, *printing out statistics* for users and *finding entries* in the hash table. The purpose of the code from this cluster is maintaining user accounting data and printing results.

The fifth cluster is contained within the function main. This cluster is formed due to the use of a while loop containing various cases based on input to the program. Because of the conservative nature of static analysis, all the code within the loop is part of the same cluster.

Finally, it is interesting to note that functions from the same file or with similar names do not necessarily belong to the same cluster. Although six of the functions considered above have the common prefix "hashtab", these functions are not part of the same cluster. Instead the functions that work together to provide a particular functionality are found in the same cluster. This case study provides an affirmative answer to *RQ5*. For acct, each of the top five clusters maps to specific functionality, which interestingly is not revealed simply from studying the names of the artifacts.

3.6.2. Case Study: indent

The next case study uses indent to further support to the answer found for *RQ5* in the acct case study. The characteristics of indent are very different from those of acct as indent has a very large dominant coherent cluster (52%) whereas acct has multiple smaller clusters with the largest being 11%. We include indent as a case study to ensure that the answer for *RQ5* is derived from programs with different cluster profiles and sizes giving confidence as to the generality of the answer.

Indent is a Unix utility used to format C source code. It consists of 6,978 LoC with 7,543 vertices in the SDG produced by CodeSurfer. Table 5 shows statistics of the five largest clusters found in the program.

Cluster	Cluster Size		Files	Functions
	%	vertices/lines	spanned	spanned
1	52.1%	3930/2546	7	54
2	3.0%	223/136	3	7
3	1.9%	144/72	1	6
4	1.3%	101/54	1	5
5	1.1%	83/58	1	1

Table 5: indent's top five clusters

Indent has one extremely large coherent cluster that spans 52.1% of the program's vertices. The cluster is formed of vertices from 54 functions spread over 7 source files. This cluster captures most of the logical functionalities of the program. Out of the 54 functions, 26 begin with the common prefix of "handle_token". These 26 functions are individually responsible for handling a specific token during the formatting process. For example, handle_token_colon, handle_token_comma, handle_token_comment, and handle_token_lbrace are responsible for handling the colon, comma, comment, and left brace tokens, respectively.

This cluster also includes multiple handler functions that check the size of the code and labels being handled, such as check_code_size and check_lab_size. Others, such as search_brace, sw_buffer, print_comment, and reduce, help with tracking braces and comments in code. The cluster also spans the main loop of indent (indent_main_loop) that repeatedly calls the parser function parse.

Finally, the cluster consists of code for outputting formatted lines such as the functions better_break, computer_code_target, dump_line, dump_line_code, dump_line_label, inhibit_indenting, is_comment_start, output_line_length and slip_horiz_space, and ones that perform flag and memory management (clear_buf_break_list, fill_buffer and set_priority).

Cluster 1 therefore consists of the main functionality of this program and provides support for *parsing, handling tokens, associated memory management*, and *output*. The parsing, handling of individual tokens and associated memory management are highly inter-twined. For example, the handling of each individual token is dictated by operations of indent and closely depends on the parsing. This code cannot easily be decoupled and, for example, reused. Similarly the memory management code is specific to the data structures used by indent resulting in these many logical constructs to become part of the same cluster.

The second largest coherent cluster consists of 7 functions from 3 source files. These functions handle the arguments and parameters passed to indent. For example, set_option and option_prefix along with the helper function eqin to check and verify that the options or parameters passed to indent are valid. When options are specified without the required arguments, the function arg_missing produces an error message by invoking usage followed by a call to DieError to terminate the program.

Cluster 3, 4 and 5 are less than 3% of the program and are too small to

warrant a detailed discussion. Cluster 3 includes 6 functions that generate numbered/un-numbered backup for subject files. Cluster 4 has functions for reading and ignoring comments. Cluster 5 consists of a single function that reinitializes the parser and associated data structures.

The case study of indent further illustrates that coherent clusters can capture the program's logical model and finds an affirmative answer to research question *RQ5*. However, in cases such as this where the internal functionality is tightly knit, a single large coherent clusters maps to the program's core functionality.

3.6.3. Case Study: bc

The third case study in this series is bc, an open-source calculator, which consists of 9,438 LoC and 5,450 SLoC. The program has nine C files from which CodeSurfer produced 15,076 slices (backward and forward).

Analyzing bc's SCG (row 3, Figure 15), two interesting observations can be made. First, bc contains two large same-backward-slice clusters visible in the light gray landscapes as opposed to the three large coherent clusters. Second, looking at the B-SCG, it can be seen that the x-axis range spanned by the largest same-backward-slice cluster is occupied by the top two coherent clusters shown in the dashed red (dark gray) landscape. This indicates that the same-backward-slice cluster splits into the two coherent clusters.

The statistics for bc's top five clusters are given in Table 6. Sizes of these five clusters range from 32.3% through to 1.4% of the program. Clusters six onwards are less than 1% of the program. *Decluvi*'s Heat Map View for bc (Figure 5) clearly shows the presence of these five clusters. The Project View (Figure 6) shows their distribution over the source files.

Cluster	Cluster Size		Files	Functions
	%	vertices/lines	spanned	spanned
1	32.3%	2432/1411	7	54
2	22.0%	1655/999	5	23
3	13.3%	1003/447	1	15
4	1.6%	117/49	1	2
5	1.4%	102/44	1	1

Table 6: bc's top five clusters

In more detail, Cluster 1 spans all of bc's files except for scan.c and bc.c. This cluster encompasses the core functionality of the program – *loading and handling of equations, converting to* bc's *own number format, performing calculations*, and *accumulating results.* Cluster 2 spans five files, util.c, execute.c, main.c, scan.c, and bc.c. The majority of the cluster is distributed over the latter two files. Even more interestingly, the source code of these two files (scan.c and bc.c) map only to cluster 2 and none of the other top five clusters. This indicates a clear purpose to the code in these files. These two files are solely used for *lexical analysis* and *parsing* of equations. To aid in this task, some utility functions

from util.c are employed. Only five lines of code in execute.c are also part of Cluster 2 and are used for *flushing output* and *clearing interrupt signals.* The third cluster is completely contained within the file number.c. It encompasses functions such as _bc_do_sub, _bc_init_num, _bc_do_compare, _bc_do_add, _bc_simp_mul, _bc_shift_addsub, and _bc_rm_leading_zeros, which are responsible for *initializing* bc's *number formatter, performing comparisons, modulo* and other *arithmetic operations.* Clusters 4 and 5 are also completely contained within number.c. These clusters encompass functions to perform *bcd operations for base ten numbers* and *arithmetic division*, respectively.

The results of the cluster visualizations for bc reveal its high-level structure. This aids an engineer in understanding how the artifacts (e.g., functions and files) of the program interact. The visualization of the clustering thus aids in program comprehension and provides further support for *RQ5.*

The following discussion illustrates a side-effect of *decluvi*'s multi-level visualization, how it can help find potential problems with the structure of a program. Util.c consists of small utility functions called from various parts of the program. This file contains code from Clusters 1 and 2 (Figure 6). Five of the utility functions belong with Cluster 1, while six belong with Cluster 2. Furthermore, Figure 7 shows that the distribution of the two clusters in red (dark gray) and blue (medium gray) within the file are well separated.

Both clusters do not occur together inside any function with the exception of init_gen (highlighted by the rectangle in first column of Figure 7). The other functions of util.c thus belong to either Cluster 1 or Cluster 2. Separating these utility functions into two separate source files where each file is dedicated to functions belonging to a single cluster would improve the code's logical separation and file-level cohesion. This would make the code easier to understand and maintain at the expense of a very simple refactoring. In general, this example illustrates how *Decluvi* visualization can provide an indicator of potential points of code degradation during evolution.

Finally, the Code View for function init_gen shown in Figure 8 includes Lines 244, 251, 254, and 255 in red (dark gray) from Cluster 1 and Lines 247, 248, 249, and 256 in blue (medium gray) from Cluster 2. Other lines, shown in light gray, belong to smaller clusters and lines containing no executable code. Ideally, clusters should capture a particular functionality; thus, functions should generally not contain code from multiple clusters (unless perhaps the clusters are completely contained within the function). Functions with code from multiple clusters reduce code separation (hindering comprehension) and increase the likelihood of ripple-effects [16]. Like other initialization functions, bc's init_gen form an exception to this guideline.

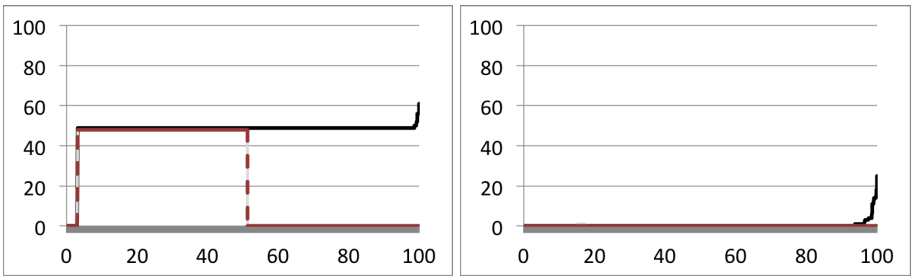
This case study not only provides an affirmative answer to research question *RQ5*, but also illustrates that the visualization is able to reveal structural defects in programs.

3.6.4. Case Study: copia

The final case study in this series is copia, an industrial program used by the ESA to perform signal processing. Copia is the smallest program considered in

Cluster number	Cluster Size %	Cluster Size vertices/lines	Files spanned	Functions spanned
1	48%	1609/882	1	239
2	0.1%	4/2	1	1
3	0.1%	4/2	1	1
4	0.1%	4/2	1	1
5	0.1%	2/1	1	1

Table 7: copia's top five clusters



(a) *Original* (b) *Modified*

Figure 17: SCGs for the program copia

this series of case studies with 1,168 LoC and 1,111 SLoC all in a single C file. Its largest coherent cluster covers 48% of the program. The program is at the top of the group with large coherent clusters. CodeSurfer extracts 6,654 slices (backward and forward).

The B-SCG for copia is shown in Figure 17a. The single large coherent cluster spanning 48% of the program is shown by the dashed red (dark gray) line (running approx. from 2% to 50% on the x-axis). The plots for same-backward-slice cluster sizes (light gray line) and the coherent cluster sizes (dashed line) are identical. This is because the size of the coherent clusters are restricted by the size of the same-backward-slice clusters. Although the plot for the size of the backward slices (black line) seems to be the same from the 10% mark to 95% mark on the x-axis, the slices are not exactly the same. Only vertices plotted from 2% through to 50% have exactly same backward and forward slice resulting in the large coherent cluster.

Table 7 shows statistics for the top five coherent clusters found in copia. Other than the largest cluster which covers 48% of the program, the rest of the clusters are extremely small. Clusters 2–5 include no more than 0.1% of the program (four vertices) rendering them too small to be of interest. This suggests a program with a single functionality or structure.

During analysis of copia using *decluvi*, the File View (Figure 18) reveals an intriguing structure. There is a large block of code with the same spatial arrangement (bounded by the dotted black rectangle in Figure 18) that belongs to the largest cluster of the program. It is unusual for so many consecutive source

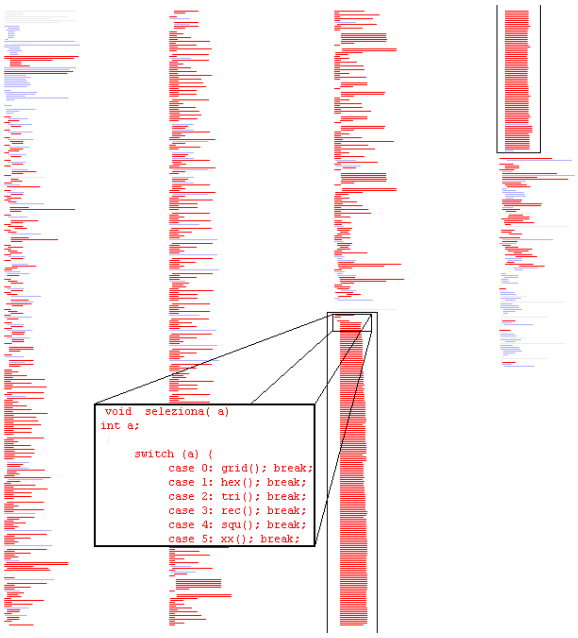


Figure 18: File View for the file copia.c of Program copia. Each line of pixels represent the cluster data for one source code line. The lines in red (dark gray in black & white) are part of the largest cluster. The lines in blue (medium gray) are part of smaller clusters. A rectangle highlights the switch statement that holds the largest cluster together.

code lines to have nearly identical length and indentation. Inspection of the source code reveals that this block of code is a switch statement handling 234 cases. Further investigation shows that copia has 234 small functions that eventually call one large function, seleziona, which in turn calls the smaller functions effectively implementing a finite state machine. Each of the smaller functions returns a value that is the next state for the machine and is used by the switch statement to call the appropriate next function. The primary reason for the high level of dependence in the program lies with the statement switch(next_state), which controls the calls to the smaller functions. This causes what might be termed 'conservative dependence analysis collateral damage' because the static analysis cannot determine that when function f() returns the constant value 5 this leads the switch statement to eventually invoke function g(). Instead, the analysis makes the conservative assumption that a call to f() might be followed by a call to any of the functions called in the switch statement, resulting in a mutual recursion involving most of the program.

Although the coherent cluster still shows the structure of the program and includes all these stub functions that work together, this is a clear case of dependence pollution [10], which is avoidable. To illustrate this, the code was

re-factored to simulate the replacement of the integer next_state with direct recursive function calls. The SCG for the modified version of copia is shown in Figure 17b where the large cluster has clearly disappeared. As a result of this reduction, the potential impact of changes to the program will be greatly reduced, making it easier to understand and maintain. This is even further amplified for automatic static analysis tools such as CodeSurfer. Of course, in order to do a proper re-factoring, the programmer will have to consider ways in which the program can be re-written to change the flow of control. Whether such a refactoring is deemed cost-effective is a decision that can only be taken by the engineers and managers responsible for maintaining the program in question.

This case study provides further affirmative support in support of *RQ5* by showing the structure and dependency within the program. It also identifies potential refactoring points which can improve the performance of static analysis tools and make the program easier to understand.

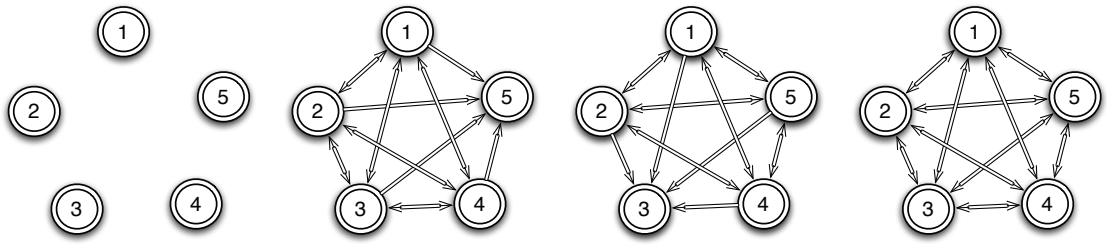
3.7. Inter-cluster Dependence

This final section addresses research question *RQ6: Does dependence between coherent clusters induce larger dependence structures?* The question attempts to reveal whether there is dependence (slice inclusion) relationship between the vertices of different coherent clusters and whether this can be used to identify larger dependence structures. If such containment occurs, it must be a strict containment relationship because of the external and internal requirements of coherent clusters. This section empirically investigates the existence of such containment. In the series of case studies presented earlier we have seen that coherent clusters map to logical components of a system and can be used to gain an understanding of the architecture of the program. If such containment relationships were present, coherent clusters could be treated as atomic subsystems which could be combined to deduce higher-level abstractions of the system, opening up the potential use of coherent clusters in reverse engineering.

All vertices of a coherent cluster share the same external and internal dependence, that is, all vertices have the same backward slice and also the same forward slice. Because of this, any backward/forward slice that includes a vertex from a cluster will also include all other vertices of the same cluster. The same is not true for non-coherent clusters. For example, in the case of a same-backward-slice cluster, a vertex contained within the forward slice of any vertex of the cluster is not guaranteed to be in the forward slice of other vertices of the same cluster.

The study exploits this unique property of coherent clusters to investigate whether or not a backward slice taken with respect to a vertex of a coherent cluster includes vertices of another cluster. Note that if vertices of coherent cluster x are contained in the slice taken with respect to a vertex of coherent cluster y, then all vertices of x are contained in the slice taken with respect to each vertex of y.

Figure 19 shows *Cluster Dependence Graphs* (CDG) for each of the four case study subjects. Only the five largest clusters of the case study subjects



(a) copia (b) acct (c) indent (d) bc

Figure 19: Cluster Dependence Graphs

are considered during this study. The graphs depict slice containment relationships between the top five clusters of each program. In these graphs, the top five clusters are represented by nodes (1 depicts the largest coherent cluster, while 5 is the 5^{th} largest cluster) and the directional edges denote backward slice inclusion relationships: $a \rightarrow b$ depicts that vertices of cluster b depend on vertices of cluster a, that is, a backward slice of any vertex of cluster b will include all vertices of cluster a. Bi-directional edges show mutual dependencies, whereas uni-directional edges show dependency in one direction only. In the graph for copia (Figure 19a), the top five clusters have no slice inclusion relationships between them (absence of edges between the nodes of the CDG). Looking at Table 7, only the largest cluster of copia is truly large at 48%, while the other four clusters are extremely small making them unlikely candidates for inter-cluster dependence.

For acct (Figure 19b) there is a dependence between all of the top five clusters. In fact, there is mutual dependence between clusters 1, 2, 3 and 4, while cluster 5 depends on all the other four clusters but not mutually. Clusters 1 through 4 contain logic for manipulating, accessing, and maintaining the hash tables, making them interdependent. Cluster 5 on the other hand is a loop structure within the main function for executing different cases based on command line inputs. Similarly for indent (Figure 19c), clusters 1, 2, 4, and 5 are mutually dependent and 3 depends on all the other top five clusters but not mutually.

Finally, in the case of bc (Figure 19d), all the vertices from the top five clusters are mutually inter-dependent. The rest of this section uses bc as an example where this mutual dependence is used to identify larger dependence structures by merging of the inter-dependent coherent clusters.

At first glance it may seem that the merging of the coherent clusters is simply reversing the splitting of same-backward-slice or same-forward-slice clusters observed earlier in Section 3.6.3. However, examining the sizes of the top five same-backward-slice clusters, same-forward-slice clusters and coherent clusters for bc illustrates that it is not the case. Table 8 shows the size of these clusters both in terms of number of vertices and as a percentage of the program. The size of the dependence structure resulting from the merging of top five coherent

Cluster Number	Same Backward-Slice Cluster Size vertices	%	Same Forward-Slice Cluster Size vertices	%	Coherent Cluster Size vertices	%
1	4,135	54.86	2,452	32.52	2,432	32.26
2	1,111	14.74	1,716	22.76	1,655	21.96
3	131	1.74	1,007	13.36	1,003	13.31
4	32	0.42	157	2.08	117	1.55
5	25	0.33	109	1.45	102	1.35
					Merged Cluster:	70.43

Table 8: Various cluster statistics of bc

clusters is 70.43%, which is 15.67% larger than the largest same-backward-slice cluster (54.86%) and 37.91% larger than the same-forward-slice cluster (32.35%). Therefore, the set of all (mutually dependent) vertices from the top five coherent clusters when merged form a larger dependence structure.

This section thus provides an affirmative answer to *RQ6*, there are dependence relationships between coherent clusters and in some cases there are mutual dependences between large coherent clusters. Furthermore, combining inter-dependent coherent clusters result in dependence structures larger than same-slice clusters. This also indicates that the sizes of dependence clusters reported by previous studies [9, 10, 11, 26, 29] maybe conservative and mutual dependence clusters are *larger* and more prevalent than previously reported. Finally, this inter-cluster relationship can also be leveraged for hierarchical clustering in reverse engineering applications where coherent clusters can be treated as atomic components that are clustered based on the dependence between coherent clusters.

3.8. Threats to validity

This section presents threats to the validity of the results presented. The primary external threat arises from the possibility that the programs selected are not representative of programs in general (i.e., the findings of the experiments do not apply to 'typical' programs). This is a reasonable concern that applies to any study of program properties. To address this issue, a set of thirty open-source and industrial programs were analyzed in the quantitative study. The programs were not selected based on any criteria or property and thus represent a random selection. However, these were from the set of programs that were studied in previous work on dependence clusters to facilitate comparison with previous results. In addition, all of the programs studied were C programs, so there is greater uncertainty that the results will hold for other programming paradigms such as object-oriented Java or aspect-oriented.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variable. In this experiment, one possible threat arises from the potential for faults in the

slicer. A mature and widely used slicing tool (CodeSurfer) is used to mitigate this concern. Another possible concern surrounds the precision of the pointer analysis used. Section 3.2 discusses the various pointer analysis settings and validates its precision. Finally, the use of hash values to approximate slice content is also a source of potential threat. Hash functions are prone to hash collision, which is minimized by carefully crafting the hash function, and its use is validated in Section 3.3.

4. Related Work

In testing, dependence analysis has been shown to be effective at reducing the computational effort required to automate the test-data generation process [2]. In software maintenance, dependence analysis is used to protect a software maintainer against the potentially unforeseen side effects of a maintenance change. This can be achieved by measuring the impact of the proposed change [16] or by attempting to identify portions of code for which a change can be safely performed free from side effects [22, 41]. A recently proposed impact analysis framework [1] reports that impact sets are often part of large dependence clusters when using time consuming but high precision slicing. When low precision slicing is used, the study reports smaller dependence clusters. This paper uses the most precise static slicing available.

Dependence clusters have previously been linked to software faults [15] and have been identified as a potentially harmful 'dependence anti-pattern' [9]. The presence of large dependence cluster was thought to reduce the effectiveness of testing and maintenance support techniques. Having considered dependence clusters harmful, previous work on dependence clusters focuses on locating dependence clusters, understanding their cause, and removing them.

The first of these studies [10, 26] were based on efficient technique for locating dependence clusters and identifying dependence pollution (avoidable dependence clusters). One common cause of large dependence clusters is the use of global variables. A study of 21 programs found that 50% of the programs had a global variable that was responsible for holding together large dependence clusters [12]. Other work on dependence clusters in software engineering has considered clusters at both low-level [10, 26] (SDG based) and high-level [19, 34] (models and functions) abstractions.

This paper extends our previous work which introduced coherent dependence clusters [29] and *decluvi* [28]. Previous work established the existence of coherent dependence clusters and detailed the functionalities of the visualization tool. This paper extends previous work in many ways, firstly by introducing an efficient hashing algorithm for slice approximation. This improves on the precision of previous approximation techniques from 79% to 96%, resulting in precise and accurate clustering. The coherent cluster existence study is extended to empirically validate the results by considering 30 production programs. Additional case studies show that coherent clusters can help reveal the structure of a program and identify structural defects. Finally, this paper studies inter-cluster

dependence for the first time which can form the base of reverse engineering techniques.

In some ways our work follows the evolutionary development of the study of software clones [6], which were thought to be harmful and problematic when first observed. Further reflection and analysis revealed that these code clone structures were a widespread phenomena that deserved study and consideration. While engineers needed to be aware of them, it remains a subject of much debate as to whether or not they should be refactored, tolerated or even nurtured [17, 30].

We believe the same kind of discussion may apply to dependence clusters. While dependence clusters may have significant impact on comprehension and maintenance and though there is evidence that these clusters are a widespread phenomena, it is not always obvious whether they can be or should be removed or refactored. There may be a (good) reason for the presence of a cluster and/or it may not be obvious how it can be removed (though its presence should surely be brought to the attention of the software maintainer). These observations motivate further study to investigate and understand dependence clusters, and to provide tools to support software engineers in their analysis.

In support of future study, we make available all data from our study at the website `http://www.cs.ucl.ac.uk/staff/s.islam/decluvi.html`. The reader can obtain the slices for each program studied and the clusters they form, facilitating replication of our results and other studies of dependence and dependence clusters.

The visualizations used in this paper are similar to those used for program comprehension. *Seesoft* [18] is a seminal tool for line oriented visualization of software statistics. The system pioneered four key ideas: reduced representation, coloring by statistic, direct manipulation, and capability to read actual code. The reduced representation was achieved by displaying files in columns with lines of code as lines of pixels. This approach allows 50,000 lines of code to be shown on a single screen.

The SeeSys System [5] introduced tree maps to show hierarchical data. It displays code organized hierarchically into subsystems, directories, and files by representing the whole system as a rectangle and recursively representing the various sub-units with interior rectangles. The area of each rectangle is used to reflect statistic associated with the sub-unit. *Decluvi* builds on the SeeSoft concepts through different abstractions and dynamic mapping of line statistics removing the 50,000 line limitation.

An alternative software visualization approach often used in program comprehension does not use the "line of pixels" approach, but instead uses nested graphs for hierarchical fish-eye views. Most of these tools focus on visualizing high-level system abstractions (often referred to as 'clustering' or 'aggregation') such as classes, modules, and packages. A popular example is the reverse engineering tool Rigi [39].

5. Summary

Dependence clusters are known to be problematic as they inhibit program understanding and maintenance. This paper introduces and evaluates the presence of a specialized form of dependence cluster: the coherent cluster. Such clusters have vertices that share the same internal and external dependencies. This paper presents new approximations that support the efficient and accurate identification of coherent clusters. Empirical evaluation finds that 23 of the 30 subject programs have at least one large coherent cluster. A series of four case studies illustrate that coherent clusters map to logical program constructs and can be used to depict the structure of a program. In all four case studies, coherent clusters map to subsystems, each of which is responsible for implementing concise functionality. As side-effects of the study, we find that the visualization of coherent clusters can identify potential structural problems as well as refactoring opportunities. Finally, the paper discusses inter-cluster dependence and how mutual dependencies between clusters may be exploited to reveal large dependence structure which can form the basis of reverse engineering efforts.

Looking forward, the copia case study highlights how static analysis can suffer from collateral damage caused by its conservative nature. This can lead to large slices resulting in large clusters. Dynamic slicing, which does not suffer from the need to make conservative approximations, may lead to more precise, smaller clusters. Future work will consider the use of dynamic slicing in dependence cluster formalization to determine how the size and location of dependence clusters within a program varies.

References

[1] Acharya, M., Robinson, B., May 2011. Practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011). ACM, pp. 746–755.

[2] Ali, S., Briand, L., Hemmati, H., Panesar-Walawege, R., 2010. A systematic review of the application and empirical investigation of search-based test case generation. Software Engineering, IEEE Transactions on 36 (6), 742–762.

[3] Andersen, L. O., May 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen, (DIKU report 94/19).

[4] Anderson, P., Teitelbaum, T., 2001. Software inspection using CodeSurfer. In: First Workshop on Inspection in Software Engineering. pp. 1–9.

[5] Baker, M. J., Eick, S. G., 1995. Space-filling software visualization. Journal of Visual Languages & Computing 6 (2), 119–133.

[6] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E., 2007. Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. 33 (9), 577–591.

[7] Beszédes, Á., Gergely, T., Jász, J., Toth, G., Gyimóthy, T., Rajlich, V., October 2007. Computation of static execute after relation with applications to software maintenance. In: 23^{rd} IEEE International Conference on Software Maintenance (ICSM 2007). IEEE Computer Society Press, Los Alamitos, California, USA, pp. 295–304.

[8] Binkley, D., May 2007. Source code analysis: A road map. ICSE 2007 Special Track on the Future of Software Engineering.

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

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

[11] Binkley, D., Harman, M., 2009. Identifying 'linchpin vertices' that cause large dependence clusters. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 89–98.

[12] Binkley, D., Harman, M., Hassoun, Y., Islam, S., Li, Z., April 2009. Assessing the impact of global variables on program dependence and dependence clusters. Journal of Systems and Software.

[13] Binkley, D. W., Harman, M., Sep. 2003. 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, pp. 44–53.

[14] Black, S., Counsell, S., Hall, T., Bowes, D., 2009. Fault analysis in OSS based on program slicing metrics. In: EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE Computer Society, pp. 3–10.

[15] Black, S., Counsell, S., Hall, T., Wernick, P., 2006. Using program slicing to identify faults in software. In: Beyond Program Slicing. No. 05451 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

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

[17] Bouktif, S., Antoniol, G., Merlo, E., Neteler, M., 8-12 Jul. 2006. A novel approach to optimize clone refactoring activity. In: GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation. Vol. 2. ACM Press, Seattle, Washington, USA, pp. 1885–1892.

[18] Eick, S., Steffen, J., Sumner, E., 1992. Seesoft-a tool for visualizing line oriented software statistics. IEEE Trans. Softw. Eng. 18, 957–968.

[19] Eisenbarth, T., Koschke, R., Simon, D., 2003. Locating features in source code. IEEE Trans. Softw. Eng. 29 (3).

[20] Fahndrich, M., Foster, J. S., Su, Z., Aiken, A., Jun. 1998. Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation. Association for Computer Machinery, pp. 85–96.

[21] Ferrante, J., Ottenstein, K. J., Warren, J. D., Jul. 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9 (3), 319–349.

[22] Gallagher, K. B., Lyle, J. R., Aug. 1991. Using program slicing in software maintenance. IEEE Trans. Softw. Eng. 17 (8), 751–761.

[23] Garey, M. R., Johnson, D. S., 1990. Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA.

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

[25] Hajnal, Á., Forgács, I., 2011. A demand-driven approach to slicing legacy COBOL systems. Journal of Software Maintenance and Evolution: Research and Practice.

[26] Harman, M., Binkley, D., Gallagher, K., Gold, N., Krinke, J., Oct. 2009. Dependence clusters in source code. ACM Transactions on Programming Languages and Systems 32 (1), article 1.

[27] Horwitz, S., Reps, T., Binkley, D., January 1990. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12, 26–60.

[28] Islam, S., Krinke, J., Binkley, D., 2010. Dependence cluster visualization. In: SoftVis'10: 5th ACM/IEEE Symposium on Software Visualization. ACM.

[29] Islam, S., Krinke, J., Binkley, D., Harman, M., 2010. Coherent dependence clusters. In: PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM.

[30] Kapser, C., Godfrey, M. W., 2008. "Cloning considered harmful" considered harmful: patterns of cloning in software. Empirical Software Engineering 13 (6), 645–692.

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

[32] Krinke, J., Oct. 2002. Evaluating context-sensitive slicing and chopping. In: IEEE International Conference on Software Maintenance. IEEE Computer Society Press, Los Alamitos, California, USA, pp. 22–31.

[33] Krinke, J., 2003. Context-sensitive slicing of concurrent programs. In: Proc. ESEC/FSE. pp. 178–187.

[34] Mitchell, B. S., Mancoridis, S., 2006. On the automatic modularization of software systems using the bunch tool. IEEE Trans. Softw. Eng. 32 (3), 193–208.

[35] Ottenstein, K. J., Ottenstein, L. M., 1984. The program dependence graph in software development environments. Proceedings of the ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environmt, SIGPLAN Notices 19 (5), 177–184.

[36] Ren, X., Chesley, O., Ryder, B. G., 2006. Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis. IEEE Trans. Softw. Eng. 32 (9), 718–732.

[37] Ren, X., Ryder, B. G., Störzer, M., Tip, F., 2005. Chianti: a change impact analysis tool for Java programs. In: 27th International Conference on Software Engineering (ICSE 2005). ACM, pp. 664–665.

[38] Savernik, L., 2007. Entwicklung eines automatischen Verfahrens zur Auflösung statischer zyklischer Abhängigkeiten in Softwaresystemen (in german). In: Software Engineering 2007 - Beiträge zu den Workshops. Vol. 106 of LNI. GI, pp. 357–360.

[39] Storey, M.-A. D., Wong, K., Muller, H. A., 1997. Rigi: a visualization environment for reverse engineering. In: Proceedings of the 19th international conference on Software engineering - ICSE '97. ACM Press, New York, New York, USA, pp. 606–607.

[40] Szegedi, A., Gergely, T., Beszédes, Á., Gyimóthy, T., Tóth, G., 2007. Verifying the concept of union slices on Java programs. In: 11^{th} European Conference on Software Maintenance and Reengineering (CSMR '07). pp. 233–242.

[41] Tonella, P., June 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Trans. Softw. Eng. 29, 495–509.

[42] Weiser, M., Park, C., 1981. Program slicing. In: International Conference on Software Engineering.

[43] Wheeler, D. A., 2004. SLOC count user's guide. http://www.dwheeler.com/sloccount/sloccount.html.

[44] Yau, S. S., Collofello, J. S., Sep. 1985. Design stability measures for software maintenance. IEEE Trans. Softw. Eng. 11 (9), 849–856.