The Journal of Systems and Software 88 (2014) 1–24

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Coherent clusters in source code☆

Syed Islam[a,*], Jens Krinke[a], David Binkley[b], Mark Harman[a]

[a] University College London, United Kingdom
[b] Loyola University Maryland, United States

ARTICLE INFO

Article history:
Received 19 November 2012
Received in revised form 16 July 2013
Accepted 18 July 2013
Available online 21 August 2013

Keywords:
Dependence analysis
Program comprehension
Software clustering

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 substantial coherent clusters. A series of subsequent case studies uncover that all clusters of significant size map to a logical functionality and correspond to a 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 also brings out subtle deficiencies in program structure and identifies potential refactoring candidates. A study of inter-cluster dependence is used to highlight how coherent clusters are connected to each other, revealing higher-level structures, which can be used in reverse engineering. Finally, studies are presented to illustrate how clusters are not correlated with program faults as they remain stable during most system evolution.

1. Introduction

Program dependence analysis is a foundation for many activities in software engineering such as testing, comprehension, and impact analysis (Binkley, 2007). For example, it is essential to understand the relationships between different parts of a system when making changes and the impacts of these changes (Gallagher and Lyle, 1991). This has led to both static (Yau and Collofello, 1985; Black, 2001) and blended (static and dynamic) (Ren et al., 2006, 2005) 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 element of the cluster. If such a dependence cluster is very large, then this mutual dependence clearly has implications related to the cost of maintaining the code.

In previous work (Binkley and Harman, 2005), 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 (Harman et al., 2009). 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 by other authors, both in open source and in proprietary code (Acharya and Robinson, 2011; Beszédes et al., 2007; Szegedi et al., 2007). Large dependence clusters were also found in Java systems (Beszédes et al., 2007; Savernik, 2007; Szegedi et al., 2007) and in legacy Cobol systems (Hajnal and Forgács, 2011).

There has been interesting work on the relationship between faults, program size, and dependence clusters (Black et al., 2006), and between impact analysis and dependence clusters (Acharya and Robinson, 2011; Harman et al., 2009). Large dependence clusters can be thought of as dependence 'anti-patterns' because of the high impact that a change anywhere in the cluster has. For example, it may lead to problems for on-going software maintenance and evolution (Acharya and Robinson, 2011; Binkley et al., 2008; Savernik, 2007). As a result, refactoring has been proposed

* Corresponding author.
E-mail address: s.islam@cs.ucl.ac.uk (S. Islam).

0164-1212/$ – see front matter © 2013 The Authors. Published by Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.07.040

as a technique for breaking larger clusters of dependence into smaller clusters (Binkley and Harman, 2005; Black et al., 2009).

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 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 paper is part of a series of work that we have conducted in the area of dependence clusters. The overarching motivation for this work is to gain a better understanding of the dependence clusters found in programs. Although this paper is a continuation of our previous work on dependence clusters, we present the work in a completely new light. In this paper we show that the specialized version of dependence clusters, *coherent* clusters are found in abundance in programs and need not be regarded as problems. We rather show that these clusters map to logical program structures which will aid developers in program comprehension and understanding. Furthermore, this paper extends the current knowledge in the area and motivates future work by presenting initial results of inter-cluster dependence which can be used as a foundation for reverse engineering. We answer several representative open questions such as whether clusters are related to program faults and how clusters change over time during system evolution.

The primary contributions of the paper are as follows:

1 An 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 validation studies consider the impact of data-flow analysis precision and the precision of the approximation used to efficiently identify coherent clusters.
3 A series of four case studies shows how coherent clusters map to logical program structures.
4 A study of inter-cluster dependence highlights how coherent clusters form the building blocks of larger dependence structures where identification can support, as an example, reverse engineering.
5 A study of bug fixes finds no relationship between program faults and coherent clusters implying that dependence clusters are not responsible for program faults.
6 A longitudinal study of system evolution shows that coherent clusters remain stable during evolution thus depicting the core architecture of systems.

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. It also includes studies on program faults and system evolution and their relationship to coherent clusters. 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. Previous work (Binkley and Harman, 2005; Harman et al., 2009) 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. The section also reviews the current graph-based visualizations for dependence clusters.

2.1. Dependence clusters

Informally, *mutually-dependent clusters* are maximal sets of program statements that mutually depend upon one another (Harman et al., 2009). They are formalized in terms of mutually dependent sets in the following definition.

Definition 2.1 *(Mutually-dependent set and cluster (Harman et al., 2009)).* 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 (Weiser, 1984). A well known approximation is based on Weiser's *program slice* (Weiser, 1984): 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) (Horwitz et al., 1990). 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

[1] Preliminary results were presented at PASTE (Islam et al., 2010b).

reached by the definition (Ferrante et al., 1987). 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 (Ferrante et al., 1987). 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 (Horwitz et al., 1990) and threads (Krinke, 1998). 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 (Binkley and Harman, 2005, 2003; Horwitz et al., 1990; Krinke, 2002, 2003).

Two kinds of SDG slices are used in this paper: backward slices and forward slices (Horwitz et al., 1990; Ottenstein and Ottenstein, 1984). The backward slice taken with respect to vertex v, denoted 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 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 Fig. 1 illustrates the non-transitivity of slice inclusion. The program has six assignment statements (assigning the variables a, 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 Fig. 2 using a directed graph where the nodes of the graph represent the assignment statements and the edges represent the backward slice inclusion relationship from Fig. 1. The table on the right in Fig. 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.

backward slice on assignment to						
a	b	c	d	e	f	P
						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:

Fig. 1. Dependence intransitivity and clusters.

2.2. Slice-based clusters

A *slice-based cluster* is a maximal set of vertices included in each other's slice. The following definition essentially instantiates Definition 2.1 using *BSlice*. Because $x \in BSlice(y) \Leftrightarrow y \in FSlice(x)$ 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 2.2 *(Backward-slice MDS and cluster (Harman et al., 2009)).* A *backward-slice MDS* is a set of SDG vertices, V, such that $\forall x, y \in V : x \in BSlice(y)$.

A *backward-slice cluster* is a backward-slice MDS contained within no other backward-slice MDS.

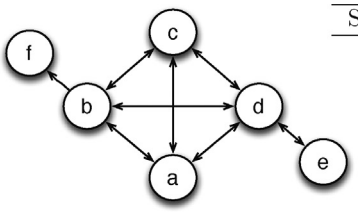
Note that as x and y are interchangeable, this is equivalent to $\forall x, y \in V : x \in BSlice(y) \wedge y \in BSlice(x)$. Thus, any unordered pair (x, y) with $x \in BSlice(y) \wedge y \in BSlice(x)$ creates an edge (x, y) in an undirected graph in which a complete subgraph is equivalent to a backward-slice MDS and a backward-slice cluster is equivalent to a maximal clique. Therefore, the clustering problem is the NP-Hard *maximal cliques* problem (Bomze et al., 1999) making Definition 2.2 prohibitively expensive to implement.

In the example shown in Fig. 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 Fig. 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.

2.3. Same-slice clusters

An alternative definition uses the *same-slice* relation in place of slice inclusion (Binkley and Harman, 2005). 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



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}

Fig. 2. Backward slice inclusion relationship for Fig. 1.

following definitions for *same-slice cluster*. The first uses backward slices and the second forward slices.

Definition 2.3 *(Same-slice MDS and cluster (Harman et al., 2009)).* A *same-backward-slice MDS* is a set of SDG vertices, V, such that

$\forall x, y \in V : BSlice(x) = BSlice(y)$.

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

$\forall x, y \in V : FSlice(x) = FSlice(y)$.

A *same-forward-slice cluster* is a same-forward-slice MDS contained within no other same-forward-slice MDS.

Because $x \in BSlice(x)$ and $x \in FSlice(x)$, 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 2.2) would be identical to a same-backward-slice MDS (Definition 2.3). However, as illustrated by the examples in Fig. 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 Fig. 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 does not form the same-slice cluster with either set because it has an 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 2.2) and same-slice clusters (Definition 2.3), 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 2.4 *(Coherent MDS and cluster (Islam et al., 2010b)).* A *coherent MDS* is a MDS 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). In the slice-based instantiation, a set of vertices V forms a coherent MDS if

$\forall x, y \in V : x \in BSlice(y)$	the internal requirement of an MDS
$\wedge a \in BSlice(x) \Rightarrow a \in BSlice(y)$	x and y depend on same external a
$\wedge a \in FSlice(x) \Rightarrow a \in FSlice(y)$	x and y impact on same external a

Because x and y are interchangeable

$$\begin{aligned}\forall x, y \in V : \quad & x \in BSlice(y) \\ & \wedge a \in BSlice(x) \Rightarrow a \in BSlice(y) \\ & \wedge a \in FSlice(x) \Rightarrow a \in FSlice(y) \\ & \wedge y \in BSlice(x) \\ & \wedge a \in BSlice(y) \Rightarrow a \in BSlice(x) \\ & \wedge a \in FSlice(y) \Rightarrow a \in FSlice(x)\end{aligned}$$

This is equivalent to

$$\begin{aligned}\forall x, y \in V : \quad & x \in BSlice(y) \wedge y \in BSlice(x) \\ & \wedge (a \in BSlice(x) \Leftrightarrow a \in BSlice(y)) \\ & \wedge (a \in FSlice(x) \Leftrightarrow a \in FSlice(y))\end{aligned}$$

which simplifies to

$$\forall x, y \in V : BSlice(x) = BSlice(y) \wedge FSlice(x) = FSlice(y)$$

and can be used to define coherent-slice MDS and clusters:

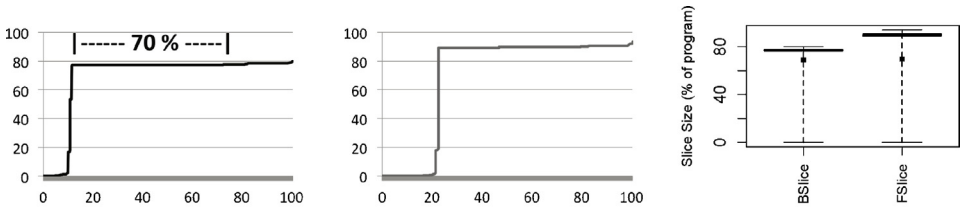
Definition 2.5 *(Coherent-slice MDS and cluster (Islam et al., 2010b)).* A *coherent-slice MDS* is a set of SDG vertices, V, such that

$\forall x, y \in V : BSlice(x) = BSlice(y) \wedge FSlice(x) = FSlice(y)$

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 $x \in BSlice(y) \Leftrightarrow y \in FSlice(x)$. 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 2.1), 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 that both external effects are captured, both backward and forward slices are required for coherent-slice clusters.

In Fig. 2 the set of vertices {a, c} form a coherent cluster as both these vertices have exactly the same backward and forward slices.



(a) B-MSG (b) F-MSG (c) Slice Size Distribution

Fig. 3. Slice size distribution for bc.

That is, they share 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.

Coherent slice-clusters have an important property: If a slice contains a vertex of a coherent slice-cluster V, it will contain all vertices of the cluster:

$$BSlice(x) \cap V \neq \emptyset \Rightarrow BSlice(x) \cap V = V \quad (1)$$

This holds because:

$$\forall y, y' \in V \quad : \quad y \in BSlice(x) \Rightarrow x \in FSlice(y)$$
$$\Rightarrow x \in FSlice(y') \Rightarrow y' \in BSlice(x)$$

The same argument clearly holds for forward slices. However, 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.

2.5. Hash based coherent slice clusters

The computation of coherent-slice clusters (Definition 2.5) 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 2.6 *(Hash-based coherent-slice MDS and cluster (Islam et al., 2010b)).* A *hash-based coherent-slice MDS* is a set of SDG vertices, V, such that

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

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

A description of the hash function H along with the evaluation of its precision is presented 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, a hash-based coherent-slice cluster is referred to simply as a *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) (Binkley and Harman, 2005), 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 9438 lines of code represented by 7538 SDG vertices. The B-MSG for bc, shown in Fig. 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) (Islam et al., 2010b), 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 ordered along the x-axis using three values, primarily according to their slice size, secondarily according to their same-slice cluster size, and finally 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 Fig. 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 (Fig. 4a) shows that bc contains two large same-backward-slice clusters consisting of around 55% and 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.

Fig. 3c shows two box plots depicting the distribution of (backward and forward) slice sizes for bc. The average size of the slices is also displayed in the box plot using a solid square box. Comparing the box plot information to the information provided by the

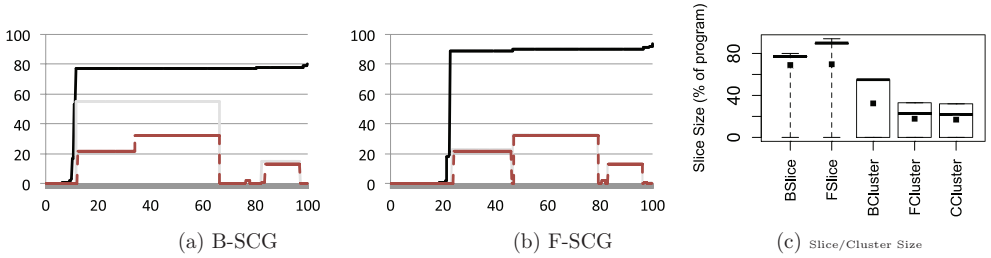


Fig. 4. Slice/cluster size distribution for bc.

MSGs, we can see that all the information available from the box plots can be derived from the MSGs itself (except for the average). However, MSGs show a landscape (slice profile) which cannot be obtained from the box plots. Similarly, the box plots in Fig. 4c show the size distributions of the various clusters (i.e. a vertex is in a cluster of size x) in addition to the slice size distributions. Although the information from these box plots can not be derived from the SCGs shown in Fig. 4a and b directly, the profiles (landscapes) give a better intuition about the clusters, the number of major clusters and their sizes. For our empirical study we use the size of individual clusters and the cluster profile to find mappings between the clusters and program components. Therefore, we drop box plots in favor of SCGs to show the cluster profile and provide additional statistics in tabular format where required.

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 considers the effect of pointer analysis precision and the second considers the validity of hashing in 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 the *decluvi* cluster visualization tool (Islam et al., 2010a), highlight how knowledge of clusters can aid a software engineer. The section then presents studies on inter-cluster dependence, and the relationship of program faults and system evolution to coherent clusters. Finally, threats to validity are considered.

To formalize the goals of this section, the empirical evaluation addresses the following research questions:

- ***RQ*1** What is the effect of pointer analysis precision on coherent clusters?
- ***RQ*2** How precise is hashing as a proxy for comparing slices?
- ***RQ*3** How large are the coherent clusters that exist in production source code and which patterns of clustering can be identified?
- ***RQ*4** Which structures within a program can coherent cluster analysis reveal?
- ***RQ*5** What are the implications of inter-cluster dependence between coherent clusters?
- ***RQ*6** How do program faults relate to coherent clusters?
- ***RQ*7** How stable are coherent clusters during system evolution?

The first two research questions provide empirical verification for the results subsequently presented. *RQ1* establishes the impact of pointer analysis on the clustering, whereas *RQ2* establishes that the hash function used to approximate a slice 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. Conversely, if the clustering is similar for every program then it is unlikely that cluster identification will reveal interesting information about programs. Our findings reveal that, despite the tight constraints inherent in the definition of a coherent dependence cluster, they are, indeed, very common. Also, the cluster profiles for programs are sufficiently different and exhibit interesting patterns.

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* studies the underlying logical structure of programs revealed by these clusters. *RQ5* looks explicitly at inter-cluster dependency and considers areas of software engineering where it may be of interest. *RQ6* presents a study of how program faults relate to coherent clusters, and, finally, *RQ7* studies the effect of system evolution on clustering.

3.1. Experimental subjects and setup

The slices along with the mapping between the SDG vertices and the actual source code are extracted from the mature and widely used slicing tool CodeSurfer (Anderson and Teitelbaum, 2001) (version 2.1). The cluster visualizations were generated by *decluvi* (Islam et al., 2010a) using data extracted from CodeSurfer. The data is generated from slices taken with respect to *source-code representing* SDG vertices. This excludes pseudo vertices introduced into the SDG, e.g., to represent global variables which are modeled as additional pseudo parameters by CodeSurfer. 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 *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.

The study considers the 30 C programs shown in Table 1, which provides a brief description of each program alongside seven 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 (Wheeler, 2004)), ELoC – the number of source code lines that *CodeSurfer* considers to contain executable code, the number of SDG vertices, the number of SDG edges, the number of slices

Table 1
Subject programs.

Program	C files	LoC	SLoC	ELoC	SDG vertex count	SDG edge count	Total slices	Largest coherent cluster size	SDG build time	Clustering time	Description
a2ps	79	46,620	22,117	18,799	224,413	2,025,613	97,170	8%	1m52.048s	583m40.758s	ASCII to Postscript
acct	7	2600	1558	642	7618	22,061	2834	11%	0m15.658s	0m12.545s	Process monitoring
acm	114	32,231	21,715	15,022	159,830	718,683	63,014	43%	1m57.652s	230m18.418s	Flight simulator
anubis	35	18,049	11,994	6947	112,282	561,160	34,618	13%	0m41.253s	70m54.322s	SMTP messenger
archimedes	1	787	575	454	20,136	91,728	2176	4%	0m3.658s	0m12.701s	Semiconductor device simulator
barcode	13	3968	2685	2177	16,721	65,367	9602	58%	0m10.234s	2m56.026s	Barcode generator
bc	9	9438	5450	4535	36,981	355,942	15,076	32%	0m15.359s	13m14.221s	Calculator
byacc	12	6373	5312	4688	45,338	203,675	16,590	7%	0m9.820s	10m15.746s	Parser generator
cflow	25	12,542	7121	5762	68,782	304,615	24,638	8%	0m23.312s	31m25.104s	Control flow analyzer
combine	14	8202	6624	5279	49,288	247,464	29,118	15%	0m14.577s	26m11.625s	File combinator
copia	1	1168	1111	1070	42,435	145,562	6654	48%	0m2.046s	2m35.680s	ESA signal processing code
cppi	13	6261	1950	2554	17,771	67,217	10,280	13%	0m10.514s	2m33.213s	C preprocessor formatter
ctags	33	14,663	11,345	7383	152,825	630,189	31,860	48%	0m27.094s	96m0.948s	C tagging
diction	5	2218	1613	427	5919	17,158	2444	16%	0m5.339s	0m8.189s	Grammar checker
diffutils	23	8801	6035	3638	30,023	113,824	16,122	44%	0m23.384s	9m11.509s	File differencing
ed	8	2860	2261	1788	35,475	142,192	11,376	55%	0m6.602s	6m38.521s	Line text editor
enscript	22	14,182	10,681	9135	67,405	423,349	33,780	19%	0m44.690s	54m0.652s	File converter
findutils	59	24,102	13,940	9431	102,910	177,822	41,462	22%	0m36.250s	21m20.795s	Line text editor
flex	21	23,173	12,792	13,537	89,806	860,859	37,748	16%	0m31.249s	77m28.885s	Lexical Analyzer
garpd	1	669	509	300	5452	14,908	1496	14%	0m1.681s	0m3.670s	Address resolved
gcal	30	62,345	46,827	37,497	860,476	4,565,570	286,000	62%	3m3.946s	5d4h18m35s	Calendar program
gnuedma	1	643	463	306	5223	14,075	1488	44%	1m12.888s	0m4.365s	Development environment
gnushogi	16	16,301	11,664	7175	64,482	277,648	31,298	40%	0m25.907s	47m40.268s	Japanese chess
indent	8	6978	5090	4285	24,109	143,821	7543	52%	0m10.000s	10m3.012s	Text formatter
less	33	22,661	15,207	9759	451,870	2,156,420	33,558	35%	1m56.968s	339m48.985s	Text reader
spell	1	741	539	391	6232	17,574	1740	20%	0m1.663s	0m4.905s	Spell checker
time	6	2030	1229	433	4946	12,971	3352	4%	0m3.120s	0m3.683s	CPU resource measure
userv	2	1378	1112	1022	15,418	54,258	5362	9%	0m13.787s	0m53.332s	Access control
wdiff	4	1652	1108	694	10,077	30,085	2722	6%	0m9.154s	0m39.241s	Diff front end
which	6	3003	1996	753	8830	29,377	3804	35%	0m4.528s	0m24.215s	Unix utility
Sum	602	356,639	232,623	175,883	2,743,073	14,491,187	864,925	–	16m3.891s	–	
Average	20	11,888	7754	5863	91,436	483,040	28,831	27%	0m32.130s	–	

produced, and finally the size (as a percentage of the program's SDG vertex count) of the largest coherent cluster. All LoC metrics are calculated over source files that CodeSurfer considers to contain executable code and, for example, do not include header files.

Columns 10 and 11 provide the runtimes recorded during the empirical study. The runtimes reported are wall clock times captured by the Unix time utility while running the experiments on a 64-bit Linux machine (CentOS 5) with eight Intel(R) Xeon(R) CPU E5450 @ 3.00 GHz processors and 32 GB of RAM. It should be noted that this machine acts as a group server and is accessed by multiple users. There were other CPU intensive processes intermittently running on the machine while these runtimes were collected, and thus the runtimes are only indicative. Column 10 shows the time needed to build the SDG and CodeSurfer project that is subsequently used for slicing. The build time for the projects were quite small and the longest build time (2m33.456s) was required for gcal with 46,827 SLoC. Column 11 shows the time needed for the clustering algorithm to perform the clustering and create all the data dumps for *decluvi* to create cluster visualizations. The process completes in minutes for small programs and can take hours and longer for larger programs. It should be noted that the runtime includes both the slicing phase which runs in $O(ne)$, where n is the number of SDG vertices and e is the number of edges, and the hashing and clustering algorithm which runs in $O(n^2)$. Therefore the overall complexity is $O(ne)$. The long runtime is mainly due to the current research prototype (which performs slicing, clustering and extraction of the data) using the Scheme interface of CodeSurfer in a pipeline architecture. In the future we plan to upgrade the tooling with optimizations for fast and massive slicing (Binkley et al., 2007) and to merge the clustering phase into the slicing to reduce the runtime significantly.

Although the clustering and building the visualization data can take a long time for large projects, it is still useful because the clustering only needs to be done once (for example during a nightly build) and can then be visualised and reused as many times as needed. During further study of the visualization and the clustering we have also found that small changes to the system does not show a change in the clustering, therefore once the clustering is created it still remains viable through small code changes as the clustering is found to represent the core program architecture (Section 3.9). Furthermore, the number of SDG vertices and edges are quite large, in fact even for very small programs the number of SDG vertices is in the thousands with edge counts in the tens of thousands. Moreover, the analysis produces an is-in-the-slice-of relation and graph with even more edges. We have tried several clustering and visualization tools to cluster the is-in-the-slice-of graph for comparison, but most of the tools (such as Gephi Bastian et al., 2009) failed due to the large dataset. Other tools such as CCVisu (Beyer, 2008) which were able to handle the large data set simply produced a blob as a visualization which was not at all useful. The underlying problem is that the is-in-the-slice-of graph is dense and no traditional clustering can handle such dense graphs.

3.2. Impact of pointer analysis precision

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 and the interplay between pointer analysis and downstream dependence analysis precision is complex (Shapiro and Horwitz, 1997). To understand how pointer analysis precision

Table 2
CodeSurfer pointer analysis settings.

Program	Average slice size			Maximum slice size			Average Cluster Size			Maximum Cluster Size		
	L	M	H	L	M	H	L	M	H	L	M	H
a2ps	25,223	23,085	20,897	45,231	44,139	43,987	2249	1705	711	10,728	9295	4002
acct	763	700	621	1357	1357	1357	79	66	40	272	236	162
acm	19,083	17,997	16,509	29,403	28,620	28,359	3566	3408	4197	9356	9179	10,809
anubis	11,120	10,806	9085	16,548	16,347	16,034	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	11,856	11,650	11,626	865	565	246	3060	2191	1097
combine	3512	3347	3316	13,448	13,448	13,448	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	12,681	11,663	11,158	15,483	15,475	15,475	7917	41,99	3955	11,080	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	16,130	16,130	16,130	543	554	539	3140	3242	3243
findutils	7020	6767	5239	11,075	11,050	11,050	1969	1927	1306	4489	4429	2936
flex	9038	8737	8630	17,257	17,257	17,257	622	657	647	3064	3064	3064
garpd	284	242	224	628	628	628	32	31	29	103	103	103
gcal	132,860	123,438	123,427	142,739	142,289	142,289	40,885	40,614	40,614	93,541	88,532	88,532
gnuedma	385	369	368	730	730	368	178	176	174	333	331	330
gnushogi	9569	9248	9141	14,726	14,726	14,726	1577	2857	2820	3787	6225	6179
indent	4104	4058	4045	5704	5704	5704	2036	2032	1985	3402	3399	3365
less	13,592	13,416	13,392	16,063	16,063	16,063	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

impacts the clustering of the programs we study the effect in this section.

Usually, one would choose the pointer analysis with the highest precision but there may be situations where this is not possible and one has to revert to lower precision analysis. This section presents a study on the effect of various levels of pointer analysis precision on the size of slices and subsequently on coherent clusters. It addresses research question *RQ1: What is the effect of pointer analysis precision on coherent clusters?*

CodeSurfer provides three levels of pointer analysis precision (Low, Medium, and High) that provide increasingly precise points-to information at the expense of additional memory and analysis time. The Low setting uses a 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. (1998), which implements a variant of Andersen's pointer analysis algorithm (Andersen, 1994) (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. There is no automatic way to determine whether the slices are correct and precise. Weiser (1984) considers smaller slices to be better. Slice size is often used to measure the impact of the analysis' precision (Shapiro and Horwitz, 1997), similarly we also use slice size as a measure of precision.

The study compares slice and cluster size for CodeSurfer's three precision options (Low, Medium, High) to study the impact of pointer analysis precision. The results are shown in Table 2. Column 1 lists the programs and the other columns present the average slice size, maximum slice size, average cluster size, and maximum cluster size, respectively, for each of the three precision settings. The results for average slice size deviation and largest cluster size deviation are visualized in Figs. 5 and 6. The graphs use the High setting as the base line and show the percentage deviation when using the Low and Medium settings.

Fig. 5 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

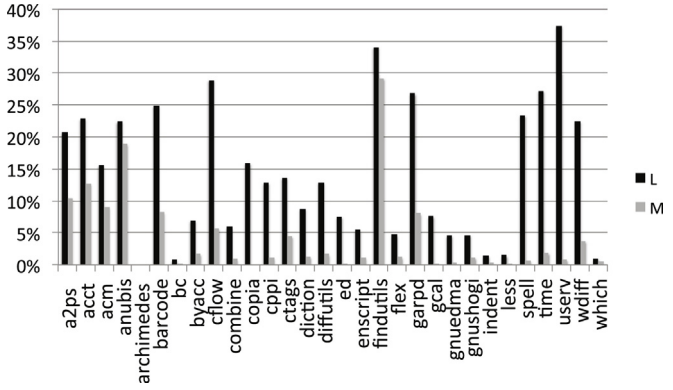


Fig. 5. Percentage deviation of average slice size for Low and Medium CodeSurfer pointer analysis settings.

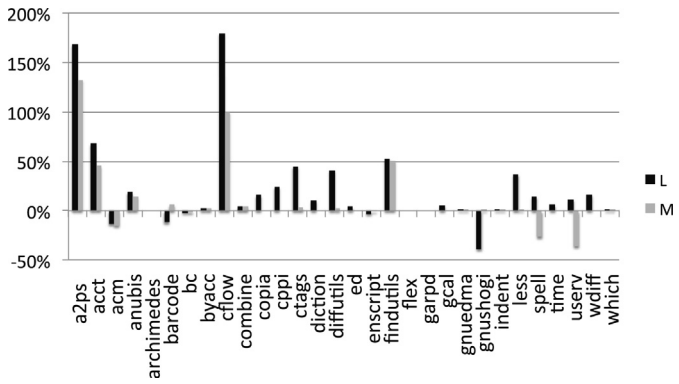


Fig. 6. Percentage deviation of largest cluster size for Low and Medium CodeSurfer pointer analysis settings.

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 29%. With the medium setting, the structure 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.

Fig. 6 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 Fig. 7a. In the graphs it is seen that the slice sizes get smaller and have increased steps in the (black) landscape indicating that the slices become more precise. The red landscape shows that there is a large coherent cluster detected when using the Low setting running from approx. 60–80% on the *x*-axis. This cluster drops in size when using the Medium setting. At the High setting this coherent cluster 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 (Fig. 7b) a similar drop in the slice size and cluster size is observed. However, unlike a2ps the large coherent cluster does not break 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 25–30% on the *x*-axis when using the High setting. In this case the largest cluster has a significant drop in size but does not break into multiple smaller clusters.

Surprisingly, Fig. 6 also shows seven programs where the largest coherent cluster size actually increases when using the highest pointer analysis setting on CodeSurfer. Fig. 7c 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

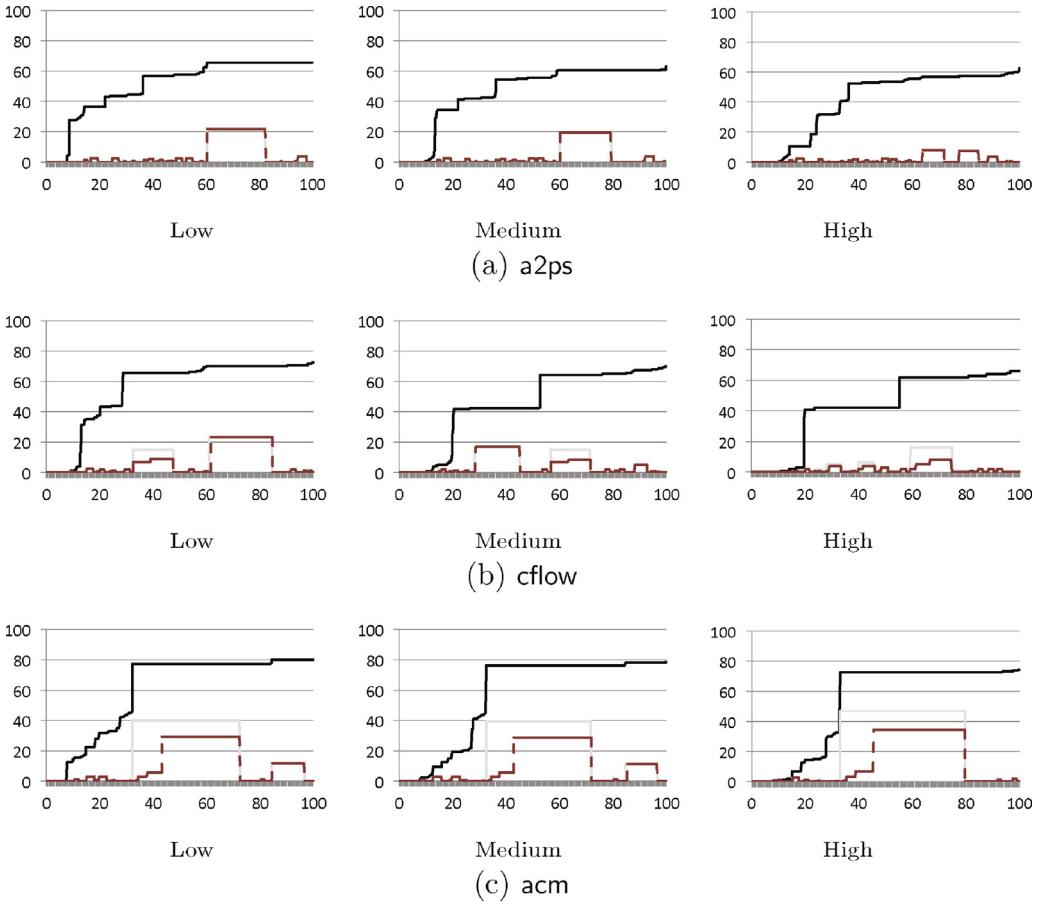


Fig. 7. SCGs for Low, Medium and High pointer settings of CodeSurfer.

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

Fig. 8. Replacement coherent cluster example.

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 Fig. 1 with the code shown in Fig. 8, 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} is one coherent cluster whereas the lower precision analysis include only the set of vertices {a, c} in the same coherent cluster.

Although we do not explicitly report the project build times on CodeSurfer and the clustering runtimes for the lower settings, it has been our experience that in the majority of the cases the build times for the lower settings were smaller. However, as lower pointer analysis settings yield large points-to sets and subsequently larger slices, the clustering runtimes were higher than when using the highest setting. Moreover, in some cases with the lower settings there was an explosive growth in summary edge generation which resulted in exceptionally high project build times and clustering runtimes.

As an answer to *RQ1*, we find that in the majority of the 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. Because 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: How precise is hashing as a proxy for comparing slices?* The section first gives a brief description of the hash function and then validates the use of comparing slice hash values in lieu of comparing actual slice content.

The use of hash values to represent slices reduces both the memory requirement and runtime, as it is no longer necessary to store or compare entire slices. The hash function, denoted H in Definition 2.6, uses XOR operations iteratively on the unique vertex IDs (of the SDG) which are included in a slice to generate a hash for the entire slice. We chose XOR as the hash operator because we do not have duplicate vertices in a slice and the order of the vertices in the slice does not matter.

A slice S is a set of SDG vertices $\{v_1, \ldots, v_n\}$ $(n \geq 1)$ and $\mathsf{id}(v_i)$ represents the unique vertex ID assigned by CodeSurfer to vertex v_i, where $1 \leq i \leq n$. The hash function H for S is defined as H_S, where

$$H_S = \oplus_{i=1}^{n} \mathsf{id}(v_i) \tag{2}$$

The remainder of this section presents a validation study of the hash function. The validation 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*: $US = |\{BSlice(x) : x \in V\}| + |\{FSlice(x) : x \in V\}|$. Note that if all vertices have the same backward slice then $\{BSlice(x) : x \in V\}$ is a singleton set. Finally, let UH be the number of *unique hash-values*, $UH = |\{H(BSlice(x)) : x \in V\}| + |\{H(FSlice(x)) : x \in V\}|$.

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 lists the programs. 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 in the strict case of *zero* tolerance for variation in slice contents (Binkley and Harman, 2005).

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 hash values matching those from another vertex with different slices is less than that of a single hash matching. 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 every 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, as an answer to *RQ2*, the hash-based approximation is found to be sufficiently accurate at 94.97% for slices and at 99.72% for clusters (for the studied programs). Thus, 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 and the following section consider the validation research question, *RQ3: How large*

Table 3
Hash function validation.

Program	Unique slices (US)	Unique hash values (UH)	Hashed slice precision (HSP)	Cluster count (CC)	Hash cluster count (HCC)	Hash Precision Clusters (HCP)
acct	1558	1521	97.63%	811	811	100.00%
barcode	2966	2792	94.13%	1504	1504	100.00%
bc	3787	3671	96.94%	1955	1942	99.34%
byacc	10,659	10,111	94.86%	5377	5377	100.00%
cflow	16,584	15,749	94.97%	8457	8452	99.94%
copia	3496	3398	97.20%	1785	1784	99.94%
ctags	8739	8573	98.10%	4471	4470	99.98%
diffutils	5811	5415	93.19%	2980	2978	99.93%
ed	2719	2581	94.92%	1392	1390	99.86%
findutils	9455	8734	92.37%	4816	4802	99.71%
garpd	808	769	95.17%	413	411	99.52%
indent	3639	3491	95.93%	1871	1868	99.84%
time	1453	1363	93.81%	760	758	99.74%
userv	3510	3275	93.30%	1827	1786	97.76%
wdiff	2190	2148	98.08%	1131	1131	100.00%
which	1213	1184	97.61%	619	619	100.00%
Sum	78,587	74,575	–	40,169	40,083	–
Average	4912	4661	94.97%	2511	2505	99.72%

are coherent clusters that exist in production source code and which patterns of clustering can be identified? 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.

To assess if a program includes a *large* coherent cluster, requires making a judgement concerning what threshold constitutes large. Following prior empirical work (Binkley and Harman, 2005; Harman et al., 2009; Islam et al., 2010a,b), 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.

Fig. 9 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 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

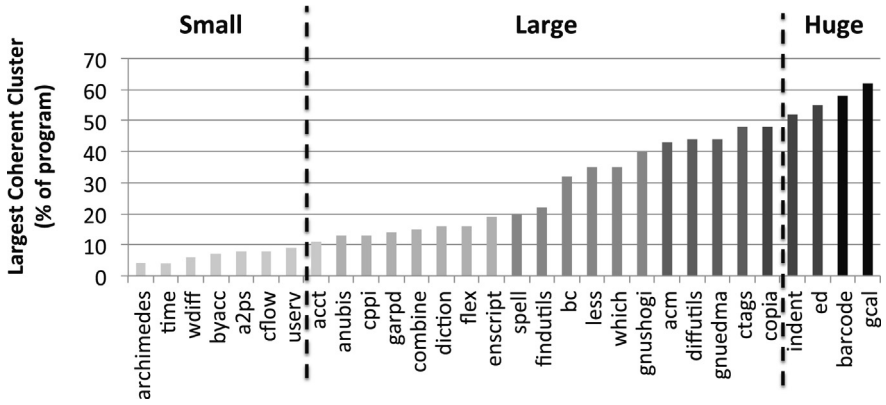


Fig. 9. Size of largest coherent cluster.

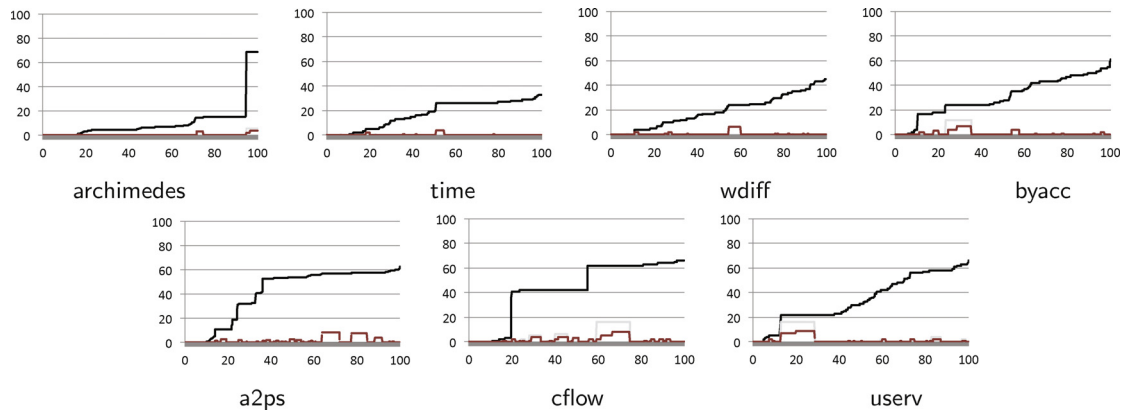


Fig. 10. Programs with *small* coherent clusters.

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%.

Therefore as an answer to first part of *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 the second part of *RQ3*. Figs. 10–12 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 Fig. 9.

Fig. 10 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.

Fig. 11 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 10–50% 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 Fig. 12. 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 second part of *RQ3*, 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 covering most of the program. Having shown that coherent clusters are prevalent in programs and that most programs have multiple significant clusters, the next section presents a series of four case studies that looks at how program structures are represented by these clusters.

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 *RQ4: Which structures within a program can coherent cluster analysis reveal?* As coherent clusters consist of program vertices that are mutually inter-dependent and share extra-cluster properties we consider such vertices of the cluster to be tightly coupled. It is our conjecture that these clusters likely represent logical structures representing a high-level functional decomposition of systems. This study will therefore look at how coherent clusters map to logical structures of the program.

The case studies 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 2600 LoC and 1558 SLoC from which CodeSurfer produced 2834 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.

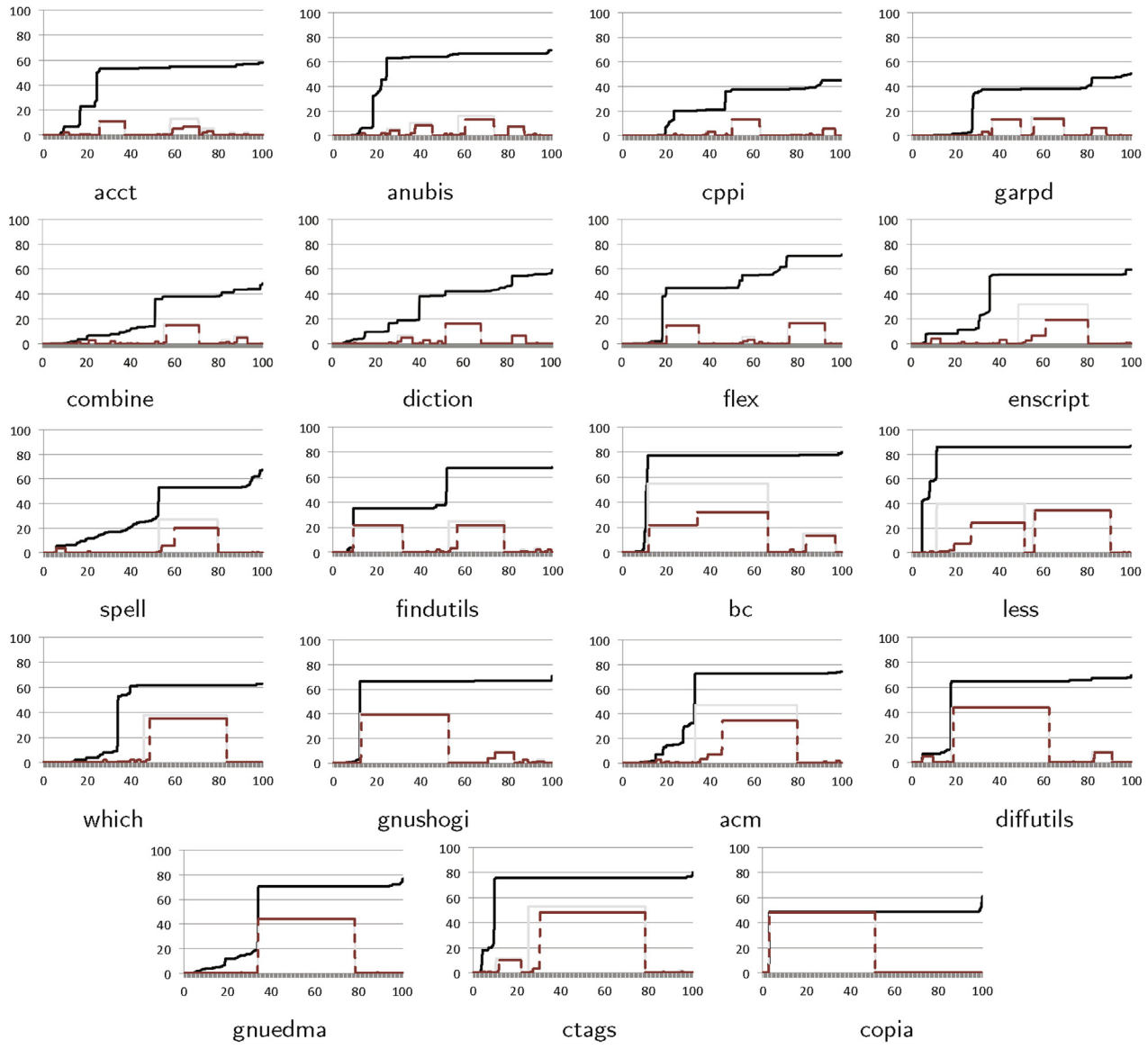


Fig. 11. Programs with *large* coherent clusters.

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 5th 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.

The B-SCG for acct (row one of Fig. 11) 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. This is because either same-backward-slice or same-forward-slice clusters only capture one of the two external properties captured by coherent clusters (Eq. (1)). In acct's B-SCG the vertices of the largest same-backward-slice cluster spanning the *x*-axis from 60%

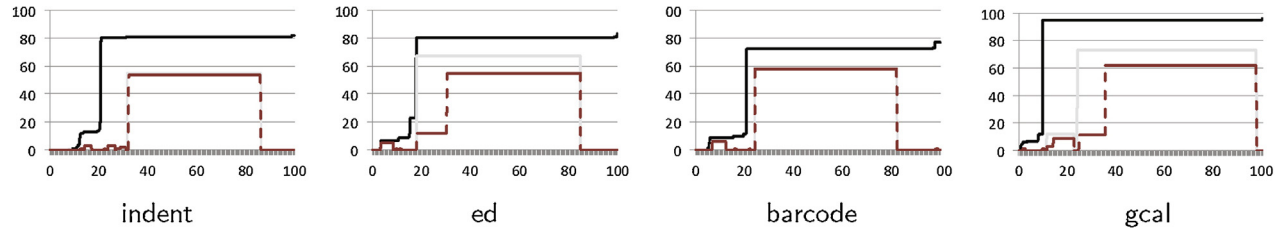


Fig. 12. Programs with *huge* coherent clusters.

Table 4
acct's top five clusters.

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 5
indent's top five clusters

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

to 75% are not part of the same coherent cluster. This is because the vertices do not share the same forward slice which is also a requirement for coherent clusters. This phenomenon is common in the programs studied and is found in both same-backward-slice and same-forward-slice clusters. This is another reason why coherent clusters are often smaller in size then same-slice clusters.

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 2nd largest cluster spans only a single file (hashtab.c), the 3rd largest cluster spans three files (file_rd.c, ac.c, and hashtab.c), the 4th largest cluster spans two files (ac.c and hashtab.c), while the 5th 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. Intuitively, it can be presumed that functions that have similar names or prefixes work together to provide some common functionality. In this case, six functions that have the same common prefix "hashtab" all perform operations on the hash table. However, these six 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. The clusters help identify functionality which is not obvious from the name of program artefacts such as functions and files. As an answer to RQ4, we find that in this case study each of the top five clusters maps to specific logical functionality.

3.6.2. Case study: indent

The next case study uses indent to further support the answer found for *RQ4* 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 *RQ4* 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 6978 LoC with 7543 vertices in the SDG produced by CodeSurfer. Table 5 shows statistics of the five largest clusters found in the program.

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.

Table 6
bc's top five clusters.

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

Clusters 3–5 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 structure as an answer to research question *RQ4*. However, in cases such as this where the internal functionality is tightly knit, a single large coherent cluster 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 9438 LoC and 5450 SLoC. The program has nine C files from which CodeSurfer produced 15,076 slices (backward and forward).

Analyzing bc's SCG (row 3, Fig. 11), 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. The Project View (Fig. 13) shows their distribution over the source files.

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.

As an answer to *RQ4*, 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, thus aiding in program comprehension. The remainder of this subsection illustrates a side-effect of *decluvi*'s multi-level visualization, how it can help find potential problems with the structure of a program.

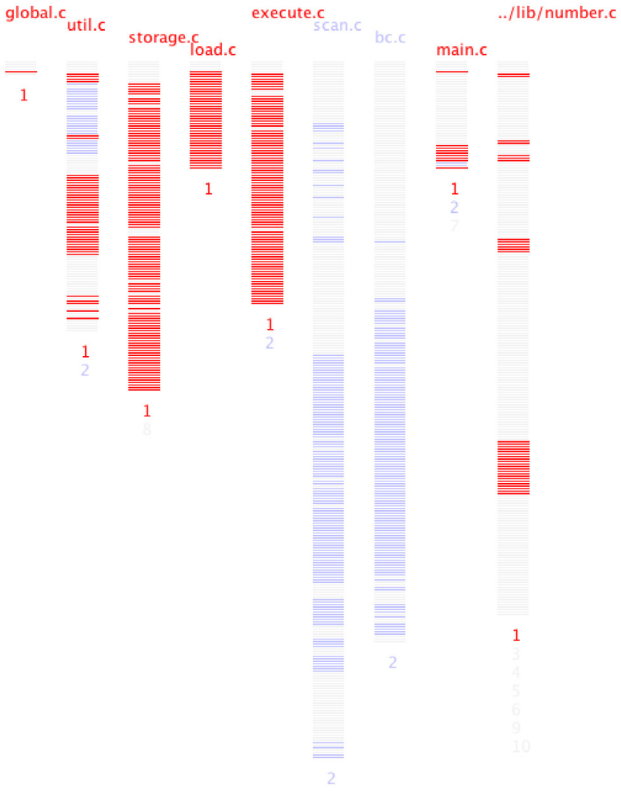


Fig. 13. *Decluvi*'s system view for the program bc showing each file using one column and each line of pixels summarizing *eight* source lines. Blue color (medium gray in black and white) represent lines whose vertices are part of smaller size clusters than those in red color (dark gray), while lines not containing any executable lines are always shown in light gray. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of the article.)

Util.c consists of small utility functions called from various parts of the program. This file contains code from Clusters 1 and 2 (Fig. 13). Five of the utility functions belong with Cluster 1, while six belong with Cluster 2. Furthermore, Fig. 14 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 Fig. 14). 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 Fig. 15 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 (Black, 2001).

Fig. 14. *Decluvi*'s 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 and white) represents lines with vertices belonging to the 2nd 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. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of the article.)

```
241:0|0/0          /* Initialize the code generator the parser. *
242:0|0/0
243:0|0/0          void
244:1|1/1          init_gen ()
245:0|0/0          {
246:0|0/0             /* Get things ready. */
247:2|1/1             break_label = 0;
248:2|1/1             continue_label = 0;
249:2|1/1             next_label  = 1;
250:20|257/1645       out_count = 2;
251:1|1/1             if (compile_only)
252:19|20/51             printf ("@i");
253:20|49/1645        else
254:1|1/1                init_load ();
255:1|1/1             had_error = FALSE;
256:2|1/1             did_gen = FALSE;
257:0|0/0          }
```

Fig. 15. *Decluvi*'s 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 and white) represents lines from the 2nd 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. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of the article.)

Table 7
copia's top five clusters.

Cluster	Cluster size		Files	Functions
Number	%	Vertices/lines	Spanned	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

Like other initialization functions, bc's init_gen is an exception to this guideline.

This case study not only provides support for the answer to research question *RQ4* found in previous case studies, 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 this series of case studies with 1168 LoC and 1111 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 6654 slices (backward and forward).

The B-SCG for copia is shown in Fig. 16a. 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 (Fig. 17) reveals an intriguing structure. There is a large block of code with the same spatial arrangement (bounded by the dotted black rectangle in Fig. 17) that belongs to the largest cluster of the program. It is unusual for so many consecutive source 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

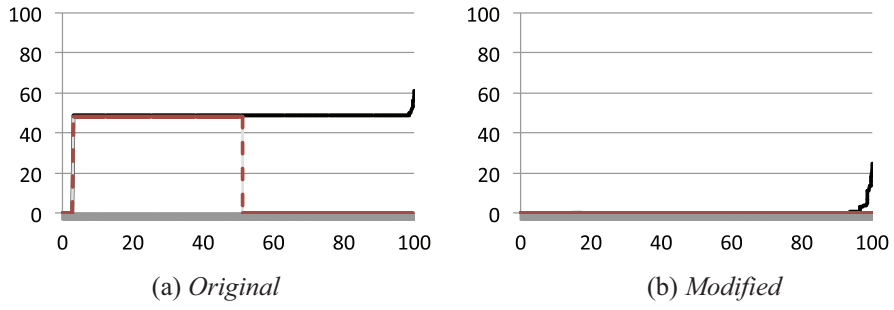


Fig. 16. SCGs for the program copia.

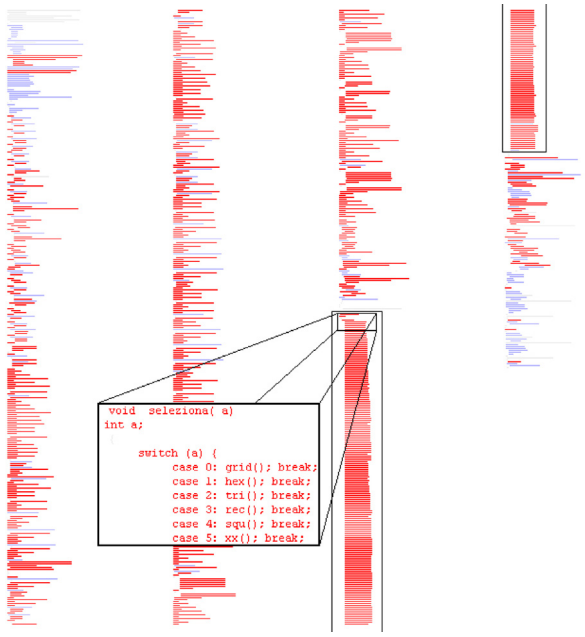


Fig. 17. *Decluvi*'s file view for the file copia.c of program copia. Each line of pixels represent the cluster data for one source code line. The lines in red (dark gray in black and white) are part of the largest cluster. The lines in blue (medium gray) are part of smaller clusters. A rectangle highlights the switch statement that holds the largest cluster together. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of the article.)

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 (Binkley and Harman, 2005), 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 Fig. 16b 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 re-factoring 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 reiterates the answer for *RQ4* 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 section addresses research question *RQ5: What are the implications of inter-cluster dependence between coherent clusters?* The question attempts to reveal whether there is dependence (slice inclusion) relationship between the vertices of different coherent clusters. A slice inclusion relationship between two clusters X and Y exist, if $\exists x \in X : BSlice(x) \cap Y \neq \emptyset$. If such containment occurs, it must be a strict containment relationship ($BSlice(x) \cap Y = Y$, see Eq. 1). Defining this relation using forward slices produces the inverse relation. 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 dependencies exist that allows entire clusters to depend on other clusters, then this dependence relationship can be used to group clusters to form a hierarchical decomposition of the system where coherent clusters are regarded as sub-systems, opening up the potential use of coherent clusters in reverse engineering. Secondly, if there are mutual dependency relations between clusters then such mutual dependency relationships can be used to provide a better estimate of slice-based clusters.

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 (Eq. 1). 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 (follows from Eq. 1).

Fig. 18 shows *Cluster Dependence Graphs* (CDG) for each of the four case study subjects. Only the five largest clusters of the case study subjects 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 5th largest cluster) and the directional edges denote

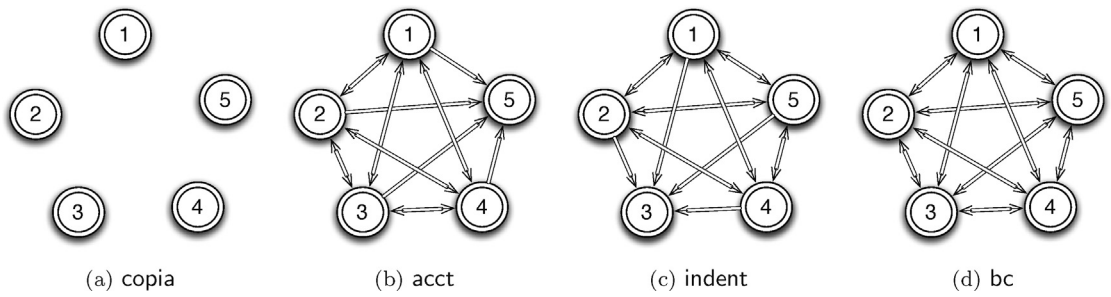


Fig. 18. Cluster dependence graphs.

Table 8
Various cluster statistics of bc.

Cluster number	Same backward-slice cluster size		Same forward-slice cluster size		Coherent cluster size	
	Vertices	%	Vertices	%	Vertices	%
1	4135	54.86	2452	32.52	2432	32.26
2	1111	14.74	1716	22.76	1655	21.96
3	131	1.74	1007	13.36	1003	13.31
4	32	0.42	157	2.08	117	1.55
5	25	0.33	109	1.45	102	1.35
Group size:						70.43

Table 9
Fault fixes for barcode.

Version	Release date	C files	LoC	SLoC	ELoC	SDG vertices	SDG edges	Total slices	Faults fixed
0.90	29-06-1999	6	1352	891	716	7184	23,072	3148	–
0.91	08-07-1999	6	1766	1186	949	8703	30,377	5328	5
0.92	03-09-1999	8	2225	1513	1221	10,481	37,373	5368	9
0.93	26-10-1999	8	2318	1593	1284	11,415	42,199	5722	5
0.94	26-10-1999	8	2318	1593	1284	11,414	41,995	5722	1
0.95	03-02-2000	8	2585	1785	1450	12,202	45,830	6514	3
0.96	09-11-2000	11	3249	2226	1799	14,733	56,802	8106	9
0.97	17-10-2001	13	3911	2670	2162	16,602	64,867	9530	2
0.98	03-03-2002	13	3968	2685	2177	16,721	65,395	9602	5

backward slice[2] 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 ($\forall x \in B : BSlice(x) \cap A = A$). Bi-directional edges show mutual dependencies, whereas uni-directional edges show dependency in one direction only. In the graph for copia (Fig. 18a), 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 (Fig. 18b) 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 (Fig. 18c), 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 (Fig. 18d), 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 grouping of the inter-dependent coherent clusters.

At first glance it may seem that the grouping 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 combined size of the group of top five inter-dependent coherent 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 taken together form a larger dependence structure, an estimate of a slice-based cluster.

As an answer to *RQ5*, this section shows that there are dependence relationships between coherent clusters and in some cases there are mutual dependences between large coherent clusters. It also shows that it may be possible to leverage this inter-cluster relationship to build a hierarchical system decomposition. Furthermore, groups of inter-dependent coherent clusters form larger dependence structures than same-slice clusters and provides a better approximation for slice-based clusters. This indicates that the sizes of dependence clusters reported by previous studies (Binkley et al., 2008; Binkley and Harman, 2005, 2009; Harman et al., 2009; Islam et al., 2010b) maybe conservative and mutual dependence clusters are *larger* and more prevalent than previously reported.

3.8. Dependence clusters and bug fixes

Initial work on dependence clusters advised that they might cause problems in software maintenance, and thus even be considered harmful, because they represent an intricate interweaving of mutual dependencies between program elements. Thus a large dependence cluster might be thought of as a bad code smell (Elssamadisy and Schalliol, 2002) or a anti-pattern (Binkley et al., 2008). Black et al. (2006) suggested that dependence clusters are

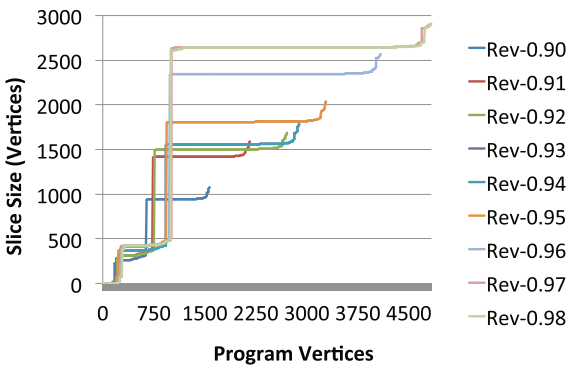


Fig. 19. Backward slice sizes for barcode releases.

[2] A definition based on forward slices will have the same results with reversed edges.

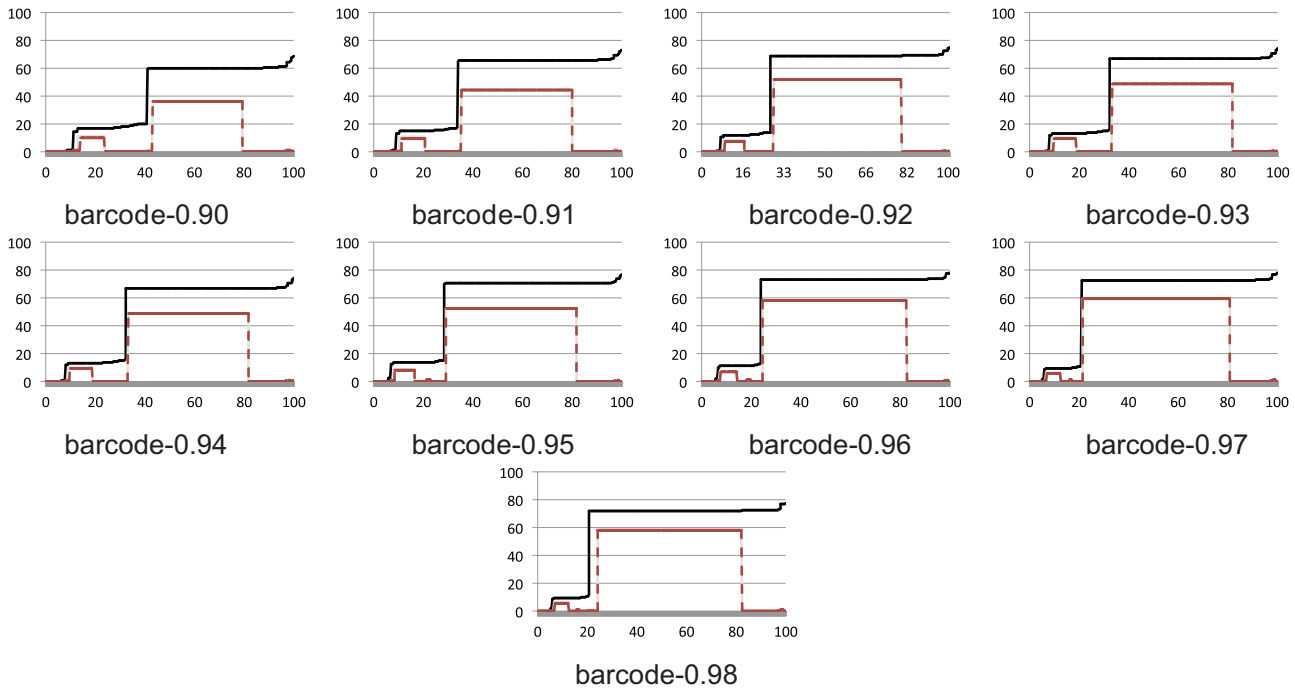


Fig. 20. BSCGs for various barcode versions.

potentially where bugs may be located and suggested the possibility of a link between clusters and program faults. This section further investigates this issue using a study that explores the relationship between program faults and dependence clusters. In doing so, it addresses research question *RQ6: How do program faults relate to coherent clusters?*

Barcode, an open source utility tool for converting text strings to printed bars (barcodes) is used in this study. A series of versions of the system are available for download from GNU repository.[3] There are nine public releases for barcode, details of which are shown in Table 9. Column 1 shows the release version, columns 3–6 show various metrics about the size of the system in terms of number of source files and various source code size measures. Columns 7–9 report the number of SDG vertices, SDG edges and the number of slices produced for each release. Finally, Column 10 reports the number of faults that were fixed since the previous release of the system. In Table 9 the size of barcode increases from 1352 lines of code in version 0.90 to 3968 lines of code in version 0.98. The total number of faults that were fixed during this time was 39.

Fault data, gathered by manually analyzing the publicly available version control repository[4] for the system, showed that total number of commits for barcode during these releases were 137. Each update was manually checked using CVSAnaly (Robles et al., 2004) to determine whether the update was a bug fix or simply an enhancement or upgrade to the system. Those commits that were identified as bug fixes were isolated and mapped to the release that contained the update. All the bug fixes made during a certain release cycle were then accumulated to give the total number of bugs fixed during a particular release cycle (Column 10 of Table 9). The reported number only includes bug fixes and does not include enhancement or addition of new functionality.

Fig. 19 shows the backward slice size plots for all versions of barcode in a single graph. The values of the axises in Fig. 19 are shown as vertex counts rather than relative values (percentages). This allows the growth of barcode to be easily visualized. From the plots it is seen that the size of the program increases progressively with each new release. The graphs also show that a significant number of vertices in each revision of the program yields identical backward slices and the proportion of vertices in the program that have identical backward slices stays roughly the same. Overall, the profile of the clusters and slices remains consistent. The graph also shows that the plots do not show any significant change in their overall shape or structure. Interestingly, the plot for version 0.92 with 9 fault fixes is not different in shape from revision 0.94 where only a single fault was fixed.

As coherent clusters are composed of both backward and forward slices, the stability of the backward slice profile itself does not guarantee the stability of coherent cluster profile. The remainder of this section looks at how the clustering profile is affected by bug fixes. Fig. 20 shows individual SCGs for each version of barcode. As coherent clusters are dependent on both backward and forward slices, such clusters will be more sensitive to changes in dependences within the program. The SCGs show that from the initial version barcode-0.90 there were two coherent clusters in the system. The smaller one is around 10% of the code while the larger is around 40% of the code. As the system evolved and went through various modifications and enhancements, the number of clusters and the profile of the clusters remained consistent other than its scaled growth with the increase in program size. It is also evident that during evolution of the system, the enhancement code or newly added code formed part of the larger cluster. This is why in the later stages of the evolution we see an increase in the size of the largest cluster, but not the smaller one.

However, we do not see any significant changes in the slice and cluster profile of the program that can be attributed to bug fixes. For example, the single bug fixed between revisions 0.93 and 0.94 was on a single line of code from the file code128.c. The changes to the line is shown in Fig. 21 (in version 0.93 there is an error in calculating the checksum value, which was corrected in version 0.94). As illustrated by this example, the data and control flow of the program and thus the dependencies between program points

[3] http://gnu.mirror.iweb.com/gnu/barcode/.
[4] cvs.savannah.gnu.org:/sources/barcode.

```
barcode-0.93, code128.c, line 139
    checksum += code * i+1;

barcode-0.94, code128.c, line 139
    checksum += code *(i+1);
```

Fig. 21. Bug fix example.

are not affected by the bug fix and hence no change is observed between the SCGs of the two releases (Fig. 20).

If dependence clusters correlated to faults, or, if dependence clusters were directly related to the number of faults in a program, then a significant difference would be expected in the shape of the SCG when faults were rectified. The SCGs for program barcode (Fig. 20) show no change in their profile when faults within the program are fixed. This provides evidence that faults may not be dictated by the presence or absence of dependence clusters. As an answer to *RQ6*, the study of barcode finds no correlation between the existence of dependence clusters and program faults and their fix. We have to be careful in generalising the answer to this question because of the small dataset considered in this study, further extended research is needed to derive a more generalised answer. Moreover, this does not exclude the possibility that most program faults occur in code that are part of large clusters. In future we plan to extend this experiment in a qualitative form to study whether program faults lie within large or small clusters, or outside them altogether.

3.9. Clusters and system evolution

The previous section showed that for barcode the slice and cluster profiles remain quite stable through bug fixes during system evolution and its growth of almost 2.5 times over a period of 3 years. This section extends that study by looking for cluster changes during system evolution. It addresses *RQ7: How stable are coherent clusters during system evolution?* using longitudinal analysis of the case studies presented earlier. From the GNU repository we were able to retrieve four releases for bc, four releases for acct and 14 releases for indent. As copia is an industrial closed-source program, we were unable to obtain any previous versions of the program and thus the program is excluded from this study.

The graphs in Fig. 22 show backward slice size overlays for every version of each program. Fig. 22a and c for bc and indent show that these systems grow in size during its evolution. The growth is more prominent in indent (Fig. 22c) where the program grows from around 4800 vertices in its initial version to around 7000 vertices in the final version. The growth for bc is smaller, it grows from around 6000 vertices to 7000 vertices. This is partly because the versions considered for bc are all minor revisions. For both bc and indent the slice-size graphs show very little change in their profile. The graphs mainly show a scale up that parallels the growth of the system.

For acct (Fig. 22b) the plots do not simply show a scale up but show a significant difference. In the 4 plots, the revisions that belong to the same major release are seen to be similar and show a scaling, whereas those from different major releases show very different landscapes. The remainder of this section gives detail of these clustering profile changes.

Fig. 23 shows the BSCGs for the four versions of bc. Initially, the backward slice size plots (solid black lines) show very little difference. However, upon closer inspection of the last three versions we see that the backward slice size plot changes slightly at around the 80% mark on the *x*-axis. This is highlighted by the fact that the later three versions show an additional coherent cluster spanning from 85% to 100% on the *x*-axis which is absent from the initial release. Upon inspection of the source code changes between versions bc-1.03 and bc-1.04 the following types of updates were found:

1 bug fixes,
2 addition of command line options,
3 reorganization of the source tree, and
4 addition of new commands for dc.

The reorganization of the program involved significant architectural changes that separated out the code supporting bc's related dc functionality into a separate hierarchy and moved files common to both bc and dc to a library. This refactoring of the code broke up the largest cluster into two clusters, where a new third cluster is formed as seen in the SCG. Thus, the major restructuring of the code between revisions 1.03 and 1.04 causes a significant change in the cluster profile. Almost no other change is seen in the cluster profile between the remaining three bc revisions 1.04, 1.05, and 1.06, where no significant restructuring took place.

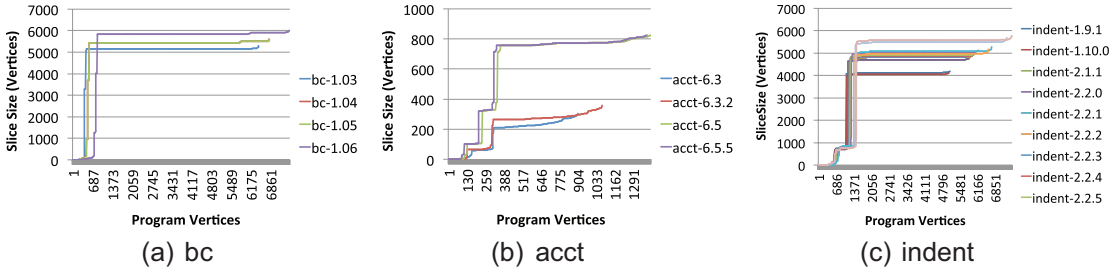


Fig. 22. Backward slice size plots for multiple releases.

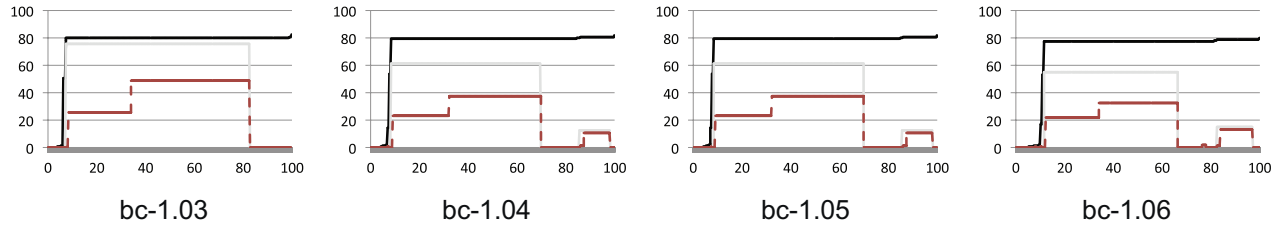


Fig. 23. BSCGs for various bc versions.

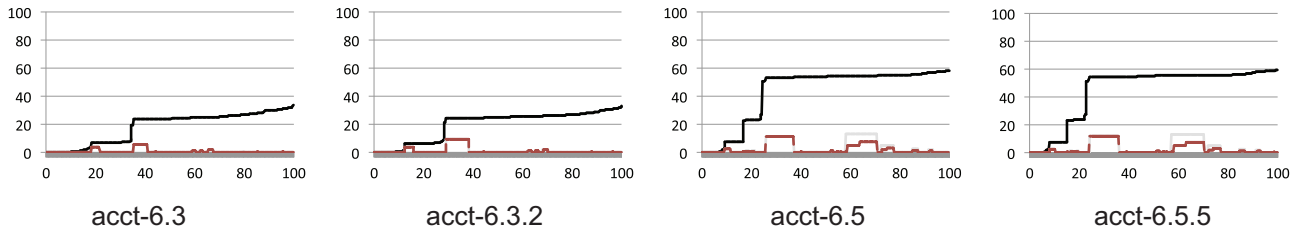


Fig. 24. BSCGs for various acct versions.

Fig. 24 shows the SCGs for the four versions of acct considered in this study. The slice profile and the cluster profile show very little change between acct-6.3 and acct-6.3.2. Similarly, not much change is seen between acct-6.5 and acct-6.5.5. However, the slice and the cluster profiles change significantly between major revisions, 6.3.X and 6.5.X. The change log of release 6.5 notes "Huge code-refactoring." The refactoring of the code is primarily in the way system log files are handled using utmp_rd.c, file_rd.c, dump-utmp.c and stored using hash tables whose operations are defined in hashtab.c and uid_hash.c.

Finally, Fig. 25 shows the SCGs for the 14 versions of indent. These revisions include two major releases. It is evident from the SCGs that the slice profile during the evolution hardly changes. The cluster profile also remains similar through the evolution. The system grows from 4466 to 6521 SLoC during its evolution which is supported by Fig. 22c showing the growth of the system SDG size. Indent is a program for formatting C programs. A study of the change logs for indent did not reveal any major refactoring or restructuring. The changes to the system were mostly bug fixes and upgrades to support new command line options. This results in almost negligible changes in the slice and cluster profiles despite the system evolution and growth.

As an answer to *RQ7*, this study finds that unless there is significant refactoring of the system, coherent cluster profiles remain stable during system evolution and thus captures the core architecture of the program in all three case studies. Future work will replicate this longitudinal study on a large code corpus to ascertain whether this stability holds for other programs.

3.10. Threats to validity

This section presents threats to the validity of the results presented. Threats to three types of validity (external, internal and construct) are considered. 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,

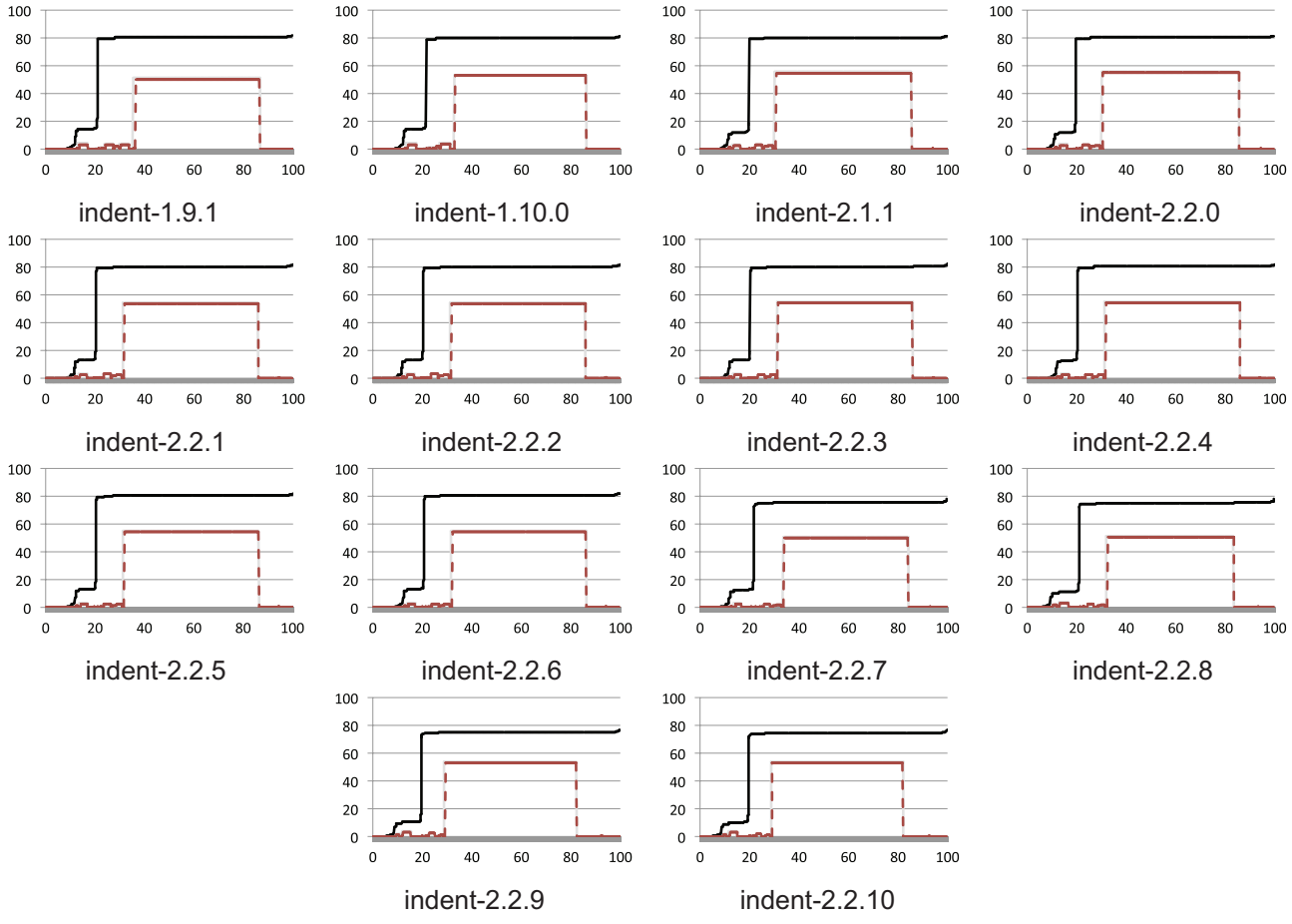


Fig. 25. BSCGs for various indent versions.

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 from various domains. 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 or aspect-oriented programming.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variable. The use of hash values to approximate slice content during clustering is a source of potential internal threat. The approach assumes that hash values uniquely identify slice contents. Hash functions are prone to hash collision which in our approach can cause clustering errors. The hash function used is carefully crafted to minimize collision and its use is validated in Section 3.3. Furthermore, the identification of logical structure in programs were done by the authors of the paper who are not involved in the development of any of the case study subjects. This brings about the possibility that the identified structures do not represent actual logical constructs of the programs. As the case studies are Unix utilities, their design specification are not available for evaluation. Future work will entail consultation with the development team of the systems to further validate the results.

Construct validity refers to the validity that observations or measurement tools actually represent or measure the construct being investigated. In this paper, one possible threat to construct 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. An overly conservative, and therefore imprecise, analysis would tend to increase the levels of dependence and potentially also increase the size of clusters. There is no automatic way to tell whether a cluster arises because of imprecision in the computation of dependence or whether it is 'real'. Section 3.2 discusses the various pointer analysis settings and validates its precision. CodeSurfer's most precise pointer analysis option was used for the study.

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 (Ali et al., 2010). 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 (Black, 2001) or by attempting to identify portions of code for which a change can be safely performed free from side effects (Gallagher and Lyle, 1991; Tonella, 2003). A recently proposed impact analysis framework (Acharya and Robinson, 2011) 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. There has also been recent work on finding dependence communities in software (Hamilton and Danicic, 2012) where social network community structure detection algorithms are applied to slice-inclusion graphs to identify communities.

Dependence clusters have previously been linked to software faults (Black et al., 2006) and have been identified as a potentially harmful 'dependence anti-pattern' (Binkley et al., 2008). 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 (Binkley and Harman, 2005; Harman et al., 2009) 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 (Binkley et al., 2009). Other work on dependence clusters in software engineering has considered clusters at both low-level (Binkley and Harman, 2005; Harman et al., 2009) (SDG based) and high-level (Eisenbarth et al., 2003; Mitchell and Mancoridis, 2006) (models and functions) abstractions.

This paper extends our previous work which introduced coherent dependence clusters (Islam et al., 2010b) and *decluvi* (Islam et al., 2010a). 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 slice approximation from 78% to 95%, 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. We also introduce the notion of inter-cluster dependence which will form the base of reverse engineering efforts in future. Finally, we also present studies which show the lack of correlation between coherent clusters and bug fixes and show that coherent clusters remain surprisingly stable during system evolution.

In some ways our work follows the evolutionary development of the study of software clones (Bellon et al., 2007), 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 (Bouktif et al., 2006; Kapser and Godfrey, 2008).

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 research, 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* (Eick et al., 1992) 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 (Baker and Eick, 1995) 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 (Storey et al., 1997).

5. Summary and future work

Previous work has deemed dependence clusters to be problematic as they inhibit program understanding and maintenance. This paper views them in a new light, it introduces and evaluates a specialized form of dependence cluster: the coherent cluster. Such clusters have vertices that share the same internal and external dependencies. The paper shows that such clusters are not necessarily problems but rather can aid an engineer understand program components and their interactions. Developers can exploit knowledge of coherent clusters as they aid in program comprehension as the clusters bring out interactions between logical constructs of the system. We also lay a foundation for research into this new application area and encourage further research. Moreover, future research could compare the aspects of various definitions of dependence clusters and the properties they capture.

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 a logical functional decomposition 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.

The paper also discusses inter-cluster dependence and how mutual dependencies between clusters may be exploited to reveal large dependence structure that form the basis of reverse engineering efforts. Furthermore, the paper presents a study on how bug fixes relate to the presence of coherent clusters, and finds no relationship between program faults and coherent clusters in barcode. Finally, a longitudinal study of three subjects shows that coherent clusters remain surprisingly stable through system evolution.

The paper is one of the first in the area of dependence clusters to suggest that dependence clusters (coherent clusters) are not problematic but represent program structure and give evidence to that cause. Future work in this area is rife with opportunities beginning with enabling the use of coherent clusters in a program comprehension and reverse engineering tools. The inter-cluster dependence study lays out the ground work in this context. There is also room for further research aimed at understanding the formation and impact of coherent clusters on software quality. For example, by studying how well dependence clusters can capture functionality. Furthermore, application of dynamic slicing in formation of dependence clusters might by considered as static analysis can suffer from over approximation caused by its conservative nature.

Acknowledgements

This work is supported by EPSRC (EP/G060525/2, EP/F059442/2), EU (ICT-2009.1.2 no 257574), and NSF (CCF 0916081). Data from the EPSRC-funded portions of this work may be available by contacting Dr. Krinke. Please note that intellectual property or other restrictions may prevent the full disclosure of this data.

References

Acharya, M., Robinson, B., 2011. Practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the 33rd International Conference on Software Engineering, ACM Press, pp. 746–755.
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. IEEE Transactions on Software Engineering 36 (6), 742–762.
Andersen, L.O., 1994. Program analysis and specialization for the C programming language, Ph.D. thesis, DIKU, University of Copenhagen, (DIKU report 94/19).
Anderson, P., Teitelbaum, T., 2001. Software inspection using CodeSurfer. In: First Workshop on Inspection in Software Engineering, pp. 1–9.
Baker, M.J., Eick, S.G., 1995. Space-filling software visualization. Journal of Visual Languages & Computing 6 (2), 119–133.
Bastian, M., Heymann, S., Jacomy, M., 2009. Gephi: An open source software for exploring and manipulating networks. In: International AAAI Conference on Weblogs and Social Media. AAAI Press.
Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E., 2007. Comparison and evaluation of clone detection tools. IEEE Transactions on Software Engineering 33 (9), 577–591.
Beszédes, Á., Gergely, T., Jász, J., Toth, G., Gyimóthy, T., Rajlich, V., 2007. Computation of static execute after relation with applications to software maintenance. In: 23rd IEEE International Conference on Software Maintenance, October. IEEE Computer Society Press, pp. 295–304.
Beyer, D., 2008. CCVisu: automatic visual software decomposition. In: Companion of the 30th International Conference on Software Engineering. ACM Press, pp. 967–968.
Binkley, D., 2007. Source code analysis: A road map. In: FOSE '07: 2007 Future of Software Engineering. IEEE Computer Society Press, pp. 104–119.
Binkley, D., Gold, N., Harman, M., Li, Z., Mahdavi, K., Wegener, J., 2008. Dependence anti patterns. In: 4th International ERCIM Workshop on Software Evolution and Evolvability (Evol'08)., pp. 25–34.
Binkley, D., Harman, M., 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, pp. 44–53.
Binkley, D., Harman, M., 2005. Locating dependence clusters and dependence pollution. In: 21st IEEE International Conference on Software Maintenance. IEEE Computer Society Press, pp. 177–186.
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.
Binkley, D., Harman, M., Hassoun, Y., Islam, S., Li, Z., 2009. Assessing the impact of global variables on program dependence and dependence clusters. Journal of Systems and Software 83 (1), 96–107.
Binkley, D.W., Harman, M., Krinke, J., 2007. Empirical study of optimization techniques for massive slicing. ACM Transactions on Programming Languages and Systems 30, 3:1–3:33.
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 Press, pp. 3–10.
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.
Black, S.E., 2001. Computing ripple effect for software maintenance. Journal of Software Maintenance and Evolution: Research and Practice 13 (4), 263–279.
Bomze, I.M., Budinich, M., Pardalos, P.M., Pelillo, M., 1999. The maximum clique problem. In: Handbook of Combinatorial Optimization. Springer, US, pp. 1–74.
Bouktif, S., Antoniol, G., Merlo, E., Neteler, M., 2006. A novel approach to optimize clone refactoring activity. GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation, vol. 2. ACM Press, pp. 1885–1892.
Eick, S., Steffen, J., Sumner, E., 1992. Seesoft – A tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering 18 (11), 957–968.
Eisenbarth, T., Koschke, R., Simon, D., 2003. Locating features in source code. IEEE Transactions on Software Engineering 29 (3), 210–224.
Elssamadisy, A., Schalliol, G., 2002. Recognizing and responding to "bad smells" in extreme programming. In: International Conference on Software Engineering. ACM Press, pp. 617–622.
Fahndrich, M., Foster, J.S., Su, Z., Aiken, A., 1998. Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation. ACM Press, pp. 85–96.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9 (3), 319–349.
Gallagher, K.B., Lyle, J.R., 1991. Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17 (8), 751–761.
Hajnal, Á., Forgács, I., 2011. A demand-driven approach to slicing legacy COBOL systems. Journal of Software: Evolution and Process 24 (1), 67–82.
Hamilton, J., Danicic, S., 2012. Dependence communities in source code. In: 28th IEEE International Conference on Software Maintenance, pp. 579–582.
Harman, M., Binkley, D., Gallagher, K., Gold, N., Krinke, J., 2009. Dependence clusters in source code. ACM Transactions on Programming Languages and Systems 32 (1), 1:1–1:33.
Horwitz, S., Reps, T., Binkley, D., 1990. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12 (1), 26–60.
Islam, S., Krinke, J., Binkley, D., 2010a. Dependence cluster visualization. In: SoftVis'10: 5th ACM/IEEE Symposium on Software Visualization. ACM Press, pp. 93–102.
Islam, S., Krinke, J., Binkley, D., Harman, M., 2010b. Coherent dependence clusters. In: PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM Press, pp. 53–60.
Kapser, C., Godfrey, M.W., 2008. "Cloning considered harmful" considered harmful: patterns of cloning in software. Empirical Software Engineering 13 (6), 645–692.
Krinke, J., 1998. Static slicing of threaded programs. In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 35–42.
Krinke, J., 2002. Evaluating context-sensitive slicing and chopping. In: IEEE International Conference on Software Maintenance. IEEE Computer Society Press, pp. 22–31.
Krinke, J., 2003. Context-sensitive slicing of concurrent programs. In: Proceedings of the 9th European Software Engineering Conference, ACM Press, pp. 178–187.
Mitchell, B.S., Mancoridis, S., 2006. On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering 32 (3), 193–208.
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.
Ren, X., Chesley, O., Ryder, B.G., 2006. Identifying failure causes in Java programs: an application of change impact analysis. IEEE Transactions on Software Engineering 32 (9), 718–732.
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. ACM Press, pp. 664–665.
Robles, G., Koch, S., Gonzalez-Barahona, J., 2004. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In: Proceedings of Second International Workshop on Remote Analysis and Measurement of Software Systems, IEE, pp. 51–55.
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.
Shapiro, M., Horwitz, S., 1997. The effects of the precision of pointer analysis. In: Static Analysis Symposium. Vol. 1302 of Lecture Notes in Computer Science. Springer Berlin, Heidelberg, pp. 16–34.
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. ACM Press, pp. 606–607.
Szegedi, A., Gergely, T., Beszédes, Á., Gyimóthy, T., Tóth, G., 2007. Verifying the concept of union slices on Java programs. In: 11th European Conference on Software Maintenance and Reengineering, pp. 233–242.
Tonella, P., 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Transactions on Software Engineering 29 (June (6)), 495–509.
Weiser, M., 1984. Program slicing. IEEE Transactions on Software Engineering 10 (4), 352–357.
Wheeler, D.A., 2004. SLOC Count User's Guide. http://www.dwheeler.com/sloccount/sloccount.html
Yau, S.S., Collofello, J.S., 1985. Design stability measures for software maintenance. IEEE Transactions on Software Engineering 11 (September (9)), 849–856.

Syed Islam is a Research Associate in the Software Systems Engineering Group at University College London, he is also a part of the CREST centre. His interests are in static program analysis, particularly in program slicing and software clustering. His other research interests include Search Based Software Engineering (SBSE) and Automatic Bug Assignment.

Jens Krinke is Senior Lecturer in the Software Systems Engineering Group at the University College London, where he is Deputy Director of the CREST centre. He is well known for his work on program slicing; current research topics include program analysis for software engineering purposes, in particular dependence analysis for software security, and clone detection and its use in code provenance. Before joining the University College London, he was at King's College London and the FernUniversität in Hagen, Germany, where he worked on aspect mining and e-learning applications for distant teaching of software engineering.

Dr. David Binkley is a Professor of Computer Science at Loyola University Maryland where he has worked since earning his doctorate from the University of Wisconsin in 1991. From 1993 to 2000, Dr. Binkley was a visiting faculty researcher at the National Institute of Standards and Technology (NIST), where his work included participating in the Unravel program slicer project. While on leave from Loyola in 2000, he worked with Grammatech Inc. on the System Dependence Graph (SDG) based slicer CodeSurfer and in 2008 he joined the researchers at the Crest Centre of Kings' College London to work on dependence cluster analysis. Dr. Binkley's current NSF funded research focuses on semantics-based software engineering tools, the application of information retrieval techniques in software engineering, and improved techniques for software testing. In 2014 he will co-chair the program for Software Evolution Week which joins The Working Conference on Reverse Engineering (WCRE) and The European Conference on Software Maintenance and Reengineering (CSMR).

Mark Harman is professor of Software Engineering in the Department of Computer Science at University College London, where he directs the CREST centre and is Head of Software Systems Engineering. He is widely known for work on source code analysis and testing and was instrumental in founding the field of Search Based Software Engineering (SBSE). SBSE research has rapidly grown over the past five years and now includes over 1000 authors, from nearly 300 institutions spread over more than 40 countries. A recent tutorial paper on SBSE can be found here: http://www.cs.ucl.ac.uk/staff/mharman/laser.pdf.