# Empirical Investigation of SEA-Based Dependence Cluster Properties

Árpád Beszédes[a,*], Lajos Schrettner[a], Béla Csaba[b], Tamás Gergely[a], Judit Jász[a], Tibor Gyimóthy[a]

[a]*Department of Software Engineering, University of Szeged, Hungary*
[b]*Department of Set Theory and Mathematical Logic, University of Szeged, Hungary*

**Abstract**

Dependence clusters are (maximal) groups of source code entities that each depend on the other according to some dependence relation. Such clusters are generally seen as detrimental to many software engineering activities, but their formation and overall structure are not well understood yet. In a set of subject programs from moderate to large sizes, we observed frequent occurrence of dependence clusters using Static Execute After ($SEA$) dependences ($SEA$ is a conservative yet efficiently computable dependence relation on program procedures). We identified potential *linchpins*; these are procedures that can primarily be made responsible for keeping the cluster together. Furthermore, we found that as the size of the system increases, it is more likely that multiple procedures are jointly responsible as sets of linchpins. We also give a heuristic method based on structural metrics for locating possible linchpins as their exact identification is unfeasible in practice, and presently there are no better ways than the brute-force method. We defined novel metrics to be able to uncover clusters of different sizes in programs, and also to relate programs in terms of their degree of clusterization. Finally, we present a possible application of $SEA$-based dependences in change impact analysis, and investigate the effect of dependence clusters on the successfulness of this activity.

*Keywords:* Source code dependence analysis, Dependence clusters, Linchpins and linchpin sets, Static Execute After, Change impact analysis

## 1. Introduction

Dependences in computer programs are natural and inevitable. We can talk about dependences among any kind of artifacts such as requirements, design

*Corresponding author

*Email addresses:* beszedes@inf.u-szeged.hu (Árpád Beszédes), schrettner@inf.u-szeged.hu (Lajos Schrettner), bcsaba@math.u-szeged.hu (Béla Csaba), gertom@inf.u-szeged.hu (Tamás Gergely), jasy@inf.u-szeged.hu (Judit Jász), gyimothy@inf.u-szeged.hu (Tibor Gyimóthy)

elements, program code or test cases, but dependences within the source code capture the physical structure as implemented best. A dependence between two program elements (*e.g.* statements or procedures) basically means that the execution of one element can influence that of the other, hence the software engineer should be aware of this connection in virtually any software engineering task involving the two elements. One of the fundamental tasks of program analysis is to deal with source code entities and the dependences between them [1].

Dependences cannot be avoided, but they do not always reflect the original complexity of the problem. Sometimes unnecessary complexity is injected into the implementation, which may cause significant problems. A relatively new research area explores *dependence clusters* in program code, which are defined as maximal sets of program elements that each depend on the other [2]. The current view is that large dependence clusters are detrimental to the software development process; in particular, they hinder many different activities including maintenance, testing and comprehension [3, 4, 5, 6, 7]. The primary problem is that in any dependence-related examination, encountering any member of a cluster forces us to enumerate all other cluster members. If large clusters covering much of the program code exist in a system, then it is very likely that one cluster member is encountered and consequently a large portion of the program code should be considered eventually.

The root causes of this phenomenon are not well understood yet; it seems to be an inherent property of program code dependence relationships. As apparently dependence clusters cannot be easily avoided in the majority of cases, research should be focused on understanding the causes for the formation of clusters, and the possibilities for their removal or reduction. Previous work revealed that in many cases a highly focused part of the software can be deemed responsible for the formation of dependence clusters [4, 5, 8]. Namely, program elements called *linchpins* are seen as central in terms of dependence relations, and are often holding together the whole program. If the linchpin is ignored when following dependences, clusters will vanish, or at least decrease considerably. To get an initial idea of the linchpin concept, consider our experimental program 'compress' from Section 5, whose elements and their dependences are shown in Figure 11. Initially, two large clusters are formed in this program as outlined by the two outer rectangles, but after removing the function *compress*, one of the clusters will be disintegrated into a number of smaller ones as indicated by the inner rectangles. The general approach to define a linchpin is to find a program element whose removal results in the largest decrease of clusters according to a given metric.

Of course, it is useful if one is aware of such linchpins, let alone be able to remove them by refactoring the program. However, currently even the first step (identifying linchpins) is largely an unexplored area. We still do not understand fully what makes a particular program point a linchpin, how they can be identified, or whether there is always a single element to be made responsible in the first place. The possibilities for linchpin removal by program refactoring are even harder to assess. Sometimes, dependence clusters are avoidable because they actually introduce unnecessary complexity to the implementation; this is

what Binkley *et al.* call "dependence pollution" [2]. In such cases the program can be refactored using reasonable effort, but this is not always the case.

In this work, we present the results of our empirical investigation of dependence clusters in a range of programs of moderate (up to 200 kLOC) and large sizes. In the latter category we investigated two industrial size open source software systems, the GCC compiler [9] and the WebKit web browser engine [10], each consisting of over a million LOC.

We are dealing with procedure-level program dependences computed using the *Static Execute After (SEA)* approach (on C/C++ functions and methods). The $SEA$ relation between two procedures is a conservative type of dependence that takes into account the possible control-flow paths and call-structures in the program elements [11]. $SEA$-based dependences can be used, among others, in software change impact analysis [12, 13]. The main advantage of $SEA$ is that it achieves acceptable accuracy, yet can efficiently be computed even for large systems of millions of LOC. We computed $SEA$-based dependences of all procedures in our subject programs and investigated the resulting dependence clusters in terms of their frequency of occurrence and by identifying potential linchpins in them.

We identified possible linchpins for all programs in the moderate size category using a naïve approach that enumerates all possibilities. We argue that as the size of the programs increases, it is increasingly less probable that only a single program element can be identified as a linchpin, rather a certain set of elements should be treated as such. Next, we investigated the feasibility of using a simple heuristic to identify the linchpins which takes into account local properties of the program elements (procedures in our case). We found that the number of outgoing invocations from a procedure was quite a good estimator for linchpins. Finally, we present a possible application of $SEA$ dependence sets and related dependence cluster analysis in impact analysis, and verify the effect of linchpins on the successfulness of impact analysis considering real software failures of our largest subject program WebKit.

A previous version of this paper was presented at the 2013 Conference on Source Code Analysis and Manipulation [14]. The current version extends the conference paper with more background material on the definitions, the subject programs, measurements and analyses, as well as theoretical analysis of the clusterization metrics. Further, we now use two kinds of cluster definitions using both backward and forward dependences, and relate them from theoretical and empirical point of view. A new research question has also been added augmenting a previous result by considering the effect of linchpin removal on the prediction capability of $SEA$-based impact analysis.

### *1.1. Summary*

The research questions in this article are the investigation of $SEA$-based dependence clusters and possible linchpins in our subject programs, heuristic methods for linchpin identification and in particular an application of the approach in impact analysis. More precise description of these can be found at the end of Section 2, after the necessary technical background has been introduced.

Our findings are summarized as follows:

- We introduce the term *clusterization* to indicate the extent programs exhibit dependence clustering, and define novel metrics that characterize this property quantitatively.
- We computed $SEA$-based dependence clusters for realistic size programs. Among the moderate-size ones there were many clusters, but only one of the big programs included significant clusters, which is an interesting result.
- We were able to identify linchpin elements – ones whose removal results in the largest decrease in clusterization – using a naïve approach that enumerates all possibilities. In many cases however, especially with the big programs, it is not to be expected that only one program element (procedure) is responsible for the formation of clusters.
- We give a heuristic method based on local procedure metric for linchpin approximation. We found that the number of outgoing invocations from a procedure was quite a good estimator for linchpins.
- After reducing the overall dependence set sizes significantly by removing the possible linchpins in WebKit, in many of the cases we observed no change in the prediction capability of impact analysis using $SEA$.

The rest of the paper is organized as follows. Sections 2 and 3 provide relevant background information about the motivation, related work and our experimental environment. Cluster identification is discussed in Section 4, while the topics related to linchpin determination are given in Section 5. Section 6 deals with cluster-related investigations in WebKit including impact analysis. Section 7 discusses threats to validity, and finally we conclude in Section 8.

## 2. Background, motivation and goals

### *2.1. Previous and related work*

The phenomenon of *dependence clusters* was first described by Binkley and Harman in 2005 [2] based on program slices and Program Dependence Graphs (PDG) [15, 16]. Initially, they were defined as maximal sets of statements that all have the same backward slice, which also means that the elements of a dependence cluster each depend on the other. So the number of elements in a cluster and their dependence size (which is at least the size of the cluster) are two important properties of such formations. Later, the definition using dependence set coincidence has been substituted with the approximation of checking only the sizes of the dependence sets, which turned out to be a practically usable approach [6].

The notion of dependence clusters can be generalized to other kinds of dependence types and different program elements at various granularity. Furthermore,

it seems that dependence clusters are independent of the programming language and the type of the system [3, 17, 18]. An interesting approach to locate interrelated program elements in programs is based on applying community structure analysis on software dependence graphs [19, 20]. A set of program elements are treated to form a community if the number of internal connections is more than expected given the overall distribution of the connections in the whole program. It is still an open question how the identified community structures relate to dependence clusters.

In a recent work, Islam *et al.* [21] introduced *coherent clusters* which combine the backward and forward slice-based dependence clusters, and which may be better predictors of logical functionality of the program.

The current view is that large dependence clusters hinder many different software engineering activities, including impact analysis, maintenance, program comprehension and software testing [3, 6, 7]. It has been suggested that large dependence clusters leading to "dependence pollution" should be refactored [2, 7], but for such opportunities the identification of the dependence cluster causes is essential. Specifically, the identification and possible removal of *linchpins*, the directly responsible program elements, is an active research area. Virtually, the only existing approach to identify linchpins is based on a brute-force method that tries all possibilities. Binkley *et al.* determined a set of conditions which, if met, will exclude certain program elements from being potential linchpins [8], and this way search for linchpins can be significantly optimized. We employ heuristic methods to identify linchpins, and we are not aware of any previous work that used a similar approach.

In a previous work [13], we investigated the concept of dependence clusters on procedure-level program dependences computed using the *Static Execute After (SEA)* approach. We computed $SEA$-based dependence clusters in the WebKit system and used them to verify the connection to the performance of change impact analysis in a practical situation, and to enhance our test case prioritization method based on code coverage analysis. In the present work, WebKit was one of our subject systems as well.

#### *2.1.1. Static Execute After*

The $SEA$ relation [11] is a conservative type of dependence on procedures that does not calculate nor use data dependences, but takes only the possible control-flow paths and call-structures inside procedures into account exclusively. This approach is more efficient at the expense of being a bit less accurate than slicing, and is defined as follows. For program elements (procedures, in our case) $f$ and $g$, we say that $(f, g) \in SEA$ if and only if it is possible that any part of $g$ is executed after any part of $f$ in any one of the executions of the program.

More formally, we define the $SEA$ relation involving two procedures $(f, g)$ as follows:

$$SEA = CALL \cup SEQ \cup RET \cup ID$$

where

$$\left.\begin{array}{lll}(f,g) & \in & CALL \\ (g,f) & \in & RET\end{array}\right\} \iff \begin{array}{l} f \text{ (indirectly) calls } g \\ \text{(or, } g \text{ (indirectly) returns into } f\text{)}\end{array}$$

$$(f,g) \in SEQ \iff \begin{array}{l}\exists h : f \text{ (indirectly) returns into } h\text{, and} \\ \text{there is a control-flow path to where } h \\ \text{(indirectly) calls } g\end{array}$$

$$(f,g) \in ID \iff f = g$$

The identity relation $ID$ is included because code dependence relations are usually defined as reflexive relations. Also, as can be seen from the definition above, $CALL$ and $RET$ are inverses of each other reflecting the fact that a called procedure returns to the caller eventually. The inverse of the $SEA$ relation as a whole is sometimes referred to as the *Static Execute Before (SEB)* relation. This way, $SEA$ may correspond to the notion of *static forward slice* while $SEB$ is analogous to the *static backward slice*.

For computing $SEA$ relations, we use a lightweight program representation called the *Interprocedural Component Control Flow Graph* (*ICCFG*) [11, 17]. It is composed of individual Component Control Flow Graphs (CCFGs) for each procedure of the program. Each CCFG represents a procedure's intraprocedural Control Flow Graph (CFG) [22] but only call site nodes and corresponding flow edges are retained. It contains one *entry* node and several *component* nodes which are connected by control flow edges. Component nodes are obtained by collapsing strongly connected subgraphs into single nodes. If the call sites in a component node are part of a loop the component will have a reflexive control flow edge. The ICCFG consists of the CCFGs of each procedure and in addition it includes call edges from each call site (a component) to the entry nodes of the called procedures.

We illustrate the ICCFG program representation on a small example program from Figure 1a. Figure 1b shows the corresponding ICCFG graph, in which nodes with the function names represent function entry nodes, while the darkly filled nodes correspond to the components. These are connected by control flow (solid) and call edges (dotted). Procedures `getIndex`, `printLast` and `printParam` are represented only by their entry nodes. We can easily see that this program representation is suitable for deriving $SEA$ (and $SEB$) relations. For example, we can follow that `printParam` may be executed after the procedures `main`, `getIndex` (and `printParam` itself), while `printLast` may be executed after all the procedures of the program including itself.

To compute the ICCFG we start from the interprocedural CFG of the program (with basic blocks computed), which can be obtained using traditional compiler algorithms and which is available in many source code analysis front ends. Then, the strongly connected components are obtinained and the ICCFG is constructed, which can be performed efficiently compared to the System Dependence Graphs used for program slicing [16].

For computing a dependency set for a particular procedure, a reachability

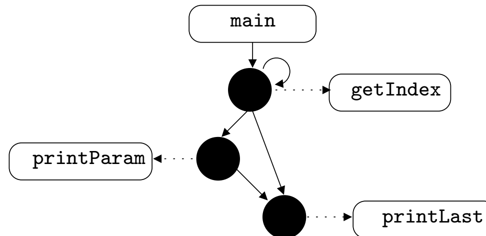
```
texts[] = {...}
procedure getIndex(out index){
  read(index);
}

procedure printParam(in index){
  write(texts[index]);
}

procedure printLast(){
  write(texts[texts.length]);
}

procedure main(){
  i = 0;
  while (i >= texts.length)
    getIndex(i);
  if (i != 0)
    printParam(i);
  printLast();
}
```

(a) A small example program




(b) The ICCFG of the example

Figure 1: A program and its ICCFG representation

algorithm can then be used on the ICCFG, similar to the SDG reachability algorithm to compute program slices. One can use either of two variants of the algorithm depending on the application. The basic algorithm can compute only one dependence set on demand [11], while an optimized version reuses already completed dependencies and produces the whole $SEA/SEB$ relation globally [17], which is more suitable for our empirical investigations.

In earlier work [11], we showed that the $SEA$ relations can be a good approximation of the static slices. In that experiment we used a suite of small to medium C programs, and calculated the precision of our relation compared to the results of the static slicing as the golden standard of static impact analysis. To this purpose we investigated the differences in the sizes of the respective dependence sets. The precision values we measured were very good, meaning that there is a comparably small amount of additional dependencies produced by the $SEA$ method due to its conservative nature. An example of a false dependency can be seen in our sample program from Figure 1a, where `printLast` is not dependent on procedures `getIndex` and `printParam` because there is no data flow between them, however $SEA$ identifies them as potential dependences. Since $SEA$ does not produce false negatives, we always get 100% recall.

In another empirical experiment we showed that the precision of the $SEA$ approximation is acceptable at procedure or higher level but not at statement level [23] (the dependencies among the statements computed by the program slices are assimilated on higher levels).

*2.1.2. SEA-based dependence clusters*

Similar to the definition of slice-based dependence clusters, we regard two procedures to be in the same *SEA-based cluster* if their dependence sets coincide. This kind of cluster definition is usual as the maximal mutual dependence-based cluster definition is prohibitively expensive to compute (which is essentially a *clique identification problem*). It is also a sensible definition because the $SEA$ relation is reflexive, so if two procedures have the same dependence set, then they depend on each other as well. This definition has the additional good property that it gives a partitioning of the procedures into clusters. We define our $SEA$-based dependence clusters more formally as follows.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of procedures in a program $X$ (for simplicity, we assume $n \geq 2$). The $SEA$ relation of program $X$ is a binary relation defined on its set of procedures, *i.e.* $SEA \subseteq P \times P$, according to the definition above. With $\bar{n} = \{0, 1, \ldots, n\}$, we give the following auxiliary definitions:

For any procedure $p$, two sets of procedures are associated with it: the set of procedures that are dependent on $p$ (*i.e.* the procedures that are successors of $p$ according to $SEA$), and the set of procedures on which $p$ depends (*i.e.* the procedures that are predecessors of $p$ according to $SEA$). The former is referred to as the *SEA-set of p*, while the latter is its *SEB-set* (as mentioned, these notions are analogous to forward and backward slices, respectively). Because the SEB relation is defined as the inverse of $SEA$, in the following we will use the notations $SEA$ and $SEA^{-1}$ for simplicity. Additionally, we use a notation that emphasizes the direction of the dependences as follows:

$$\overrightarrow{D}(p) = \{q \in P \mid SEA(p, q)\}$$

using forward dependences, and

$$\overleftarrow{D}(p) = SEA^{-1}(p) = \{q \in P \mid SEA(q, p)\}$$

using backward dependences.

As mentioned, in the context of slice-based clusters often the approximation is used to check the sizes of the dependence sets instead of actual set coincidence [2, 6]. Although dependence set size difference is a good approximation to set difference in the case of $SEA$ as well (we verified this and found negligible difference with the smallest sets only), in order to ease formal definition of cluster metrics we define two types of dependence clusters, one with set coincidence ($\mathcal{I}$) and one with size comparison ($\mathcal{S}$), both having backward and forward versions. For the latter we will need an additional definition of the dependence set size, the weight functions $\overrightarrow{w}$ and $\overleftarrow{w}$:

$$\overrightarrow{w} : P \to \bar{n}, \quad \overrightarrow{w}(p) = \left|\overrightarrow{D}(p)\right| \qquad \text{and} \qquad \overleftarrow{w} : P \to \bar{n}, \quad \overleftarrow{w}(p) = \left|\overleftarrow{D}(p)\right|$$

The formation of dependence clusters of a program based on $SEA$ dependences is in fact a partitioning of the procedure set $P$ (not being transitive, the $SEA$ relation itself does not exhibit partitions). For both forward and backward dependence sets we define two kinds of clusters, one by assigning two

procedures to the same cluster (partition) if they have the same dependence set, and another by considering only the sizes (weights) of the dependence sets of the procedures. For forward dependence sets, the definitions are the following:

$$\vec{\mathcal{I}} = \left\{ \left\{ q \in P \mid \vec{D}(q) = \vec{D}(p) \right\} \,\middle|\, p \in P \right\}$$

$$\vec{\mathcal{S}} = \left\{ \left\{ q \in P \mid \vec{w}(q) = \vec{w}(p) \right\} \,\middle|\, p \in P \right\}$$

For any $c \in \vec{\mathcal{S}}$ the weights of its members are equal, so we can assign the same weight to cluster $c$ itself. Clearly $\vec{D}(q) = \vec{D}(p)$ implies $\vec{w}(q) = \vec{w}(p)$, so $\vec{\mathcal{I}}$ is a refinement of $\vec{\mathcal{S}}$. This also means that the weight function can be extended naturally to $\vec{\mathcal{I}}$ as well. Note, however, that the weight and the size of a cluster are different notions, the former may also be referred to as dependence set size.

The corresponding backward definitions can be derived from the ones above easily, and the same considerations apply hence we did not include the definitions here.

Note, that there is no obvious relationship between $SEA$-based and slice-based dependence clusters. From our preliminary investigations we found that in many cases similar cluster structures will be formed, but we plan to investigate this more systematically in the future, and see if our findings can be applied to other notions of dependence clusters.

### *2.2. Problem statement*

Despite the advances in dependence cluster research so far, we do not fully understand the nature of these formations yet. It is not even clear whether clusters are good or bad in every situation; when do they represent dependence pollution. As noted above, clusters are usually treated as detrimental to software processes, however they carry useful information about program functionality and structure, and they may not always be avoided. In the case of linchpins, researchers started investigating practical methods for their identification and potential removal only recently. Consequently, more research is needed in the area before considering applications in practice. With this work we contribute to the topic by using our $SEA$-based dependences and providing further useful instruments for the investigation of dependence clusters.

Monotone Size Graphs (MSG) and the related "area under the MSG" metric [2] are often used to characterize dependence clusters in programs. An MSG of a program (see Figures 2–6 for examples) is a graphical representation of all dependence sets belonging to the procedures of the program by drawing the sizes of the sets in monotonically increasing order along the $x$ axis from left to right. Then the area metric mentioned above is the total sum of all dependence set sizes. In the case of $SEA$-based dependences the total number of dependence sets equals the number of procedures in a program, and this number is also the maximal dependence set size. In these figures we can see MSGs of such dependences in which – despite its rectangular shape – the same number of procedures

is represented on both axes. The most straightforward way of interpreting this graph is to observe dependence clusters as plateaus, whose width corresponds to the cluster size and the height is the dependence set size. Note, that it may happen that the same dependence set size incidentally corresponds to different sets which will not be noticeable in this graphical representation. We verified the amount of such incidental correspondence and found it negligble as is the case with slice-based clusters.

It has not yet been investigated thoroughly whether the MSG and its associated area metric are good enough descriptors of the level of clusterization. We further elaborate on this concept with $SEA$-based clusters and verify the ways of characterizing dependence clusters using this and other kinds of metrics. A related investigation was performed by Islam *et al.*, who defined alternative descriptions of the clusterization in form of various graphical representations [24]. These approaches, however, resort to visual investigation only.

As noted above, it is believed that a single linchpin can be associated to a program with dependence clusters in many cases (see, for example, program 'compress' in Section 5, Figure 11). In this work we investigate the effect of joint removal of linchpin candidates in cases where the removal of a single element does not produce the desired result. An additional problem is linchpin identification itself. The naïve linchpin identification algorithm – a brute-force method trying all possible solutions one by one – is not scalable. Hence, previous research that employed fine grained analysis could deal with programs of up to 20 kLOC [4] or 66 kLOC using the advanced method [8]. In a similar fashion, our $SEA$-based analysis makes it possible to investigate programs with sizes of a magnitude larger thanks to the higher level granularity and a simpler, albeit less precise, analysis method. However, this method is still not usable for bigger programs as is the case with our two large systems.

We articulate the following **Research Questions**:

**RQ1** How common are large $SEA$-based dependence clusters in a variety of programs of different sizes and how can we categorize the programs more objectively in terms of their degree of clusterization?

**RQ2** How typical it is that cluster structures are held together by at most a few clearly identifiable procedures (linchpins), *i.e.* ones whose removal reduces program clusterization significantly?

**RQ3** Currently the exact linchpins can be determined by brute-force only, which is infeasible for bigger programs; Are there any low-cost heuristic methods that are able to approximate linchpins with acceptable accuracy?

**RQ4** How is the prediction capability of $SEA$-based impact analysis affected if possible linchpin nodes are disregarded during calculations?

## 3. Experiments setup

We collected a set of programs that served as the subjects following these principles: the set had to be comparable to other researchers' and our own

previous results, our existing tools had to be able to handle them with ease, the programs had to be from different domains, and their sizes had to vary in a wide range. Based on these requirements, we fixed the language of the programs to C/C++, and in the first instance started with the collection of programs Harman *et al.* used in their experiments [6]. We could reuse 60% of these programs but also extended this set to finally arrive at 29 programs written in C (we will refer to this set as the *moderate size* programs). The basic properties of these programs can be seen in the first three columns of Table 1. We provide names, lines of code (LOC) and the number of procedures (NP). The purpose of the other columns will be explained later.

The second part of our data set consisted of two large industrial software systems from the open source domain. The first one was the WebKit system, which we have already used in some of our previous investigations [13, 25]. WebKit is a popular open source web browser engine integrated into several leading browsers by Apple, KDE, Google, Nokia, and others [10]. It consists of about 2.2 million lines of code, written mostly in C++, JavaScript and Python. In this research we concentrated on C++ components only, which attributes to about 86% (1.9 million lines) of the code. In our measurements we used the Qt port of WebKit called QtWebKit on x86_64 Linux platform. We performed the analysis on revision 91555, which contained 91,193 C++ functions and methods as the basic entities for our analysis.

The other large system we used was the GNU Compiler Collection (GCC), the well-known open source compiler system [9]. It includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages. The GCC system is large and complex, and its different components are written in various languages. It consists of approximately 200,000 source files, of which 28,768 files are in C, which was the target of our analysis. In terms of lines of code, this attributes to about 13% of the code, 3.8 million lines in total (note that in C, the size of individual functions is usually larger than that of an average C++ method, hence this difference in lines of code compared to WebKit). We chose revision 188449 (configured for C and C++ languages only) for our experiments, in which there were 36,023 C functions as the basic entities for our analysis.

In our experiments we used our custom build tools as well as some existing components. To extract base program representations, as parser front ends we used Grammatech CodeSurfer [26] in the case of moderate size programs and Columbus [27] for the big programs. For the $SEA$ dependence computation, our existing implementation of the $SEA$ algorithm using ICCFG graphs [17] was applied. The benefit of using Columbus with ICCFG graphs for the bigger programs is that it is more scalable due to higher granularity of analysis.

We modified the $SEA$ computation tool by adding the capability to ignore one or more procedures (as if they were removed), which was required for linchpin determination.

Since we needed to process and store a large number of dependence sets, we implemented additional tools (for MSG computation, cluster metrics computation, etc.) that employ efficient specialized data structures and algorithms.

Table 1: Moderate size subject programs with clusterization information, sorted by Visual class and NP

| **Program name** | **LOC** | **NP: # of procedures** | **Visual class** | **Clusterization metrics** | | | |
|---:|---:|---:|---|---|---|---|---|
| | | | | AREA | ENTR | REGU | REGX |
| lambda | 1766 | 104 | ▼ low | | | | |
| epwic | 9597 | 153 | ▼ low | | | | |
| tile-forth | 4510 | 287 | ▼ low | | | | |
| a2ps | 64590 | 1040 | ▼ low | | | | |
| gnugo | 197067 | 2990 | ▼ low | | | | |
| time | 2321 | 12 | ■ med | | | | |
| nascar | 1674 | 23 | ■ med | | | | |
| wdiff | 3936 | 29 | ■ med | | | | |
| acct | 7170 | 54 | ■ med | | | | |
| termutils | 4684 | 59 | ■ med | | | | |
| flex | 22200 | 153 | ■ med | | | | |
| byacc | 8728 | 178 | ■ med | | | | |
| diffutils | 17491 | 220 | ■ med | | | | |
| li | 7597 | 359 | ■ med | | | | |
| espresso | 22050 | 366 | ■ med | | | | |
| findutils | 51267 | 609 | ■ med | | | | |
| compress | 1937 | 24 | ▲ high | | | | |
| sudoku | 1983 | 38 | ▲ high | | | | |
| barcode | 5164 | 70 | ▲ high | | | | |
| indent | 36839 | 116 | ▲ high | | | | |
| ed | 3052 | 120 | ▲ high | | | | |
| bc | 14370 | 215 | ▲ high | | | | |
| copia | 1168 | 242 | ▲ high | | | | |
| userv | 8009 | 255 | ▲ high | | | | |
| ftpd | 31551 | 264 | ▲ high | | | | |
| gnuchess | 18120 | 270 | ▲ high | | | | |
| go | 29246 | 372 | ▲ high | | | | |
| ctags | 18663 | 535 | ▲ high | | | | |
| gnubg | 148944 | 1592 | ▲ high | | | | |

Specifically, we used the SoDA library [28] to store and process the dependence sets.

We computed the $SEA$-based dependence sets for each procedure in the programs and for both directions (*i.e.* $SEA$-sets and $SEB$-sets) and determined the two types of associated dependence clusters, as well as the corresponding sets of clusterization metrics. The calculation of the final results and the whole measurement procedure was performed using shell scripts and spreadsheets.

We will describe our set of experiments and additional details regarding the measurements and tools in the corresponding sections.

## 4. Existence of Dependence Clusters

### *4.1. Identification of clusters*

To obtain the dependence clusters and investigate the level of clusterization we computed the $SEA$-based dependence sets for both forward and backward directions for all procedures in our subject programs. The structure of our dependence sets is fortunately simple: for each procedure in a program we compute the corresponding set of procedures it is in $SEA$ relation with. Hence, the total number of dependence sets equals the number of procedures in a program, and this number is also the maximal dependence set size (note that for other types of dependence sets, for example program slice based dependences [2], this is not necessarily true). Altogether we computed 23,970 dependence sets for the moderate size programs, 182,386 for WebKit and 72,046 for GCC.

In the following, we will investigate the *clusterization* of programs, which is the extent they exhibit dependence clustering. To express clusterization we followed two approaches:

**Visual classification** is carried out by (subjective) visual inspection of the MSGs, and assigns one of three levels to each program: *low*, *medium* and *high*.

**Clusterization metrics** are rigorously defined measures that are designed to express clusterization in easily quantifiable numerical form (values from $[0, 1]$).

For the moderate size programs, the fourth column (Visual class) of Table 1 shows the results of the visual classification we performed by inspecting the MSGs of the programs. Overall, we found 5 programs to be low, 11 medium, and 13 highly clusterized. Figures 2–6 show the actual MSGs organized into the three levels of clusterization, which were used to perform the classification. For each program, we provide MSGs computed for both forward and backward dependence-based clusters (denoted by **-succ* and **-pred*, respectively). The differences between the two directions will be discussed shortly.

A cluster reveals itself as a wide plateau consisting of a number of equal-sized dependence sets. Typically, in a low class we cannot identify any plateaus, while for the high class there are one or two big ones, the rest being medium. In

this graphs, we chose to display only the sizes of the dependence sets and this way the same-size but different sets cannot be distinguished. As mentioned, same dependence set size may incidentally correspond to different sets, however this is very unlikely except for the smallest sets, so this will not distort our visualization.

Let us consider as examples three typical programs from each category. Visual classification reveals the following:

- Program `epwic` (Figure 2) does not show any plateaus, the landscape ascends in small increases.
- `findutils` (Figure 4) contains some moderately wide plateaus. They are not significant individually, but altogether cover much of the width of the landscape.
- For the program `gnubg` (Figure 6) we can see a single plateau occupying nearly the whole width of the landscape.

### *4.2. Measuring clusterization*

Beyond visual interpretation, we will need an exact numerical expression (metric) of the level of clusterization and its relative change for two reasons. First, this way an automatic classification of programs could be made with the help of appropriate thresholds. Second, the metrics can be applied for the analysis of linchpins and measuring the effect of their removal (discussed in subsequent sections).

The obvious choice of metric to be used in these kinds of experiments is based on Binkley and Harman's work [2], who measured the area under MSG and used the change of this metric to analyze linchpins (we also rely on this metric and denote it by AREA in the following). The apparent weakness of AREA is that it increases if all dependence sets are increased by the same amount, although intuitively clusterization should not be different in such cases. Programs with no dependence clusters can have both small and large dependence sets, and vice versa.

We experimented with alternative metrics to express clusterization in programs that are independent of the actual dependence set sizes and could reflect this property (a preliminary version of the metrics has been used in previous work [29]). We define the measures so that they are comparable to each other and yield a value in the interval $[0, 1]$. For all metrics, 0 is set to mean that the level of clusterization is close to none, while 1 means that clusterization is maximal.

We define two variants for each metric corresponding to the two directions of the base $SEA$ relation and the associated dependence clusters following the definitions above. However, in the following we will provide formal definitions only for the forward direction because all metrics for the opposite direction can be derived in a straightforward way by changing the direction of the base dependence cluster sets.

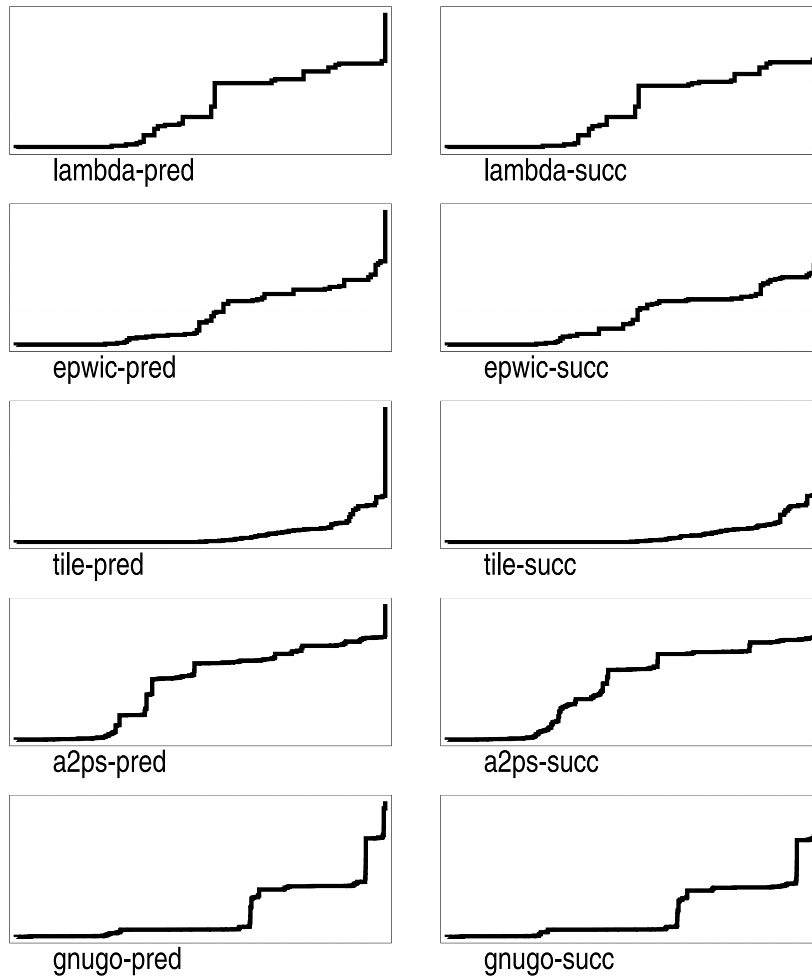



Figure 2: Low clusterization

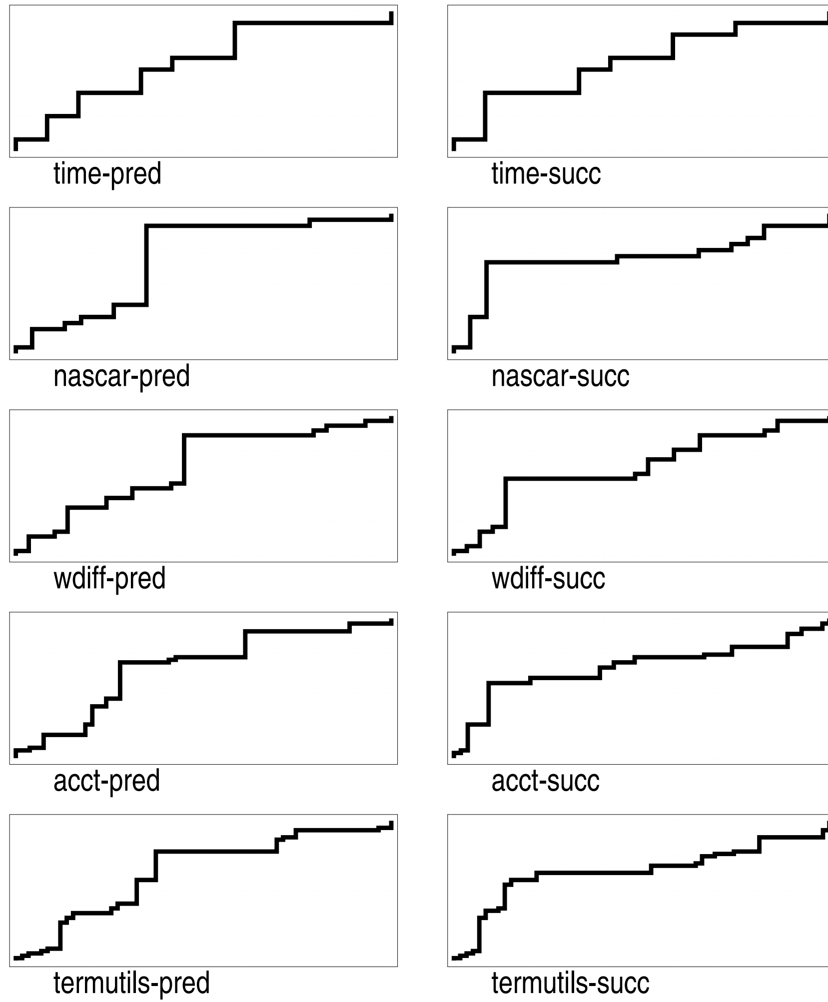



Figure 3: Medium clusterization, part 1

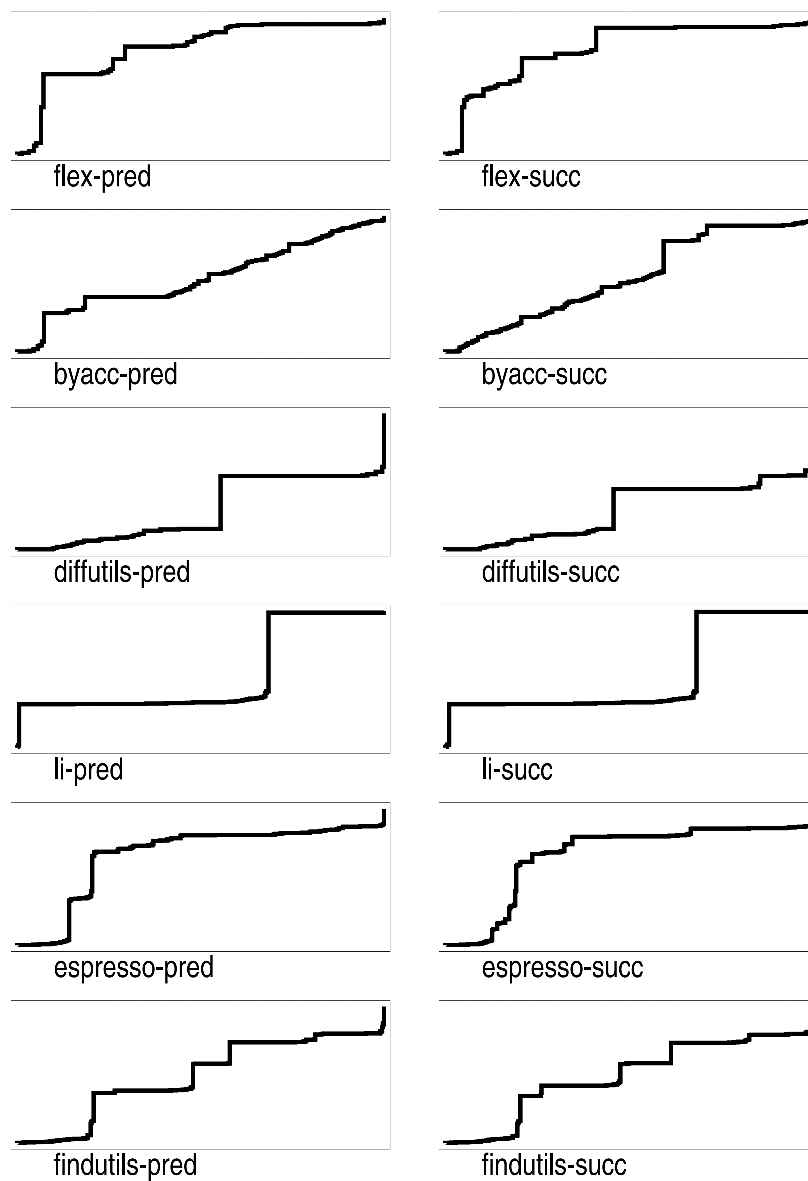



Figure 4: Medium clusterization, part 2

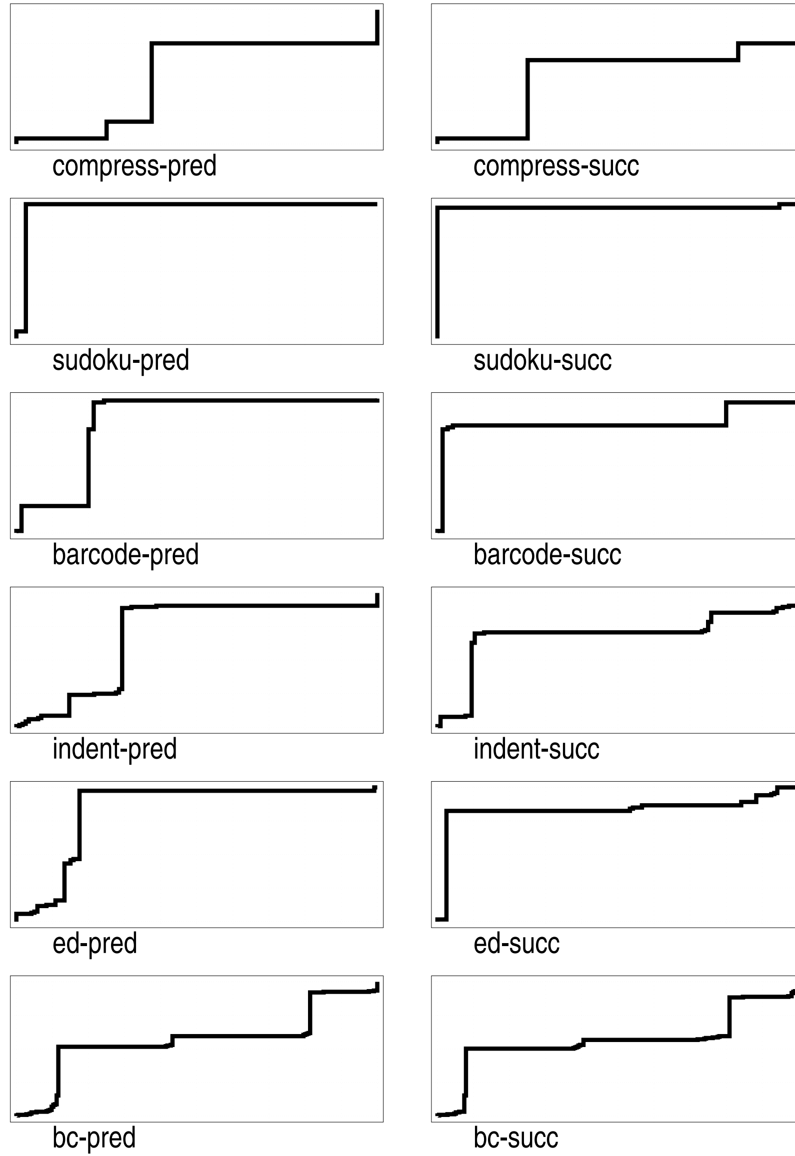



Figure 5: High clusterization, part 1

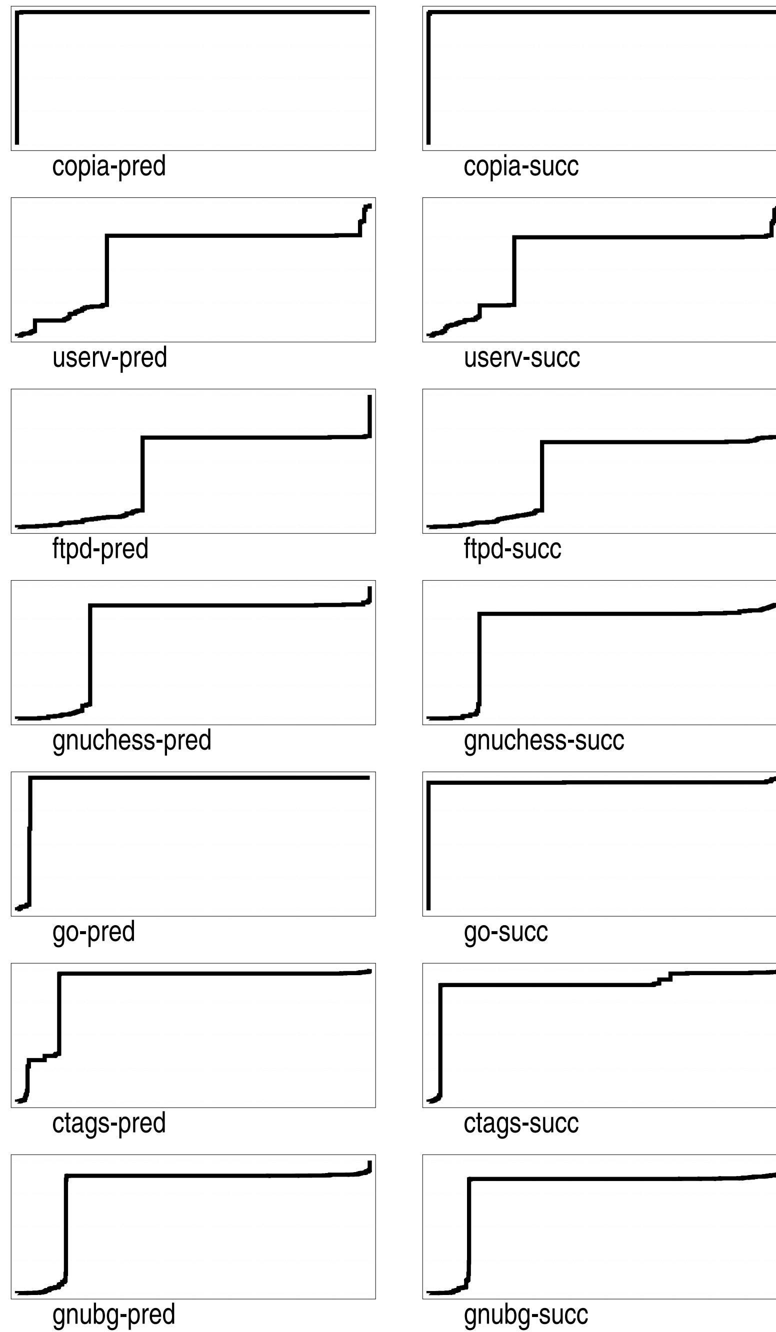



Figure 6: High clusterization, part 2

*4.2.1. Definition of metrics*

We define the different clusterization metrics as follows (the metrics always have to be interpreted in the context of a given program). Consistently with earlier descriptions, $\overrightarrow{\text{AREA}}$ has the following definition:

$$\overrightarrow{\text{AREA}} = \frac{1}{n^2} \sum_{c \in \overrightarrow{\mathcal{I}}} |c| \cdot \vec{w}(c)$$

Our next metric is based on an analogy of traditional *entropy* and measures the "(dis)order" in the system of dependence sets in terms of their sizes. We consider a program more clusterized in this respect if there is a greater number of equal-sized dependence sets, *i.e.* when the entropy is lower (note, that this inverse relationship is required to obtain comparable metric intervals with the other metrics). Our entropy-based clusterization measure is formally defined as:

$$\overrightarrow{\text{ENTR}} = 1 - \frac{\sum_{c \in \overrightarrow{\mathcal{S}}} f(c) \cdot \log_2 f(c)}{\log_2 1/n}, \text{ where } f(c) = \frac{|c|}{n}$$

The above can be simplified to the following formula:

$$\overrightarrow{\text{ENTR}} = \frac{\sum_{c \in \overrightarrow{\mathcal{S}}} |c| \cdot \log_2 |c|}{n \log_2 n}$$

Finally, our two metrics referred to as *regularity* metrics are based on the number of partitions. The idea is that the fewer partitions there are, the larger their size must be, so there have to be more large clusters among them. Inversely, more partitions have to take more "regular" different sizes hence they will represent low clusterization. This metric has two variants, the first is based on $\overrightarrow{\mathcal{S}}$, the other (extended, $\overrightarrow{\text{REGX}}$) is based on $\overrightarrow{\mathcal{I}}$.

$$\overrightarrow{\text{REGU}} = \frac{n - \left|\overrightarrow{\mathcal{S}}\right|}{n-1} \qquad \overrightarrow{\text{REGX}} = \frac{n - \left|\overrightarrow{\mathcal{I}}\right|}{n-1}$$

As noted earlier, all metrics are normalized (*i.e.* their value is a real number from the interval $[0, 1]$), which is useful to be able to compare the metrics of programs with different size to each other.

*4.2.2. On the difference between forward and backward*

In Figures 2–6 we can observe that the MSGs corresponding to the two different directions for a given program generally do not differ very much. Also, measurements showed that the metric values based on either the backward or the forward cluster definitions are very close to each other. Table 2 shows the absolute differences in the different metrics, which we may describe as practically negligible. This is an interesting finding because theoretically an arbitrary difference could be possible, and one might easily produce a counterexample.

Table 2: Differences between backward and forward metric values for the moderate size programs

| | AREA | ENTR | REGU | REGX |
|---|---|---|---|---|
| minimum | 0.000 | 0.000 | 0.004 | 0.004 |
| maximum | 0.000 | 0.264 | 0.300 | 0.200 |
| average | 0.000 | 0.056 | 0.051 | 0.039 |

Note, that metrics for AREA will always be equal, as:

$$\overrightarrow{\text{AREA}} = \frac{1}{n^2}\sum_{c\in\overrightarrow{\mathcal{I}}}|c|\cdot\vec{w}(c) = \frac{1}{n^2}\sum_{p\in P}\vec{w}(p) = \frac{1}{n^2}\sum_{p\in P}\overleftarrow{w}(p) = \frac{1}{n^2}\sum_{c\in\overleftarrow{\mathcal{I}}}|c|\cdot\overleftarrow{w}(c) = \overleftarrow{\text{AREA}}.$$

However, in the case of the other metrics in turned out that the difference may be arbitrarily large between the forward and backward variants. To demonstrate this, we are going to construct examples with essentially the largest possible differences between the forward and backward cases. As an aid to these examples a graph-based representation of the $SEA$ relation will be used, so that we will have an $(a, b)$ edge in the graph for $a, b \in P$ if and only if $b \in SEA(a)$.

Let us consider first the REGU metric. We build a directed graph $\vec{G}_1$ with vertex set $P$ as follows.

- For every $p \in P$ we include a loop edge $\vec{pp} \in E(\vec{G}_1)$.
- For every $i \in \{1, 2, \ldots, n-1\}$ we choose a random subset $R_i \subset P \setminus \{p_i\}$ independently of other choices, so that $|R_i| = i$. Then we have all edges $\vec{p_i q}$ where $q \in R_i$.

Observe that the outdegree of $p_i$ is exactly $i + 1$ for every $1 \le i \le n - 1$, and the outdegree of $p_n$ is 1. Hence, the sizes of the outneighborhoods are different, and the REGU value for this graph is zero. The sum of the outdegrees is $n(n + 1)/2$, and each outgoing edge is also an incoming edge for the other endpoint of the edge. However, the indegrees are randomly and evenly distributed among the vertices of the graph. On average, every indegree is $(n + 1)/2$, and standard probabilistic reasoning (for example, using a martingale inequality by Azuma [30]) shows that with positive probability each vertex of $\vec{G}_1$ has indegree in the range

$$\left[(n+1)/2 - 6\sqrt{n\log n}\,,\ (n+1)/2 + 6\sqrt{n\log n}\right]$$

This implies that the number of different outdegrees is at most $1 + 12\sqrt{n \log n}$. Hence, the number of backward clusters is at most $1+12\sqrt{n \log n}$, and the REGU value is at least $\dfrac{n-1-12\sqrt{n\log n}}{n-1} \approx 1$ if $n$ is not small.

Next, we construct a directed graph $\vec{G}_2$ with vertex set $P$ that will prove to be a good example for REGX and ENTR. In the following, let $k = \lceil \log_2 n \rceil$.

- For every $p \in P$ we include a loop edge $\vec{pp} \in E(\vec{G_2})$.
- Let $A = \{p_1, p_2, \ldots, p_{n-k}\}$, and let $B = P \setminus A = \{p_{n-k+1}, \ldots, p_n\}$. For easier notation, we rename the vertices of $B$, that is, we let $B = \{b_1, b_2, \ldots, b_k\}$.
- For every $p_i \in A$ we have the $\vec{p_i b_j}$ edge in $E(\vec{G_2})$, if in the binary representation of $i$ the $j$th bit is 1.

Observe that with this construction we made sure that the outneighborhoods of the vertices of $A$ are all different. That is, we have at least $n - k + 1$ forward dependence clusters, and at least $n - k$ of these have size 1. On the other hand, the inneighborhood of every vertex of $A$ is $A$ itself, that is the number of backward dependence clusters is at most $1 + k$, furthermore, one of these clusters has size at least $n - k$. These facts imply, using the definitions of REGX and ENTR, that for both metrics, the forward clusterization is very close to zero, while the backward clusterization is very close to 1, if $n$ is not small.

As a conclusion, although theoretically arbitrary difference could be possible with the forward and backward variants of the metrics, in practice it is negligible. This is probably due to the fact that real programs do not produce arbitrary dependence graph structures but specialized ones, however it would be an interesting future line of research to investigate this phenomenon more deeply. In the following we will work with only forward dependence ($SEA$-based) clusters.

*4.2.3. Comparison to the visual classification*

We computed all four metrics for all of our moderate size subject programs and compared the rankings of the procedures based on these individual metrics to the visual classification of the programs. These values are shown in the last four columns of Table 1 (to ease interpretation, the gray areas inside the small rectangles are set to be proportional to the metric values). Note that the ordering of the programs in this table was done based on the visual ranking first, then on the number of procedures inside each rank group.

Visual clusterization created three groups with 5 (low), 11 (medium), and 13 (high) elements, respectively. We would expect an ideal clusterization metric to yield values in such a way that the 5 smallest would be assigned to the "low" level, the middle 11 would be assigned to "medium", and the largest 13 to the "high" level group. Based on these criteria, the clusterization metrics can be characterized by counting how many programs they fail to assign to the group given by visual ranking. The counts are as follows: AREA $\rightarrow$ 10, ENTR $\rightarrow$ 7, REGU $\rightarrow$ 15, REGX $\rightarrow$ 8. The differences in these counts can also be observed by visual inspection of the metric values. It can clearly be seen that AREA and REGU are significantly worse than the other two metrics, while the difference is not so great regarding ENTR and REGX. ENTR is more precise on low and medium clusterization levels, while REGX performs better on highly clustered programs.

Based on the above observations we will use ENTR and REGX to measure the degree of clusterization in the rest of the paper, and will mostly rely on

ENTR where low or medium clusterization is concerned and use the other for high clusterization.

*4.3. Dependence clusters in the big programs*

So far, we have been dealing only with the moderate size programs, but our dataset contains two big programs as well, which need more thorough investigation. The MSGs for these two programs, GCC and WebKit, can be seen in Figure 7.

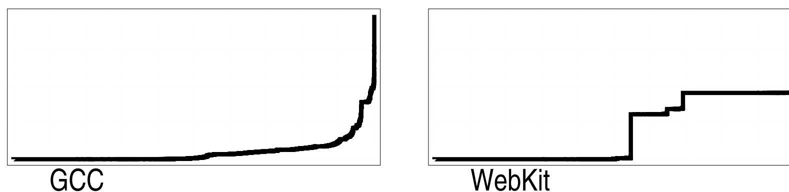



Figure 7: MSGs for `GCC` and `WebKit`

The differences between the two programs are clear. GCC belongs to the low level clusterization category, while WebKit exhibits some clusterization (it would belong to the medium category in the visual ranking). The ENTR values are 0.4347 for GCC and 0.6980 for WebKit, while REGX is 0.3134 and 0.3552, respectively, which supports our initial (visual) classification for these two systems. While ENTR shows a notable difference, in the case of REGX it is not so significant, which may also reflect our finding from above that ENTR was better for low or medium clusterization.

It would be interesting if we could find any properties of these systems that justify their classification in terms of dependence clusters. In other words, what makes GCC not having significant dependence clusters as opposed to WebKit? In previous work [13], we analyzed the structure of source code and the dependences in WebKit in a slightly different context. After consulting with some key WebKit developers and showing them the members of the clusters, we came to the conclusion that clusterization is related to architectural concepts in the system.

We speculate that the most notable difference between the two systems in this respect is that while WebKit is essentially a library consisting of highly coupled elements for the distinct functional areas, GCC is a complex application but with much clear behavioural paths that are independent of each other. In WebKit, most complex functionalities are implemented in a set of highly interacting procedures (for example, webpage rendering is performed by several hundred procedures calling each other recursively). On the other hand, GCC implements functionalities like compiler optimization passes that are more isolated from each other. In addition, the two systems are written in different

programming paradigms (C vs. C++) which may influence their internal structure. More detailed analysis of the causes for this difference remains for future work.

In the remaining parts of this paper we will be concerned with the identification of linchpins, however for programs of this size only the heuristic approaches are feasible. Hence we will subsequently use WebKit only to verify the effect of our heuristic methods, while GCC will play no role from now on.

## 5. Linchpin determination

Informally, linchpins are those elements that are responsible for keeping large clusters together, *i.e.* whose removal causes the clusters to break up into smaller ones or to disappear. Earlier we introduced clusterization metrics that allow us to define linchpin elements more precisely: in a program, the linchpin is a procedure whose removal results in the largest decrease in clusterization according to a given metric.

First, we identified the linchpins for our moderate size programs using the brute-force method that enumerates all procedures. As the biggest challenge in this topic is how to locate linchpins using more efficient methods, in the next experiment we investigated approximate heuristic methods for this task and compared their results to the exact results of the brute-force method. Since we could not apply the brute-force method to our large program WebKit, we verified the successfulness of the heuristic method on it in the final step.

### *5.1. Linchpin identification by brute-force*

The simplest way to identify a possible linchpin in a program is to remove procedures one by one and see which one brings the biggest gain according to some metric. In the following, *gain* will mean the amount the respective metric is reduced in percentage: $\frac{m-m'}{m}$ [%], $m$ being the original metric value and $m'$ the value after linchpin removal.[1]

Specifically, we computed all $SEA$ dependence sets for a program by removing each procedure, *i.e.* by ignoring the candidate procedure and all of its dependences during dependence set calculations. We compared the ENTR and REGX metrics of the reduced versions of the program to the corresponding metrics of the original program. This calculation was then repeated for all procedures in the program with these two metrics. To get these results we had to compute over 15 million $SEA$ sets altogether, but this was possible to complete in hours on an average server machine.

For simplicity, we will present our results for REGX in the cases when the results were similar for both metrics, and will note explicitly in other situations. For the purposes of the remaining discussion, the procedure that caused the biggest reduction in the REGX metric was considered to be the linchpin.

[1] Note, that we do not actually *refactor* linchpins and get equivalent programs but we merely remove the procedures in order to identify them.

Table 3: Linchpin removal gains as measured by REGX

| | Minimum gain | Maximum gain | Typical gain |
|---|---|---|---|
| Low clusterization | 5% | 20% | — |
| Medium clusterization | 18% | 55% | 20% |
| High clusterization | 13% | 99% | 37% |

Table 3 summarizes results of linchpin determination for different clusterization classes of moderate sized programs. It shows how much reduction in the REGX clusterization value could be attained in the worst case (*minimum gain*) for a given clusterization class, and similarly how much was the *maximum gain*. It also indicates how much reduction could be attained if the few outlier programs with least gain—4 in the medium and 3 in the high class—are ignored (*typical gain*).

One would expect that programs with low clusterization do not contain linchpins, *i.e.* there is no procedure whose removal significantly reduces the already low clusterization. This was not entirely supported by our findings, as there were cases when as much as 20% gain could be achieved. This is not negligible, but as noted earlier, REGX performs better at high clusterization levels, so results for the low clusterization level are not entirely relevant. However, the achieveable gain is definitely larger for the medium and high classes, and gains for highly clusterized programs vary widely.

An interesting observation we made was that in many programs the procedure with the highest gain was not the only one causing a big change in clusterization; there were several others in the row that performed comparably. More precisely, typically the first 1–3 procedures caused a high drop in clusterization, while the others followed with much less gain. Figure 8 shows the overall results for the moderate size programs with high clusterization, where the gain of individual removal of the first 30 procedures is shown. As can be seen, in many cases the second and the third procedure also behaves as a linchpin, not only the first one.

In Table 4, we listed the actual linchpins identified (the first 1–3 procedures are shown that showed significant difference to the remaining ones). The second column shows the REGX gain after removing the linchpin with the highest gain, which was quite significant (at least 37%) in almost all of the cases, 43.7% on average for this class of programs. The last columns of the table show the names of the respective procedures identified (which, except for `compress`, `gnubg` and `go`, were the same for ENTR as well). It is interesting to observe that, as names themselves suggest and a manual analysis of the programs confirms, most of the identified procedures indeed have central role in the programs. It is an open question, however, how many of these procedures could be deemed responsible for avoidable dependence clusters, in other words, dependence pol-

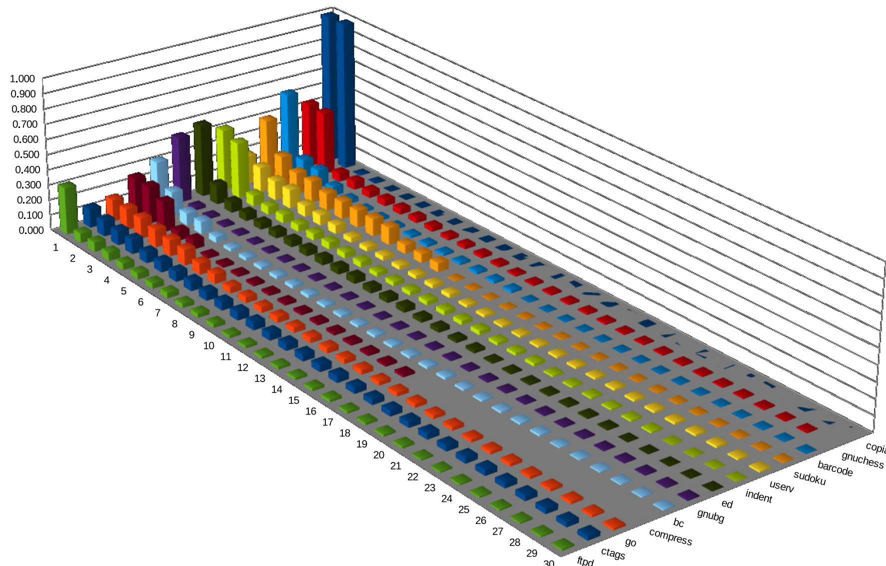



Figure 8: Highest reductions gained by individual linchpin removal for moderate size programs with high clusterization

lution [2]. Expectedly, procedures acting as the main procedures could not be easily refactored.

*5.2. Heuristic determination of linchpins*

We estimated that the brute-force method to determine the potential linchpin for the WebKit system would take about 70 years to complete using our strongest servers. So, obviously, we must find alternative methods to find (or at least approximate) the linchpins to enable practical application of dependence cluster related research.

The existence of dependence clusters and any related linchpins are determined by the structure of the dependences under investigation ($SEA$ and the underlying ICCFG program representation in our case). Therefore, it is to be expected that by investigating the topology of the underlying dependence graph one could gain insight into what makes a program point a potential linchpin.

The problem does not have an obvious solution, so we wanted to investigate whether local properties of the dependence graph nodes (procedures) could be leveraged to approximate lichpins. We used the following heuristic metrics as potential indicators: NOI (Number of Outgoing Invocations from the procedure), NII (Number of Incoming Invocations to the procedure), sum of the former two (SOI=NOI+NII), and their product (POI=NOI·NII). We tried the sum and the product because we expected that in linchpin formation both incoming and outgoing dependences could be important.

Table 4: Linchpins for moderate size programs with high clusterization

| Program | Max gain | Procedure$_1$ | Procedure$_2$ | Procedure$_3$ |
|---|---|---|---|---|
| barcode | 57% | Barcode_Encode | | |
| bc | 37% | dc_func | execute | |
| compress | 37% | compress | main | spec_select_action |
| copia | 99% | scegli | seleziona | |
| ctags | 13% | createTagsForFile | | |
| ed | 53% | exec_command | | |
| ftpd | 43% | parser | | |
| gnubg | 52% | HandleCommand | | |
| gnuchess | 53% | main | parse_input | |
| go | 14% | evalshapes | callfunc | get_reason_for_moves |
| indent | 47% | indent_main_loop | handle_the_token | |
| sudoku | 41% | rsolve | | |
| userv | 22% | parser | servicerequest | main |

To compare the actual linchpins identified by the brute-force method to the performance of the heuristic metrics, we related two values for each procedure in the programs: a clusterization metric (ENTR or REGX) after removing the procedure and one of the heuristic metrics (NOI, NII, SOI, POI) associated with the procedure. We then used Pearson and Kendall correlation checks between the corresponding vectors of these values.

We do not provide detailed data for these measurements because they all pointed out the same best heuristic estimation. Instead, in Table 5 we show Pearson correlation results for all programs. We marked the strongest correlation values for each program underlined; the last two rows show the average correlation values and the counts of strongest cases for each metric. It can clearly be seen that the NOI metric (Number of Outgoing Invocations) is the best estimator for both ENTR and REGX. The best values are negative in the NOI columns, which means that for the procedures of a program there is a high correlation between a high NOI value and a low clusterization value resulting from the removal of that procedure. In other words, the higher NOI value a procedure has, the more likely it is that its removal would decrease the clusterization considerably, *i.e.* the more likely it is that the procedure is a linchpin.

In the case of ENTR and REGX metrics, in 59% and 79% of the cases NOI showed the strongest correlation; the average correlation was $-0.36$ and $-0.63$ (with standard deviations 0.4 and 0.18), respectively. The second best was POI showing strongest correlation in 38% and 14% of the programs with average correlation values $-0.26$ and $-0.42$. NII performed poorly, which was surprising because we expected NOI and NII will perform similarly. The promising results for NOI are strengthened by the fact that the highest NOI value predicts a linchpin correctly in most of the cases: in the highly clustered group in 12 out of 13 programs, in the medium group in 7 out of 11 programs the procedure

Table 5: Pearson correlation between heuristic metrics and the ENTR and REGX metric. Underlined numbers indicate strongest correlation in the corresponding block.

| | ENTR | | | | REGX | | | |
|---|---|---|---|---|---|---|---|---|
| **Program** | **NOI** | **NII** | **SOI** | **POI** | **NOI** | **NII** | **SOI** | **POI** |
| lambda | 0.30 | 0.53 | 0.50 | 0.58 | -0.61 | -0.49 | -0.64 | -0.57 |
| epwic | 0.28 | 0.12 | 0.32 | 0.32 | -0.50 | -0.02 | -0.48 | -0.32 |
| tile | 0.48 | 0.46 | 0.62 | 0.63 | -0.27 | -0.18 | -0.29 | -0.28 |
| a2ps | -0.27 | 0.03 | -0.16 | -0.04 | -0.57 | -0.01 | -0.39 | -0.40 |
| gnugo | -0.45 | 0.04 | -0.06 | 0.01 | -0.53 | -0.01 | -0.13 | -0.05 |
| time | 0.70 | -0.29 | 0.47 | 0.70 | -0.55 | 0.08 | -0.47 | -0.12 |
| nascar | -0.13 | -0.18 | -0.23 | -0.41 | -0.77 | 0.15 | -0.76 | -0.33 |
| wdiff | 0.04 | -0.23 | -0.02 | -0.50 | -0.89 | 0.18 | -0.89 | -0.66 |
| acct | -0.67 | 0.21 | -0.52 | -0.53 | -0.67 | 0.13 | -0.57 | -0.46 |
| termutils | -0.35 | 0.18 | -0.21 | -0.13 | -0.46 | 0.17 | -0.33 | -0.20 |
| flex | -0.79 | 0.07 | -0.70 | -0.54 | -0.88 | 0.08 | -0.78 | -0.62 |
| byacc | -0.11 | -0.01 | -0.08 | -0.20 | -0.72 | 0.05 | -0.42 | -0.40 |
| diffutils | -0.42 | -0.02 | -0.36 | -0.51 | -0.66 | -0.02 | -0.56 | -0.57 |
| li | -0.07 | -0.17 | -0.18 | -0.18 | -0.09 | -0.15 | -0.17 | -0.18 |
| espresso | -0.55 | 0.03 | -0.33 | -0.46 | -0.70 | 0.04 | -0.42 | -0.43 |
| findutils | -0.25 | 0.07 | -0.20 | -0.04 | -0.34 | 0.07 | -0.29 | -0.01 |
| compress | -0.72 | 0.04 | -0.63 | -0.49 | -0.89 | -0.09 | -0.85 | -0.63 |
| sudoku | -0.69 | 0.22 | -0.26 | -0.40 | -0.79 | 0.20 | -0.35 | -0.52 |
| barcode | -0.59 | 0.07 | -0.55 | -0.65 | -0.71 | 0.06 | -0.66 | -0.74 |
| indent | -0.64 | 0.04 | -0.45 | -0.17 | -0.69 | 0.05 | -0.48 | -0.16 |
| ed | -0.67 | 0.03 | -0.49 | -0.56 | -0.82 | 0.04 | -0.59 | -0.62 |
| bc | -0.72 | 0.04 | -0.56 | -0.57 | -0.75 | 0.05 | -0.58 | -0.59 |
| copia | -0.72 | -0.66 | -0.98 | -1.00 | -0.70 | -0.68 | -0.98 | -1.00 |
| userv | -0.49 | 0.02 | -0.35 | -0.40 | -0.57 | 0.04 | -0.39 | -0.39 |
| ftpd | -0.74 | 0.03 | -0.53 | -0.40 | -0.78 | 0.02 | -0.57 | -0.42 |
| gnuchess | -0.54 | 0.07 | -0.47 | -0.31 | -0.55 | 0.06 | -0.48 | -0.29 |
| go | -0.49 | 0.03 | -0.16 | -0.31 | -0.58 | 0.04 | -0.18 | -0.33 |
| ctags | -0.42 | 0.03 | -0.18 | -0.23 | -0.53 | 0.04 | -0.23 | -0.24 |
| gnubg | -0.66 | -0.07 | -0.55 | -0.68 | -0.69 | -0.07 | -0.57 | -0.71 |
| average | -0.36 | 0.03 | -0.25 | -0.26 | -0.63 | -0.01 | -0.50 | -0.42 |
| **strongest** | 17 | 0 | 1 | 11 | 23 | 0 | 2 | 4 |

with the highest NOI value turned out to be a linchpin. What causes NOI to be the best estimator will be discussed in the next section.

As a statistical test to support our choice for NOI, we make a null-hypothesis that the other three metrics are at least equally good. For instance, consider NOI and NII with ENTR measure. Out of the 28 programs, in 22 instances NOI has stronger correlation than NII. Using Chernoff's bound we get that the probability of NII being at least as good as NOI is at most 0.01035. Chernoff's bound shows that NOI is in fact better with a probability of at least 0.9235 than any of the other three with respect to any of the considered measures.

Another interesting observation we made about the data is that for smaller programs the agreement between the NOI metric and both clusterization metrics was slightly better, suggesting that this heuristic will perform better for smaller programs. Figure 9 shows how the correlation values between NOI and ENTR as well as NOI and REGX change as a function of program size. Only programs with high clusterization are shown because in the other cases the relationship was not so evident. Particularly, from left to right, we can see the average correlation values for the programs ordered increasingly by their number of procedures. Although not drastically, but a tendency of worsening correlation

can clearly be observed. This could also indicate the need for combined identification of linchpins as outlined at the end of this section. The exact causes of this phenomenon are not clear yet, they are probably related to the different topologies of small and bigger programs.

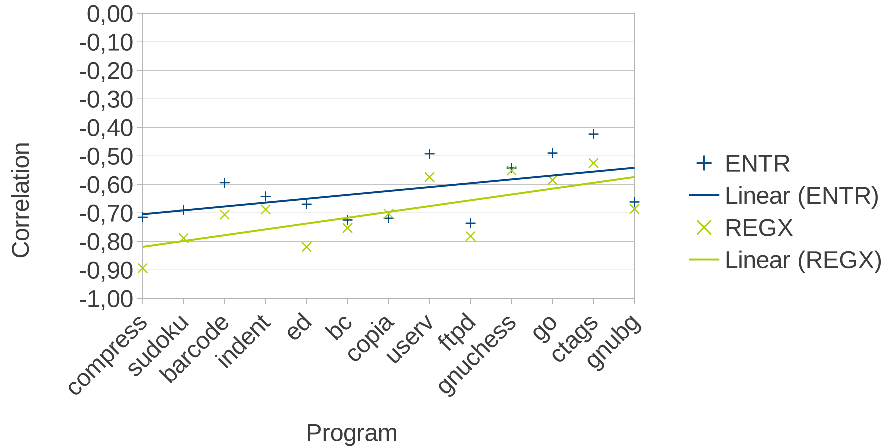



Figure 9: Correlation change with program size of highly clusterized programs

Once we got these results about the best linchpin estimator heuristic metric, we applied it to WebKit to see whether we can achieve significant ENTR or REGX metric reduction and hence potentially find linchpins in that system too. In the first instance we calculated the NOI metrics for WebKit and applied the filtered dependence set calculation excluding the first 10 procedures with highest NOI values individually, thus obtaining a set of 10 clusterization reduction values. Unfortunately, after this experiment we could not observe any notable improvement in clusterization: even the largest ENTR and REGX reductions were negligible and visual inspection could not reveal anything either. Then we tried the other heuristic metrics as well in a similar way, but we got even worse results, so we decided to continue the research with the combined exclusion of procedures as discussed in later sections of this paper.

### *5.3. On the connection between NOI and linchpin procedures*

To explain the high correlation between the NOI metric and clusterization we made a few observations and performed additional measurements as follows.

1. The high linchpin prediction capability of NOI was observed for dependence clusters computed based on forward dependences ($\vec{\mathcal{I}}$). Our first hypothesis was that there could be a dual property that linchpins in clusters based on backward dependences ($\overleftarrow{\mathcal{I}}$) could be predicted by high NII values. However, this turned out to be false, which directly follows from

the observation that the clusterization metrics for the two types of clusters were essentially identical (see Section 4.2).

Consequently, what we only have to show is why procedures with small NOI values are unlikely to be linchpins in either of the dependence cluster types. We use results of our study on the structure of the (forward) $SEA$ relation, namely $SEA = CALL \cup RET \cup SEQ \cup ID$. Clearly, the ID component of the relation does not affect the clusterization aspects, so we concentrate on the typical composition of $SEA$ sets regarding calling structures and sequential executions.

2. We empirically investigated how the $SEA$ sets of procedures with high NOI values are typically composed. We found that the ratio of $CALL \cup RET$ in the $SEA$ relations positively correlates with NOI: Pearson correlation was between 0.45–0.87 with the median of 0.58 for programs where the correlation of NOI to the linchpins was at least 0.5. In other words, a large NOI value implies a small $SEQ$ component in the $SEA$ set and vice versa. This will be an important basis for our arguments that follow.
3. We illustrate the basic mechanisms how a linchpin node in the ICCFG graph breaks up a large dependence cluster. Figure 10 shows a typical situation. Assume that we remove a procedure $p \in P$, and this results in a significant change in the clusterization of the $SEA$ dependences in some metric. That is, there is at least one cluster $\mathcal{C}$ such that after removing $p$ it will fall apart into at least two subclusters, $\mathcal{C}_1$ and $\mathcal{C}_2$ (here $\mathcal{C}_1, \mathcal{C}_2 \subset \mathcal{C}$ and $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$). For example, there is an $a \in \mathcal{C}_1$ and a $b \in \mathcal{C}_2$ such that before the removal of $p$ we had $SEA(a) = SEA(b)$, but after the removal $SEA'(a) \neq SEA'(b)$ (here $SEA'$ denotes the new dependence relations after removal). In this case we also say that the removal of $p$ *separates* $a$ and $b$.

   Separation happens in the following way. There is an execution of the program in which first some part of $b$ is executed, then some part of $p$ is executed, finally, some part of $q \in SEA(b)$ is executed. However, after $p$ is removed, no part of $q$ is executed after any part of $b$, while $a$ still has a part that is executed before some part of $q$, hence, $SEA'(a) \neq SEA'(b)$. This will happen because $SEA(a)$ does not require $p$ to include $q$, while $SEA(b)$ does.
4. Now, observe the following important fact: if $(p, b) \notin CALL \cup RET$ and $(b, p) \notin CALL \cup RET$, then the removal of $p$ will not separate $b$ from any other $a \in P \setminus \{p\}$. Namely, by definition of SEQ, there must be a caller procedure $p'$ that calls $p$ and subsequently all other $q \in SEQ(p)$ as well, so no matter if we remove $p$ the other callees will still remain parts of the common dependence set.

   Let us illustrate this situation on a program from our data set, which is 'compress', the smallest program with high clusterization. Figure 11 shows a simplified version of this program's ICCFG graph (essentially a call-graph), in which procedure *compress* is the linchpin. The rectangular areas represent the clusters before (the two outer ones denoted by *cluster #1* and *cluster #2*) and after removing the linchpin (the inner boxes). This

procedure has the highest NOI value as expected, and if it is removed, all its callees will be separated from the original dependence cluster and it will significantly collapse.

However, consider now procedure *getbyte*, which also has the same large $SEA$ set (it is in the same cluster). The difference is that it comes with a large SEQ component and not many callers and callees so after removing it the dependence cluster size will be reduced but it will not be broken.

5. To summarize, the necessary properties for $p$ to be a linchpin are as follows. First, it has to have a large $SEA$ set (this is similar to one of the findings by Binkley *et al.* [8]). But merely the size of the $SEA$ set is not enough: we measured very low correlation (in the range 0–0.15) between the $SEA$ size and clusterization. This large $SEA$ set must include a large $CALL \cup RET$ component for $p$ to be a linchpin. If the NOI value of $p$ is small, we expect that a large proportion of its $SEA$ set is a SEQ component, which will not be separated by the removal of $p$ making it unlikely a linchpin (while we cannot completely rule out this possibility). On the other hand, if the NOI value of $p$ is large then the proportion of the $CALL \cup RET$ component of its $SEA$ will be large as well and the SEQ component relatively small. In this case it will be more likely that the removal of $p$ will separate many procedures.
6. Finally, a large $CALL \cup RET$ component does not necessarily have to come from a large number of direct calls (high NOI). Theoretically, it may happen that there are many more indirect calls and returns. But we found that procedures with higher NOI values do not include comparably more indirect calls: the ratio of $CALL \cup RET$ in all the transitively accessible procedures usually positively correlates with NOI (the median is 0.5).

This essentially explains why is high NOI a likely predictor for a linchpin. However, since this is very hard to show analytically for a general case, this finding should be generalized with caution. Theoretically, many different situations are possible for a general program structure, however it seems that for realistic programs these special properties of dependences are more probable. We expect that similar phenomena would be observed for slice-based dependence clusters as well, but this should be verified separately.

### *5.4. Reducing clusterization by sets of linchpins*

Linchpin identification and removal can be rephrased in graph theoretical terms as well. Given a graph $G$ on an $n$ element vertex set $V$, we say that $S \subset V$ is a separator set, if $G \setminus S$ contains only small connected components. If $G$ does not contain certain substructures (large excluded minors, to be precise), then it must contain a small separator set. Still, in many cases the size of the separator set grows with $n$, usually it is about $O(\sqrt{n})$ [31].

While clusters in a $SEA$ graph (or other graphs associated with a program) are more complex than connected components (for instance, most dependence graphs including $SEA$ are not transitive), it is easy to see an analogy between the two kinds of problems. We think that analogously to separation of graphs,

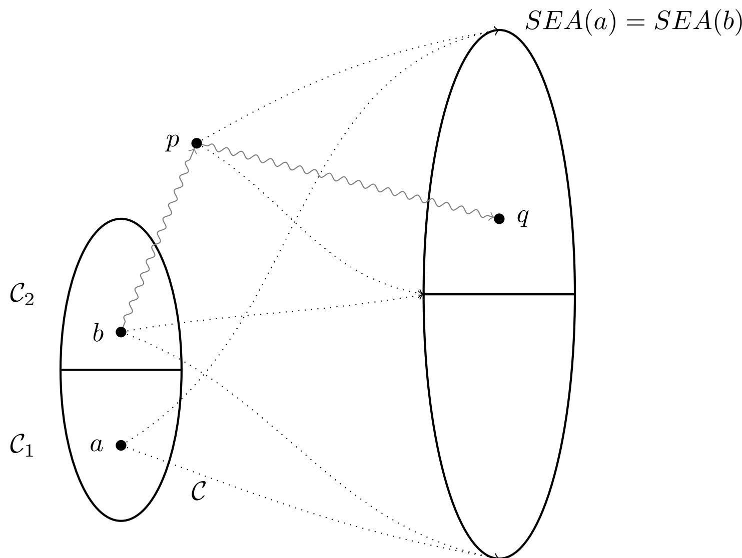



Figure 10: Illustration of how a linchpin breaks a dependence cluster

one cannot always expect to find a single linchpin vertex whose removal can significantly decrease clusterization. Rather, one should look for a hopefully small subset of vertices that somehow glue together the graph, and deleting them results in small clusters.

Graph theory also tells that not every graph has a small separator, for example, one has to delete a large number of vertices from a so-called expander graph in order to make it disconnected. We expect that the graphs associated with programs are not expander graphs, and therefore the deletion of a relatively small subset can reduce clusterization. Finding such a linchpin set effectively seems to be challenging.

To verify this theoretical concept, we performed empirical measurements with sets of linchpins as opposed to only one on some representative programs from our moderate size subjects. In this series of measurements, three programs from the medium level clusterization group (`findutils`, `termutils`, `nascar`), and three other from the high clusterization group (`go`, `ed`, `sudoku`) were selected. Moreover, care was taken to include programs with different sizes in both groups. We then took the first 10 procedures of a program with the highest NOI values, performed the calculation of the dependence sets while ignoring the first 0, 1, 2, . . . , 10 procedures together and investigated the resulting clusterization metrics. We were interested to see whether there was indeed one linchpin that brought significantly more gain alone than together with its followers, or were the differences not so significant between these first 10 candidate groups. (Note, that in Section 5.1 we investigated the individual gains of linchpin candidates, while here we are interested in their joint effect.)

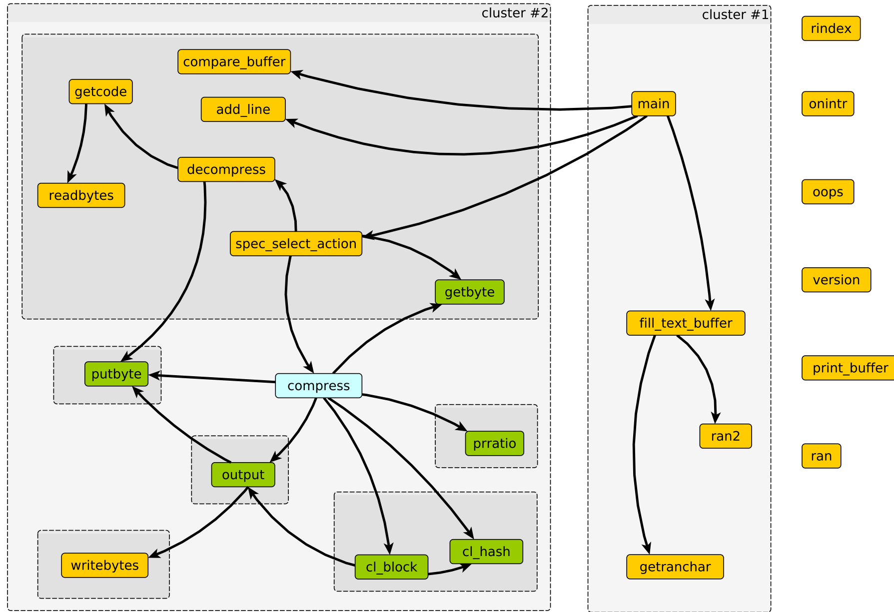



Figure 11: Simplified ICCFG (call) graph of program 'compress'

Figure 12 shows the ENTR metric for the cumulative removal of the first $k$ procedures ($k$-element linchpin sets), where the different $k$ values are represented on the horizontal axis. One can observe that at most the sets with the first three procedures are notable and require further investigation. It can also be observed that for the three programs of medium level clusterization, the overall decrease in the metric is less than for the other group, as expected. More interestingly, it seems that regardless of the level of clusterization, program size plays an important role in the rate of decrease. Consider programs `sudoku` and the ten times larger `go`, for instance. We can see that for `sudoku` a significant decrease of clusterization occurs right after the first procedure, while for `go` even the second procedure contributes to the decrease significantly. As an example from the other group, the same effect can be observed for `nascar` and `findutils`. (The anomalous behavior of increasing clusterization can be expected in certain cases, the effect is more pronounced in the case of `nascar` due to its small size.)

## 6. Cluster-related investigations in WebKit

In this section we present the results of our detailed investigations performed on the WebKit system. In particular, we deal with two topics: linchpin identification and the application of clusters in impact analysis.

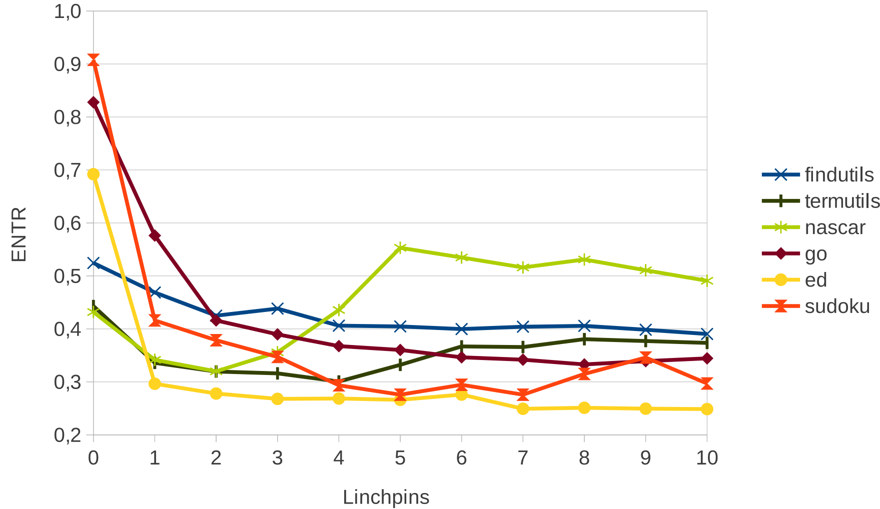



Figure 12: Changing of the gains for the first 10 linchpins removed together

*6.1. Linchpins in WebKit*

Findings from the previous section suggest that in many cases, especially for large programs, not only one program element (procedure) could be responsible alone for the formation of dependence clusters but a set of program elements together.

We performed similar experiments with the WebKit system in the hope to identify linchpin sets that significantly reduce clusterization. We expected that it would need more than only 2-3 procedures to achieve the same effect but we did not know how many. We tried removing several first procedures with the biggest NOI values but could not observe significant change in the clusterization up to until we removed about the first 200-250 procedures together. Removing 256 methods with the highest NOI values resulted in ENTR reduced by 7.2%, and REGX metric value reduced by 19.1% (AREA was reduced by 32.2%). In other words, the dependence sets collapsed this way, also changing dependence cluster formations, which can be observed in Figure 13.

Here, we can compare the original dependence clusters and the ones after removing these procedures (note, that in both cases the total number of procedures are represented on both axes, which is in the second case smaller by a comparatively negligible amount, hence the graphs are still comparable to each other). Although we cannot state that the clusters completely disappeared, the change is significant. To check if this result can indeed be attributed to the special role of the procedures with the top 256 NOI, we performed validation tests with three sets of randomly chosen 256 procedures. We found that randomly filtering procedures does not bring any improvement to the clusterization, change

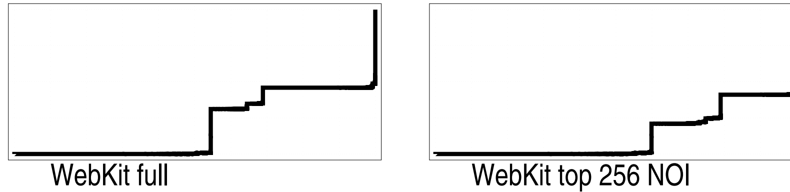



Figure 13: WebKit MSGs before and after removing top 256 NOI procedures

in ENTR was 0.04%, in REGX it was 0.25% on average.

It remains for future work to analyze in depth these linchpin sets and their effect on clusterization. From preliminary investigation, it seems that the clusters did not really disappear, they just changed their structure. Some of the procedures with high NOI could represent some very general connecting procedures, which do not really bear significant functionality (for example, we noticed a procedure that consisted of only a big switch statement and a huge number of method calls). When removing such procedures, the core dependences responsible for the main functionalities will remain, although the overall size will be smaller.

*6.2. Reduced $SEA$ sets in impact analysis*

One of the primary applications of dependence relations such as $SEA$ is in software change impact analysis [12]. Here one seeks to predict the required propagation of changes made to the system based on the initial changes and following the dependences (in this context, we call the ($SEA$) dependence set the *impact set*). If the change propagation is imperfect, then residual defects will be present in the system, so dependence based impact analysis has great significance in practice.

In an earlier work, we successfully applied $SEA$ sets for change impact analysis experimentation in the WebKit system [13]. In particular, we used real defect data based on regression test execution result histories, and related them to changes that introduced and later eliminated defects. We checked whether the $SEA$ impact sets of the failure introducing changes contained the modifications at the fixing revisions (we call this the *prediction capability* of impact analysis). In the present work, we recreate previous results using the reduced dependence sets of WebKit with the heuristic linchpin identification of the preceding section, and verify the effect of this reduction to the prediction capability.

Our basic approach is the following (the details are described in the mentioned publication). We identify revision pairs from the version control system, in which the first revision is treated as the failure introducing one and the second as the failure fixing one. This process is based on monitoring the regression test results and revisions where some of the test cases change their status from

'pass' to 'fail' or the opposite, respectively. Our hypothesis is that the impact set of the modified procedures at the revision the error was introduced contains the procedures that were modified between this point and the revision where it was fixed.

In the original experiment, we examined 33,542 revisions altogether, which corresponds to a development period of over a year. After we identified a number of revision candidates we had to filter them due to various reasons, so we arrived to a total list of 240 revision pairs for further investigation. We then computed the corresponding $SEA$ impact sets for the changes at the first element of each pair and counted the number of correctly predicted procedures by the $SEA$ set at the second element of the pair. The prediction numbers varied greatly within the whole range of 0-100%; the average was 83.9% with a deviation of 27.7%. This means that, in the original experiment, on average about 83.9% of the failure fixing procedures have been correctly identified by the corresponding impact sets. The downside was that the $SEA$ sets were often large. In terms of percentage of the total number of procedures it was just below 19% (17,203 procedures) on average. This can be a problem when using the impact sets for manual change propagation, but for other applications such as automatic regression test selection and prioritization they still can be useful. Consequently, methods to reduce the impact set sizes, such as based on dependence cluster and linchpin identification, are important in this application. Earlier we did not investigate the possibilities for the impact set size reduction, but the results from the preceding section could be reused for this purpose.

We performed the above experiment again, but this time using the reduced set of $SEA$ dependences after ignoring our heuristic linchpin candidates: procedures with the top 256 NOI values. On average, the impact sets of the changes at the investigated revisions became smaller by 12.4%. At the same time the overall prediction reduced to 77.7% (from 83.9%), but if we look at individual changes we see that in 77.5% of the cases there was no change in the prediction, while a significant reduction in impact set size could be observed. There were very few cases when the prediction degraded significantly, in only 7 cases the prediction reduced by more than 50%. We attribute this result also to the special nature of the removed dependences, namely it seems that they were too general for these particular cases in change propagation.

To go into the details of dependence set size reduction and prediction capability of impact analysis, Figure 14 shows the data points for all 240 investigated revision pairs on an XY plot. Each dot corresponds to one measurement point where the X dimension shows the reduction rate of the prediction, while Y is the set size reduction rate, both in percentages. We can observe that even in the case of more significant size reduction the predicition capability did not reduce significantly: there were many data points with 0 prediction loss with various size reductions. Generally, any reduction in impact set size is welcome even if it involves a certain amount in precision loss, but data points close to the Y axis are especially good.

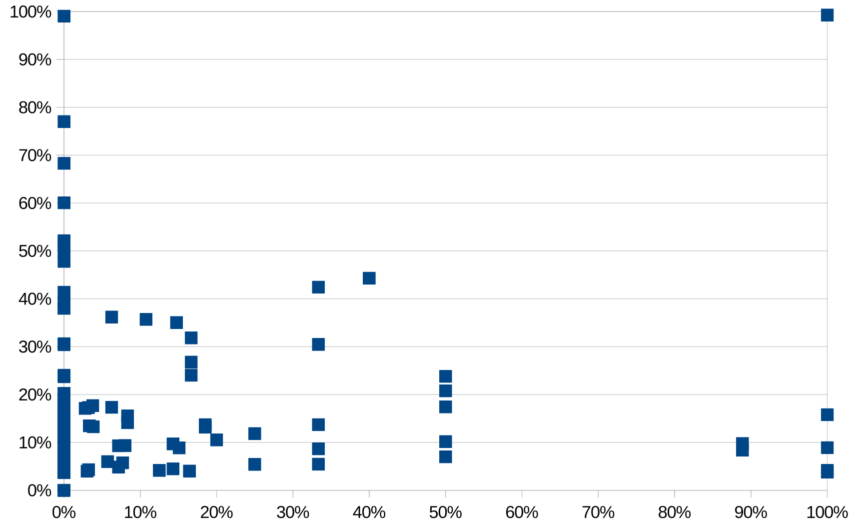



Figure 14: Dependence set size reduction and impact analysis prediction capability (X: prediction loss, Y: reduction rate)

## 7. Threats to validity

We believe that the range of subject programs we used is representative for C/C++ as it ranges over various sizes and the domains are different. However, it would be important to see whether these findings can be generalized to other languages and types of dependences. The imprecision of our $SEA$-based dependences could slightly distort the results due to dependences that could have been avoided using a more precise method. However, as previous research showed [11], such false dependences are expectedly tolerable. We did not investigate if similar results would have been obtained using different, for instance, slice-based clusters.

We generalized our findings regarding the heuristic method based on the NOI metric to the WebKit system. We could not verify how good this heuristic actually performs on this system as we do not know the exact linchpin data, we could only state that some improvement has been obtained.

Our findings related to the presence of linchpin sets instead of individual linchpins showed that this is more probable with bigger programs. However, this highly depends on the system itself so this observation should be generalized with caution.

We verified the results in an application related to change impact analysis, however it would be beneficial if more applications are tried out to find out about practicaly uses of dependence clusters in general.

## 8. Discussion and conclusions

This paper presented an empirical investigation of *SEA*-based dependence clusters. We may now answer the research questions set forth in this paper, however, our contributions to the better understanding of dependence clusters raised a number of additional questions.

Answering RQ1, we found that large dependence clusters occur frequently in programs regardless of their domain and size, however there are also programs which exhibit very little clusterization. We gave precise definitions for four clusterization metrics and used them throughout to evaluate the results of our experiments. The results so far enable us to expect that the clusterization of programs can be measured with greater precision in the future. Additionally, we plan to experiment with more complex metrics that are less sensitive to the cluster- and dependence set sizes.

RQ2 dealt with identifying one or more linchpins in programs. We were able to identify linchpin procedures using the brute-force method for all moderate size programs, but it is still to be explored in which cases we must seek for multiple linchpins and not only one. What we found is that as the program size increases it is less probable that only one program element is responsible for the formation of clusters.

Exhaustive exploration of possible linchpin combinations is computationally even more hopeless than for a single case. Hence additional, more sophisticated heuristic methods should be explored. Specifically, analyzing certain properties of the dependence graphs in terms of the graphs' topology is promising. Answering RQ3, we found that a specific metric based on graph structure, the Number of Outgoing Invocations for a procedure (NOI) is quite a good heuristic estimator for the linchpin, and the highest NOI value indeed predicts linchpins correctly in most of the cases. We also tried to explain what causes this metric to be such a good predictor. Investigating other more sophisticated but still efficient methods is among our future plans, including the use of machine learning classifiers based on graph topology properties. Involving the hierarchical structure of the programs (packages, classes and methods) could also be promising.

Assessing the fitness of a particular heuristics is not simple. In the case of large programs where we do not really know what *are* the linchpins, we can only indirectly assess the heuristic. The correlation of our linchpin prediction metric to the clusterization metrics gradually degraded as the system size grew. For WebKit, we could achieve significant clusterization reduction (although the three large clusters did not completely vanish) only after we removed the first 256 procedures with the highest NOI metric.

We are still looking for the causes of the dependence clusters in WebKit, however. Although our heuristics gave interesting insights into the structure of these clusters, further investigation of the internals of this software is required. For this task we will consult WebKit developers, who already noticed that there are some elements in the identified clusters which they cannot explain. We need to investigate whether these are the consequence of the imprecision of the *SEA*

algorithm or they represent some more hidden dependences in the system.

Recent advances in dependence cluster research with slice-based clusters will give additional themes for future research, for instance the identification and analysis of coherent clusters [21], which will probably be observable in $SEA$-based clusters as well, but it is a question if they would better represent logical structures in code.

We think that further research is needed to find out the connection of dependence clusters and the structures identified by methods of community structure analysis on software dependence graphs [19, 20]. Since this method is based on probabilities and is “less strict” we could possibly reuse related results in linchpin identification.

In this work we did not systematically investigate to decide if a dependence cluster reflects a dependence pollution [2] or not, and if the associated linchpins could be actually refactored, which is one of our future research directions.

A possible way to assess the role of dependence clusters would be through actual software engineering applications. In this work we started this line of research with change impact analysis (RQ4) and found that dependence set sizes can be significantly decreased by removing linchpin candidates (by about 15%) without seriously affecting the prediction capability of impact analysis. We speculate that by removing linchpins (which are very general, collecting program points like main control flow routers) the program dependences become more specialized, but the actually important dependences are retained. However, we need to perform additional experiments on other real life problems to verify the experimentation results, *e.g.* for test case prioritization [13]. A study involving human evaluation could be a possible way as well to gain more insight into the benefits and risks related to dependence clusters.

### Acknowledgements

The authors would like to thank Z. Herczeg, L. Langó, Cs. Osztrogonác, J. Taylor, D. Tengeri and B. Vancsics for their supporting work in this research. This research was partially supported by the Hungarian national grant GOP-1.1.1-11-2011-0049 and the EU FP7 STREP project REPARA (ICT-609666).

### References

[1] D. Binkley, Source code analysis: A road map, in: Proceedings of 2007 Future of Software Engineering (FOSE’07), IEEE Computer Society, 2007, pp. 104–119.

[2] D. Binkley, M. Harman, Locating dependence clusters and dependence pollution, in: Proceedings of the 21st International Conference on Software Maintenance (ICSM’05), 2005, pp. 177–186.

[3] M. Acharya, B. Robinson, Practical change impact analysis based on static program slicing for industrial software systems, in: Proceedings of the 33rd

ACM SIGSOFT International Conference on Software Engineering (ICSE), 2011, pp. 746–765.

[4] D. Binkley, M. Harman, Identifying 'linchpin vertices' that cause large dependence clusters, in: Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09), 2009, pp. 89–98.

[5] D. Binkley, M. Harman, Y. Hassoun, S. Islam, Z. Li, Assessing the impact of global variables on program dependence and dependence clusters, Journal of Systems and Software 83 (2010) 96–107.

[6] M. Harman, D. Binkley, K. Gallagher, N. Gold, J. Krinke, Dependence clusters in source code, ACM Transactions on Programming Languages and Systems 32 (2009) 1–33.

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

[8] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, Z. Li, Efficient identification of linchpin vertices in dependence clusters, ACM Trans. Program. Lang. Syst. 35 (2013) 7:1–7:35.

[9] GCC, GCC, the GNU Compiler Collection, `http://gcc.gnu.org/`, 2014. Last visited: 2014-06-12.

[10] WebKit, The WebKit open source project, `http://www.webkit.org/`, 2014. Last visited: 2014-06-12.

[11] J. Jász, Á. Beszédes, T. Gyimóthy, V. Rajlich, Static Execute After/Before as a replacement of traditional software dependencies, in: Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08), 2008, pp. 137–146.

[12] S. A. Bohner, R. S. Arnold (Eds.), Software Change Impact Analysis, IEEE Computer Society Press, 1996.

[13] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, T. Gyimóthy, Impact analysis in the presence of dependence clusters using Static Execute After in WebKit, in: Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'12), 2012, pp. 24–33.

[14] Á. Beszédes, L. Schrettner, B. Csaba, T. Gergely, J. Jász, T. Gyimóthy, Empirical investigation of SEA-based dependence cluster properties, in: Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13), 2013, pp. 1–10.

[15] M. Weiser, Program slicing, IEEE Trans. Softw. Eng. SE-10 (1984) 352–357.

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

[17] Á. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, V. Rajlich, Computation of Static Execute After relation with applications to software maintenance, in: Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07), 2007, pp. 295–304.

[18] Á. Hajnal, I. Forgács, A demand-driven approach to slicing legacy COBOL systems, Journal of Software: Evolution and Process 24 (2012) 67–82.

[19] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the Bunch tool, IEEE Transactions on Software Engineering 32 (2006) 1–16.

[20] J. Hamilton, S. Danicic, Dependence communities in source code, in: Proceedings of the 28th International Conference on Software Maintenance (ICSM'12), Early Research Achievements track, 2012, pp. 579–582.

[21] S. Islam, J. Krinke, D. Binkley, M. Harman, Coherent clusters in source code, Journal of Systems and Software (2013) –. Available online at http://dx.doi.org/10.1016/j.jss.2013.07.040.

[22] M. S. Hecht, Flow Analysis of Computer Programs, Elsevier Science Inc., New York, NY, USA, 1977.

[23] J. Jász, Dependence-based static program slicing and its approximations, PhD dissertation, University of Szeged, 2009.

[24] S. Islam, J. Krinke, D. Binkley, Dependence cluster visualization, in: Proceedings of the 5th international symposium on Software visualization (SOFTVIS'10), 2010, pp. 93–102.

[25] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, T. Gyimóthy, Code coverage-based regression test selection and prioritization in WebKit, in: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12), 2012, pp. 46–55.

[26] GrammaTech CodeSurfer Homepage, Homepage of GrammaTech's CodeSurfer, http://www.grammatech.com/research/technologies/codesurfer, 2013. Last visited: 2013-05-08.

[27] R. Ferenc, Á. Beszédes, M. Tarkiainen, T. Gyimóthy, Columbus – reverse engineering tool and schema for C++, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 172–181.

[28] SoDA, SoDA library, http://soda.sed.hu, 2014. Last visited: 2014-06-11.

[29] B. Csaba, L. Schrettner, Á. Beszédes, J. Jász, P. Hegedűs, T. Gyimóthy, Relating clusterization measures and software quality, in: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13), Early Research Achievements Track, 2013, pp. 345–348.

[30] N. Alon, J. H. Spencer, The Probabilistic Method, John Wiley and Sons, Inc., Hoboken, NJ, USA, 2008.

[31] K. Kawarabayashi, B. Reed, A separator theorem in minor-closed classes, in: Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on, Oct., pp. 153–162. doi:10.1109/FOCS.2010.22.