

Identifying ‘Linchpin Vertices’ that Cause Large Dependence Clusters

David Binkley and Mark Harman
King’s College London

Centre for Research on Evolution, Search and Testing (CREST)
Strand, London, WC2R 2LS, UK

Abstract—A dependence cluster is a maximal set of program components that all depend upon one another. Previous work has highlighted the prevalence of large dependence clusters in source code, presenting potential problems for comprehension, testing, and maintenance. This paper is concerned with source code analysis techniques for identifying the causes of large dependence clusters. The paper presents results of a study of low-level causes of dependence clusters, which reveals that a large cluster can be caused by the smallest atomic unit source code: a single vertex or edge of the program’s dependence graph. These are termed the *linchpin* vertices and edges in this paper.

I. INTRODUCTION

Dependence analysis plays a role in many forms of source code analysis and manipulation. For example, it has been linked to ease of program understanding [2], [8], used to delimit the changes that may be performed [12], [25], and to capture the impact of a change [7].

This paper is concerned with one of the potential issues that can arise because of the structure of dependence in source code; the dependence cluster, and specifically the causes of large dependence clusters. A *dependence cluster* is a maximal set of program components where each depends on the others. Dependence clusters can have a negative impact on both programmer effectiveness and dependence-based static analysis tools. For example, consider a source code manipulation tool designed to extract a selected subcomponent from a program. If this subcomponent overlaps a dependence cluster then the tool must extract the entire cluster in order to preserve the semantics of the subcomponent.

Previous work provides evidence of the widespread presence of very large dependence clusters in source code from a wide range of open source and commercial software [14]. The authors believe that large dependence clusters are a form of dependence anti-pattern [3], which have been shown to have a negative impact on software engineering concerns, such as software maintenance [4].

The empirical observation of the prevalence of large dependence clusters, coupled with the belief that they are potential sources of problems, suggests an investigation into the causes of dependence clusters. The initial work on dependence cluster causes [5] revealed evidence that global variables can be the source of capillary data flows [4]. That

is, through its global scope, the definitions and uses of a global variable can, figuratively speaking, ‘sew together’ islands of dependence into one very large cluster of mutual dependence, thereby forming a large dependence cluster.

Unfortunately for the source code maintainer, re-writing the program to avoid the use of these cluster-causing global variables can be a complex and error-prone activity; such globals are typically defined and used at multiple disparate sites in the source code. By contrast, for those dependence clusters where the cause could be isolated to a single dependence-graph vertex, it may be easier to understand, isolate, and ameliorate the effects of the cluster. This paper focuses on these low-level causes of dependence clusters. The paper presents empirical evidence that single vertices within a program’s dependence graph can hold together very large clusters of dependence.

The primary contributions of the paper are as follows:

- 1) The paper introduces a technique for measuring the effect of a single vertex and a single edge on dependence levels and uses this to identify causes of dependence clusters using a graph marking procedure.
- 2) The paper presents results that reveal how the dependence on these low-level features follows a power law, with a very small number of vertices being responsible for holding together the vast majority of the dependence connections in a program.
- 3) The paper presents results of an empirical study of 44 programs, which consist of 293KLoC, revealing the *linchpin* vertices that cause large dependence clusters.

The remainder of this paper first formalizes the notion of dependence cluster and summarizes prior work on their identification and size. Then, in Section III, the core of the paper presents results from the search for dependence cluster causes. Section IV considers related work and finally Section VI summarizes the paper.

II. DEPENDENCE CLUSTERS

Two common dependences used in static analysis are the *data dependence* and *control dependence*. A *data dependence* connects a definition of a variable with each use of the variable reached by the definition [10]. A *control dependence* connects a predicate p to a component c when p has at least two control-flow-graph successors, one of which can lead to the exit node without encountering c and the

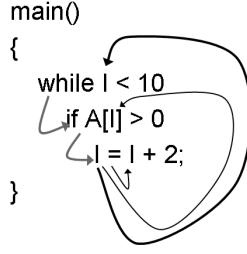


Figure 1. A cluster created from data (black) and control (grey) dependencies.

other always leads eventually to c [1], [16]. For structured code, control dependence reflects the nesting structure of the program.

Figure 1 shows several example dependencies that form a simple dependence cluster. In this program, the predicate $l < 10$ data depends on the assignment to l , this assignment control depends on the predicate of the if statement, and the if control depends on the predicate $l < 10$. Thus, the three statements are tied together into a cluster by their dependencies.

Recent empirical work has shown that dependence clusters are surprisingly large and prevalent [4]. On a small scale, programmers and tools can overcome the impact of large dependence clusters. However, this ability diminishes as program size increases. Thus, a challenge in scaling the software development process is the dealing with large dependence clusters.

A. Dependence Cluster Identification

This section formalizes the definition of dependence cluster and then presents results from an empirical search for them. A naïve definition of a dependence cluster would be based on transitive closure of the dependence relation and thus would define a cluster to be a strongly connected component. Unfortunately, for certain language features, dependence is not transitive. Examples of such features include procedures [17] and threads [19]. Thus, in the presence of these features, strongly connected components overstate the size and number of dependence clusters.

To illustrate, consider the program shown in Figure 2. This code includes the following two sequences of data dependencies: first, starting from the assignment of a , to the use of a as an actual parameter in the call to h , to formal parameter z , and then to the statement $z = z + 1$. There is also a sequence of data dependencies from this assignment, to the return value of the call in g , and then to the assignment of y . However, the concatenation of these two paths, while in the transitive closure of the dependence relation, ignores calling-context and thus does not reflect a valid execution of the program.. Therefore, forming strongly

```

void f(int a)
{
    a = 42
    b = h(a)
}

void g(int x)
{
    x = 1968
    y = h(x)
}

int h(int z)
{
    z = z + 1;
    return z;
}

```

Figure 2. An example illustrating how the data dependence relation is not transitive in the presence of procedures. In this example there is a path of dependencies from the assignment to variable a through function h to the assignment of variable y that does not correspond to a legal execution of the program.

connected components wrongly places the assignments $a = 42$ and $y = h(x)$ in the same cluster.

Fortunately, program slicing, a context-sensitive interprocedural dependence analysis, captures the necessary context information [17], [19]. Intuitively, a slice extracts a sub-program that captures a semantically meaningful sub-computation from a program. Slices can be efficiently computed using a two-pass reachability algorithm over a program’s *System Dependence Graph* (SDG) [17].

Using slicing, dependence clusters can be defined as maximal sets of statements that all have the same slice. In practice, the identification is facilitated by the observation that *same slice* can be approximated by *same slice size* [4]. That is, two program components that have the same slice size are deemed to reside in the same cluster. This is a conservative approximation because two slices may differ, yet, coincidentally, have the same size. However, two identical slices must have the same size. In practice, the approximation is over 99% accurate [14].

This approximation also provides a useful visualization: the *Monotone Slice-size Graph* (MSG) [4]. An MSG is a graph of slice sizes plotted in monotonically increasing order on the x -axis. The y -axis measures slice size. That is, slices are sorted according to increasing size and the sizes are plotted on the vertical axis against slice number, in order, on the horizontal axis. To facilitate comparison of MSGs, the MSGs shown in this paper use the percentage of the slices taken on the x -axis and the percentage of the entire program on the y -axis.

The MSG provides a visual aid to dependence cluster identification because dependence clusters appear as a sheer-drop cliff face followed by a long flat plateau. To illustrate,

III. DEPENDENCE CLUSTER CAUSES

A. Research Question

Having briefly reviewed the literature which contains evidence that dependence clusters are common and noting that they have a negative impact on software construction, there is a natural question:

“What causes large dependence clusters?”

This paper seeks to answer this question by introducing and evaluating techniques for source code analysis that search for dependence cluster causes among the lowest level features of source code. The results indicate the presence of a form of ‘*source code butterfly effect*’ through which the very smallest items of source code (a single vertex or edge of the program’s dependence graph) can influence the very largest whole-program-level dependence structures.

Identifying cluster causes provides a first step towards improving static analysis tools, which must cope with programs that contain large dependence clusters. One way of characterizing the search is as the search for a *linchpin component*. That is, a single component that holds a dependence cluster together. If such components exist, then ignoring its dependences will cause the dependence cluster to disappear. For example, if the component is a vertex, then ignoring its dependences amounts to treating the vertex as if it uses no variables and is not control dependent on any other vertex.

B. Experimental Set Up

While partially successful, an initial manual search proved too time consuming and too error prone; thus, an experiment that employed a semi-automated exhaustive search process was set up. Search results are evaluated by measuring the *area under the MSG*, as illustrated in the top chart of Figure 5. A *reduction* in the area is a necessary but not sufficient condition for identifying a *linchpin* vertex or edge. This is because there are two possible outcomes: a *drop* and a *break*. These two are illustrated by the center and lower MSGs shown in Figure 5. Both show a reduction in area; however, the center MSG reflects only a reduction in the size of the slices making up the cluster. Only the lower MSG shows a true breaking of the cluster. From these MSGs, it is clear that a reduction in area must accompany the breaking of a cluster, but does not imply the breaking of a cluster.

Starting ‘small’, the search for *linchpin* components begins by considering individual vertices and edges from the program’s *System Dependence Graph* (SDG) [17]. The vertices of an SDG represents the statements and predicates of the program. They are similar to the nodes of a control-flow graph. The edges of an SDG represent the intraprocedural and interprocedural control and data dependences between components. Figure 6 shows the 44 programs whose SDGs were analyzed. While these include many of the programs studied in previous work, they also include some new programs and include both C and C++ code.

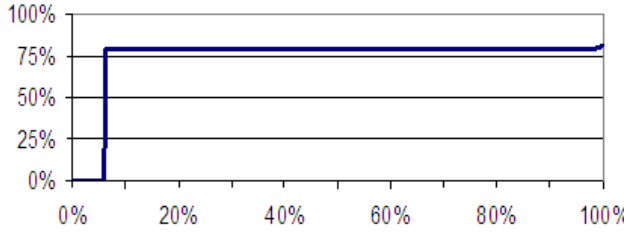


Figure 3. An MSG with a large dependence cluster.

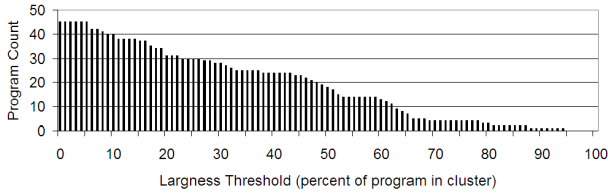


Figure 4. For *large* ranging from 0 to 100% of a program, the *y*-axis shows the number of the 45 programs with at least one large cluster.

Figure 3 shows an example MSG where, reading along the horizontal axis, approximately the first 8% of slices are very small, after which the MSG reveals a sharp increase to just over 75% of the program. Most of the remaining slices are essentially the same size though there is a small up-tick at the far right of the MSG.

The MSG visualization helps in the inherently subjective task of deciding whether a cluster is large (how long is the plateau at the top of the cliff face relative to the surrounding landscape?) and whether it denotes a discontinuity in the dependence profile (how steep is the cliff face relative to the surrounding landscape?).

Looking at a collection of 45 programs, which contain just over one million lines of C code, a surprisingly large number of large clusters were found to be present in many programs [4], [14]; the prevalence and size of dependence clusters should be even higher in program’s with greater *data-flow uncertainty* such as object-oriented programs. The 45 programs ranged in size from 600 LoC to almost 180 KLoC and cover a range of application domains such as utilities, games, and operating system code.

Figure 4, taken from the authors’ previous work [14], quantifies the data by showing the number of programs with at least one large cluster for a range of values for *large*. From the figure, all 45 programs include a cluster of at least 5% of the program. Two-thirds have a cluster of at least 20% of the code and one, at the far right, includes a cluster containing 94% of the program.

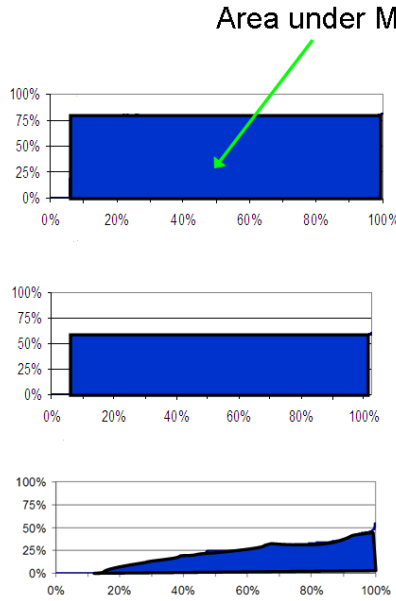


Figure 5. The area under the MSG drops under two conditions: the slices of the cluster get smaller (center MSG), or when the cluster breaks (lower MSG). Thus, a reduction in area is a necessary, but not sufficient, condition for a cluster breaking.

The search for dependence clusters is conducted by computing the MSG while *ignoring* the dependence associated with a specific component. Here, a single dependence edge represents the smallest component considered. If all of the dependence ‘flows’ through a single edge then ignoring this edge will break the flow and thus break the cluster. A vertex, which has a collection incoming and outgoing edges, is the next smallest component considered. For a vertex, it is sufficient to ignore either the vertex’s incoming or outgoing dependence edges. Without loss of generality, the experiments ignore the incoming dependence edges.

Rather than starting with the edges, vertices are considered first. This is done for two reasons: first, it is conceptually simpler to present the impact on dependence clusters of a vertex because each vertex maps to a particular program component. In contrast, an edge represents an interaction between two components.

The second reason is efficiency. Recomputing the MSG while ignoring the dependence represented by an edge or a vertex is marginally expensive; thus, it is advantageous to avoid unnecessary work. By construction the reduction caused by ignoring the dependences associated with an edge can be no greater than that caused by ignoring the dependences associated with its target vertex. This is because ignoring the dependence associated with a vertex ignores the dependence associated *all* its incoming edges. Therefore, all the vertices were processed first and then only edges targeting a vertex associated with a significant reduction were considered.

name	LOC	vertices	edges
pos	230	127	282
conversion	237	538	1,358
universal	276	420	908
replace	563	867	2,766
apartment	592	808	1,664
driver	604	373	801
sudoku	703	272	526
protest	755	601	1,265
h_server	792	617	1,282
address	890	738	1,315
sudoku1	915	292	544
compress	1,234	1,085	3,071
copia	1,170	4,686	13,671
fass	1,140	922	2,264
interpreter	1,560	947	2,286
lottery	1,365	1,004	2,517
nascar	1,097	996	2,063
pc2c	1,238	1,749	2,454
tile-forth	3,717	12,076	43,881
gcc.cpp	4,079	7,460	34,971
which	4,880	1,163	3,322
barcode	5,562	3,909	12,818
wdiff.0.5	5,958	2,421	7,315
byacc	6,337	10,151	34,129
indent	6,100	6,748	152,786
termutils	6,697	3,113	9,632
time-1.7	6,033	1,044	3,210
gnubg-0.0	7,229	9,556	30,762
userv-0.95.0	7,150	12,517	48,219
epwic-1	8,631	12,492	44,434
acct-6.3	9,536	7,250	31,527
space	9,126	11,277	36,356
cadp	11,068	15,672	45,472
ed	12,493	16,533	83,009
bc	14,609	5,133	17,057
oracolo2	14,326	11,812	37,593
prepro	14,328	11,745	37,301
flex2-4-7	15,143	11,104	53,056
ftpd	15,914	25,018	160,737
findutils	16,891	14,445	81,318
gnuchess	16,659	15,069	54,791
wpst	17,321	20,889	80,053
diffutils	18,374	17,092	92,544
flex2-5-4	20,252	14,114	99,126
sum	293,774	296,845	1,376,458

Figure 6. Characteristics of the subject programs studied.

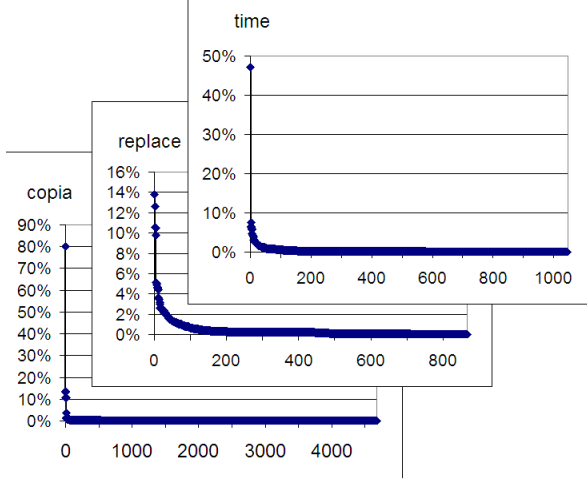


Figure 7. Fall-off in reduction for three example programs. Note that the scale for the percent reduction shown on the y -axis is not the same in the three graphs. The x -axis shows the vertex number after sorting vertices into monotonically decreasing order based on the reduction they cause.

C. Results: Linchpin Vertices

To begin with, the dependence associated with each vertex is ignored in turn. To better understand the potential impact that this has on a dependence cluster, consider the two extreme possibilities. At one extreme, a vertex v is on *one of many* paths connecting the vertices of a cluster. In this case ignoring v 's dependences does not impact the cluster at all because alternate paths exist. At the other extreme, the set of paths necks down to *a single path* that contains v . In this case, ignoring v 's dependences has a significant impact: it removes the cluster completely. In between, it is possible to have redundant paths that connect parts of a cluster and thus the impact of ignoring the dependences associated with a vertex can range between the two extremes.

The investigation starts by considering, in turn, the reduction in the area under the MSG associated with each vertex, it then looks more closely at those vertices that produce a large reduction. Finally, the investigation considers and categorizes the MSGs for the largest reductions.

To begin with, all the reductions for three representative programs are shown in Figure 7. Each chart shows, in monotonically decreasing order, the reduction in area under the MSG obtained by ignoring the dependences of each SDG vertex. All programs are dominated by a long *almost zero* tail on the right. Thus, at most a small handful of vertices cause a significant reduction. For example, this pattern is seen in the three programs shown in Figure 7. Given that a vertex is a rather small part of an SDG, that such vertices exist is, in and of itself, interesting.

Considering all 296,845 vertices collectively, only 1611 (0.54%) cause a reduction greater than 1%. Those causing at least a 10% and at least a 20% reduction number 71

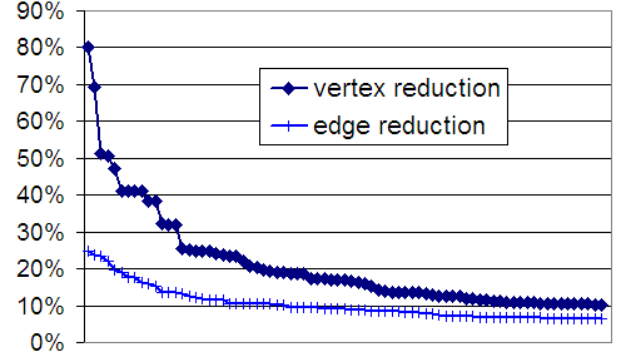


Figure 8. Top reduction causing vertices and edges. The y -axis shows the percent reduction and the x -axis the top 71 reduction-causing vertices and edges.

(0.024%) and 21 (0.0071%), respectively. The reduction caused by the 71 vertices producing at least 10% reduction is shown in Figure 8. Details of the top 21 reduction-causing vertices, which cause a reduction of at least 20% are presented in the top of Figure 9. The most striking commonality of the *linchpin* vertices shown in the top of Figure 9 is that they are almost exclusively control points. Only two non-control vertices (a function return value and the expression `done++`) occur in the top 21. In contrast when considering the 50 *linchpin* vertices causing between 10% and 20% reduction, control has lost its prominence. A selection of these vertices is shown in the bottom of Figure 9. Because the top of the figure is dominated by the smaller programs, this selection favors the larger programs, but the pattern is the same with the smaller programs as well.

The final step in the analysis is to generate and examine the MSGs for those vertices that cause a large reduction in the area under the MSG. From these MSGs it is possible to categorize each reduction as cluster *breaking* or simple producing a *drop*. Figure 10 shows several example MSGs for vertices that produced a large reduction. The top chart, for the program `time`, shows the classic example of a small cluster being completely broken. While the largest slices in this MSG are essentially unchanged in size, the cluster (evidenced by the plateau in the middle of the graph) is replaced by a gradual rise when the dependences for the vertex representing `switch(*_fmt)` are ignored. This vertex is in a loop that processes a string. The string (similar to the first argument of `printf`) determines how to format the time begin output. The loop and `switch` tie together all of the formatting cases.

In the center chart, the same pattern is seen on a larger scale with the vertex representing `copia`'s `switch` statement `switch(a)` (which is the core of a finite-state machine where `a` is the machine's next-state). Ignoring dependences associated with this vertex produces the lowest MSG shown in

Program	%age	Vertex Type	Source
For all programs, vertices producing at least a 20% reduction			
copia	80.20	control-point	switch (a)
fass	69.13	control-point	if(tmp[i-1]==':')
apartment	51.37	control-point	switch(choice)
fass	50.70	call-site	lookup()
time-1.7	47.06	control-point	switch (*++fmt)
apartment	40.96	control-point	while(choice != '5')
pos	38.21	control-point	if (bag_plus_lollie >= 10)
h_server	32.43	control-point	if(fp==NULL)
h_server	31.88	control-point	switch(request_no)
h_server	31.84	control-point	if(fp==NULL)
nascar	25.57	control-point	if(cars[x].is_done())
nascar	25.15	control-point	while(done<NUM_CARS)
sudoku1	24.82	control-point	while(!check_completed())
sudoku1	24.81	actual-out	check_completed()
nascar	23.98	control-point	if(count%mod==0)
pos	23.86	control-point	if (num_lollie<15)else
nascar	23.58	expression	done++
apartment	23.46	call-site	GuestMenu()
nascar	21.95	call-site	sort_display()
fass	20.80	control-point	if(flag==0)
fass	20.43	call-site	synt()
For larger programs only, vertices producing a 10% to 20% reduction			
replace	13.82	control-point	if (in_set_2())
copia	13.70	formal-in	int a
copia	13.66	actual-in	m
which	13.57	control-point	while (next)
copia	13.57	expression	m=urna[n]
copia	13.11	expression	urna[j]=probtabs[j].posizione
replace	12.58	actual-out	in_set_2()
findutils	11.92	expression	parse_function=find_parser()
findutils	11.91	actual-out	find_parser ()
findutils	11.63	indirect-call	(*parse_function) ()
copia	10.75	control-point	while (!iriga)
compress	10.73	expression	text_buffer[buindex]=c2
replace	10.56	actual-out	amatch()
compress	10.56	expression	c2=getranchar()
copia	10.55	formal-in	int riga
replace	10.49	expression	m = amatch()
compress	10.43	actual-out	getranchar()
which	10.11	call-site	find_command_in_path()

Figure 9. Vertices causing a large reduction.

the center chart. The chart also includes the MSGs produced when ignoring the dependence through the second and third largest reduction causing vertices. These two produce classic examples of drops in the size of the slices making up the cluster, but do not break the cluster. Such vertices segregate off a portion of the program, but do not remove the true cause of the cluster.

The bottom chart of Figure 10 shows the MSGs for the top three reduction causing vertices of the program **replace**. All three reflect the architecture of **replace**, which first generates an internal representation of a search pattern and then attempts to match this pattern. All slices taken with respect to vertices in cluster labeled **Match Pattern** include the vertices of those in the cluster labeled **Generate Pattern**.

While none of the three reductions shown provides significant evidence of cluster breaking, they are useful because they illustrate two patterns seen in non-cluster breaking reductions that can be directly tied back to the source code; thus, they serve to refine the notion of a *drop*. The first of these, termed a *shift*, is illustrated in the upper part of Figure 11. Ignoring the dependences associated with the vertex that represents the assignment of **amatch**'s return value to variable **m** essentially *cuts off* slices that

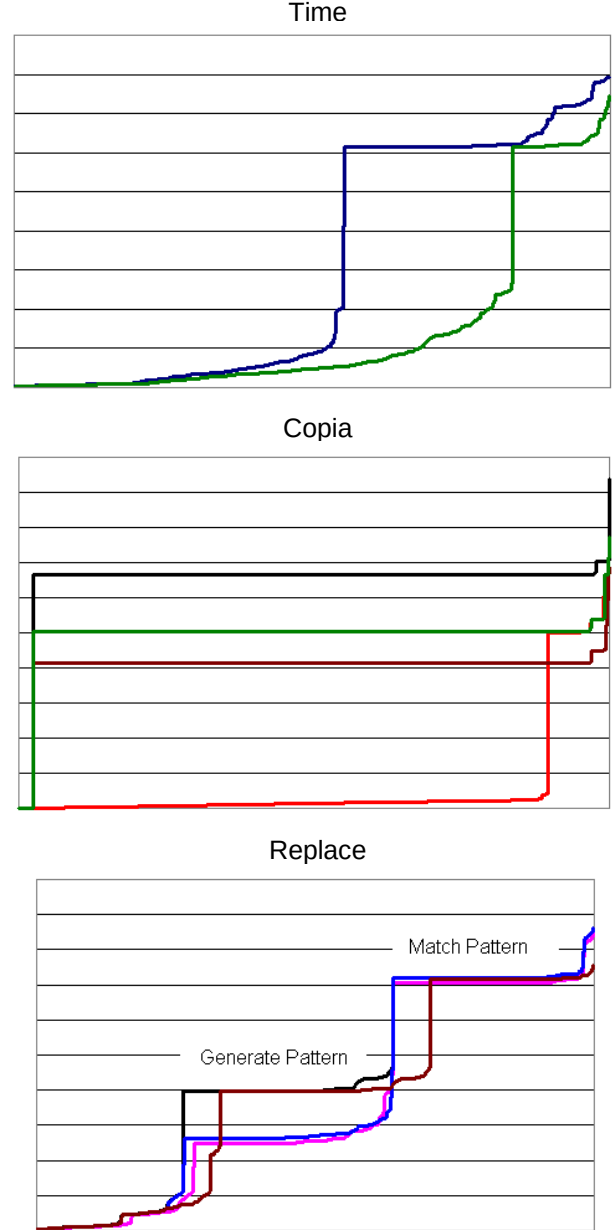


Figure 10. MSGs for high reduction vertices that show the different patterns seen in the dependence cluster search.

include uses of **m**. For example, the assignment **lastm = m** depends (through the assignment to **m**) on the results of calling **amatch** and consequently the computation it entails. Ignoring the dependence associated with the assignment to **m** makes the slice on **lastm = m** quite small because it omits the computation contained in **amatch**. This, in essence, replaces a large slice (that was part of the cluster) with a smaller slice and thus shifts the plateau of the dependence cluster to the right (as well as shortening its length).

Second, the example in the middle of Figure 11 shows two patterns: a *drop* and the combination of a *shift* and a *drop*. The drop comes from ignoring the actual parameter that rep-

resents the passing of `pat[lj]` to the function `in_set_2`. This effectively cuts the statements that represent the computation of `pat[lj]` out of all the slices that include this call. As the call is in the cluster, this removes the computation from all the slices of the cluster, which appears as a drop in the MSG (because all the slices are smaller).

This example also illustrates a case in which a *shift* and a *drop* occur together. In this case, ignoring the dependence associated with `if (in_set_2(pat[lj]))` includes the impact of ignoring the dependence of `pat[lj]` and also orphans statements such as `done = true` (in the same way `lastm = m` was orphaned).

The final example in Figure 11 includes the MSG shown in the center chart of Figure 10. It shows another drop (from ignoring the dependence of the formal parameter `a` (which holds the finite state machines next state) and the breaking of the cluster (from ignoring the dependence of the switch statement). Considering these two together is illustrative. In the program, each of the functions called from within the switch statement (e.g., `grid`) eventually leads to a recursive call to `seleziona`. This leads to almost the entire program being tied together via control dependence. In contrast ignoring the dependence of the formal parameter `a` (the FSMs next state) removes from each slice only those parts of the next-state computation not (recursively) connected to the switch via control dependence. This leads to the drop in the plateau of the MSG, but still leaves the bulk of the cluster connected. Comparing these two helps to illuminate why control vertices dominate the large reductions shown in Figure 9.

Generating the 71 MSGs for each of the 71 vertices that causes at least a 10% reduction produces examples of all of the patterns described above. To begin with, eight of the programs include no clusters and thus can only produce drops. Of the remaining 63, five cause complete cluster breaking (as seen with `copia`) and 13 partial cluster breaking (as seen with `time`). This is about 30% of the 71 vertices. The remaining vertices produce 23 drops and 22 shifts. In identifying drops and shifts the dominant feature was used, so no drop-shift combinations are reported.

D. Results: Linchpin Edges

The search of *linchpin* edges considers the 8,678 incoming edges of the 1,623 vertices that produced at least a 1% reduction in area under the MSG. This represents a tremendous savings in effort because the total number of edges in all the graphs is over 1.3 million. All the unconsidered edges, by definition, produce no more than a 1% reduction. Of the 8,678 edges considered, 5,106 (59%) led to *no* reduction at all, 2,682 (31%) led to between 0% and 1% reduction, 860 (10%) led to between 1% and 10% reduction, and 30 (0.34%) led to a reduction greater than 10% reduction. Figure 8 shows the reduction attained by the top 78 edges (the number of edges was chosen to match the number of

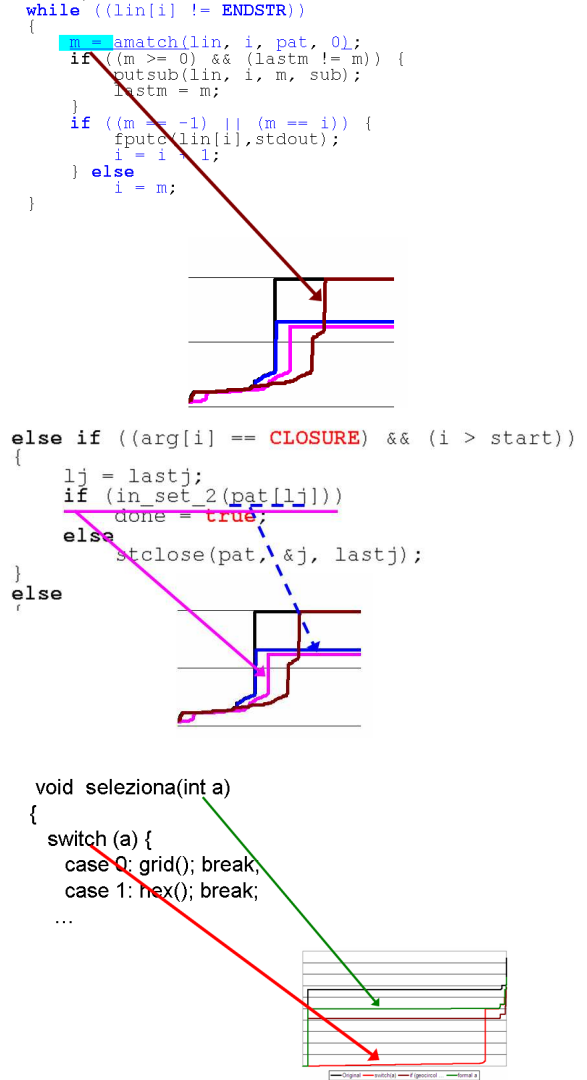


Figure 11. Source code patterns illustrating the MSG produced for several large reduction vertices.

vertices shown in the chart). As would be expected the reduction from the edges is in general lower.

Of the 30 edges associated with at least a 10% reduction 23 have essentially the same reduction as their associated vertex. Five of these are the sole incoming edge of the target vertex. For the others, the target vertex includes edges that contribute little of the size of a slices. For example, the most common such edge is from a declaration vertex.

The remaining seven edges only cause part of the reduction obtained by ignoring all the edges into their target vertex. This pattern is rare because it requires incoming edges whose sources are dependent on separate parts of the code. Thus, each such edge includes a separate part of the program. For example, two such edges target `copia`'s switch statement `switch(a)` (recall that `a` holds the finite-state machine's next state). These two edges are a data-

dependence from the vertex representing *a* as a formal parameter, and a control-dependence from the function entry vertex. In this case, part of the computation of *a* is separate from other computations of the program. Thus, ignoring the data-dependence reduces all the slices that include the switch by the size of computation of *a*. This does not cause a breaking of *copia*'s cluster because most of the code impacting *a* is also reachable through the control dependence. This situation is not symmetric as ignoring the control edge disconnects very little because the vertex representing the formal parameter *a* is dependent on the function entry vertex. Consequently, there is transitive dependence *covering* the control dependence and thus maintaining a connection.

Finally, the MSGs for the 30 edges associated with at least a 10% reduction were constructed. These showed five breaks and five partial breaks. They also include ten shifts and ten drops (all subsets of the vertex data). While not many edges cause significant cluster breaking, it is important to realize that an edge is a rather small component and thus that any edge acts as a linchpin is interesting.

E. Threats to Validity

This section concludes by considering threats to the external, internal, and construct validity of the results presented. The main external threat arises from the possibility that the selected programs are not representative of programs in general, with the result that the findings of the experiments do not apply to 'typical' programs. The programs studied perform a wide variety of different tasks including, applications, utilities, games, and system code. There is, therefore, reasonable cause for confidence in the results obtained and the conclusions drawn from them. However, all of the programs studied were C/C++ programs. Therefore, it would be premature to infer that the results necessarily apply to other programming languages.

Before considering threats to internal validity, there is one important *non-threat* that is often mistaken as a threat to external validity of an experiment that uses the same or similar subjects. As in this study, the selected subject programs have been used in previous studies involving dependence clusters. However different questions are being asked and thus no statistical concern exists. Prior work with these programs has observed the existence of large dependence clusters and identified heuristics for finding them. This paper looks into the low-level causes of dependence clusters.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. In this experiment, the primary threat comes from the potential for faults in the tools used to gather the data. A mature and widely used slicing tool (CodeSurfer [13]) was used to mitigate this concern.

Finally, construct validity assesses the degree to which the variables used in the study accurately measure the concepts they purport to measure. Note that in the presence of human

judgments, construct validity is a more serious concern. In this study the only measurement is of slice size, which can be done accurately.

IV. RELATED WORK

This paper considers the role that individual vertices and edges play in holding together dependence clusters. There has been no previous quantitative assessment of their impact on dependence. Previous work on dependence clusters in software engineering [9], [21], [22] has focused on higher levels of abstraction, such as models or functions, while work on source-level dependence clusters has been primarily carried out in support of compiler analysis where semantics preservation is a key requirement [11], [18], [24].

The primary motivation for the study of dependence cluster causes derives from the impact that they have on testing and maintenance. In testing, dependence analysis has been shown to be effective at reducing the computational effort required to automate the test-data generation process [15]. However, the presence of large dependence clusters means that no such reduction can be achieved when testing any part of the program that lies in a cluster.

In software maintenance, dependence analysis is used to protect a software maintenance engineer against the potentially unforeseen side effects of a maintenance change. This can be achieved by measuring the impact of the proposed change [7] or by attempting to identify portions of code for which a change can be safely performed, free from side effects [12], [25]. Once again, as with testing, the presence of dependence clusters reduces the effectiveness of these maintenance support techniques.

Previous work has also considered other potential harmful effects of dependence structures that can be uncovered using static analysis. Chief among these are 'dependence anti-patterns' [3] and dependence clusters [4]. This work provided a definition of several forms of anti-patterns and dependence-based techniques for locating them. The results include the identification of multiple examples of anti-patterns in open source and production industrial code. Other work illustrated the way in which the normally static nature of these forms of dependence analysis could be brought to life in animations, that provide a 'fourth dimension' to dependence visualization [6].

Looking beyond dependence analysis to extra-dependence applications, the search for linchpins is akin to rarity measurements and anomaly detection used, for example, in social networking research. In graph theory terms, a social network is a directed graph composed of vertices that most often represent people and edges that represent relationships such as shared experience or common interests. An example is a graph of telephone calls across multiple customers. Two representative examples of the many such techniques are briefly considered.

In the first, Lin and Chalupsky used *rarity measurements* to discover unusual links within a graph [20]. This approach assumes that a graph is build from a pattern that repeats itself over and over. It looks for subgraphs that are *different*. This approach is local in nature (similar to tree-based pattern matching code generators). In contrast, a dependence graph vertex or edge being a linchpin is not a local property and thus pattern based matching is ineffective. However some patterns might find use to filter from consideration components that cannot play the linchpin role. This is a topic for future research.

In the second extra-dependence technique, Rattigan and Jensen seek to identify outliers in any data set that can be represented as a graph using a statistical approach to anomalous link detection [23]. They observe that “Relational learning techniques seem especially suited to the anomaly detection problem, because structured data lend themselves to a host of possible methods for finding interesting instances in a data set.” For example, in an authorship graph where vertices represent authors and papers are represented as links that connect coauthors, it is useful to know when an interesting collaboration exists. The technique used finds outliers based on the Katz measure. This measure is a weighted sum of the number of paths in the graph that connect two nodes, with shorter paths being given higher weight. A similar approach could be applied to dependence graphs with all paths being given equal weight. However, paths capture transitive dependence and thus overstate the connectedness in a dependence graph.

Finally, the identification of linchpin vertices and edges suggests the search for *language-level* features that may act as *high-level linchpins*. Examples include procedure calls and global variables. A search for global-variable linchpins in a collection of 21 programs found that 30 of the 849 global variables played the roll of a linchpin with just of over 50% of the programs including at least one such global [5]. Better understanding of the graph-level and source-level features the hold together large dependence clusters will allow programmers and static-analysis tools to better deal with the complications caused by large dependence clusters.

V. FUTURE WORK

Future work will consider the following: criteria that help separate refactorable clusters from unavoidable clusters, techniques for aiding the programmer break dependence clusters into smaller more manageable clusters, empirical assessment of their effects on programmer comprehension, other potential causes of dependence clusters, and more efficient detection techniques. Some of these are considered in more detail in this section.

To begin with, whether linchpins can be removed from code without affecting behaviour remains to be studied. Clearly some human guided intervention will be required. It may be considered more trouble than the perceived accrued

benefit in some cases. However, the knowledge of the existence and location of linchpins may be useful information in itself. Future work will explore the extent to which a tool can guide, support and, reduce human effort in refactoring code to remove the need for linchpins, thereby breaking up clusters.

Future work will also consider sub-cluster relationships. For example, where the slices of one cluster include the slices of another. Such relationships are an example of a feature not easily visualized using the MSG. For example, the lower chart in Figure 10 includes two clusters (labeled Generate Pattern and Match Pattern). From the MSG, it is not clear that Generate Pattern is related to Match Pattern when in fact it contains it.

Operationally, future work will try to develop more sophisticated techniques for identifying linchpins. The current approach employs a rather brute force search. Improvement are expected to come from borrowing techniques used in search-based software engineering and from exploiting patterns in the SDG. An example of such a pattern is shown in Figure 12 where the highlighted vertex labeled $y = x + 1$ *cannot* be a linchpin. This is because there are two paths connecting the vertex labeled $x = 1$ to the vertex labeled $z = x * y$. Ignoring the incoming dependences associated with the vertex labeled $y = x + 1$, does not disconnected these two vertices and thus the level of connectedness in the graph does not change. In this case, the slice taken with respect to any vertex except the highlighted one (*e.g.*, A or B) continues to include C (and the vertices C depends on); thus the slice size of all slices except the one taken with respect to $y = x + 1$ is unchanged.

VI. SUMMARY

This paper describes recent work on analyzing and understanding the causes of large dependence clusters. These surprisingly common features in programs, negatively impact programmer performance as well as static-analysis tool performance. Previous work has identified the prevalence of dependence clusters. This paper introduces a technique for measuring the amount of dependence channeled through a single vertex or edge, and then uses this technique to study the effects of almost 300,000 vertices and over 1.3 million edges. The results show that, while most vertices and edges have essentially no impact on program dependence clusters, there are a handful of the vertices and edges act as linchpins in that they hold together large dependence clusters. Moreover, though there may be few such linchpins, they were found spread over a quarter of the programs studied.

VII. ACKNOWLEDGEMENT

Discussions and working with Jens Krinke were essential in the running of the experiments and preparing of this paper. Chris Morrell for his statistical advice and Greg Livingston

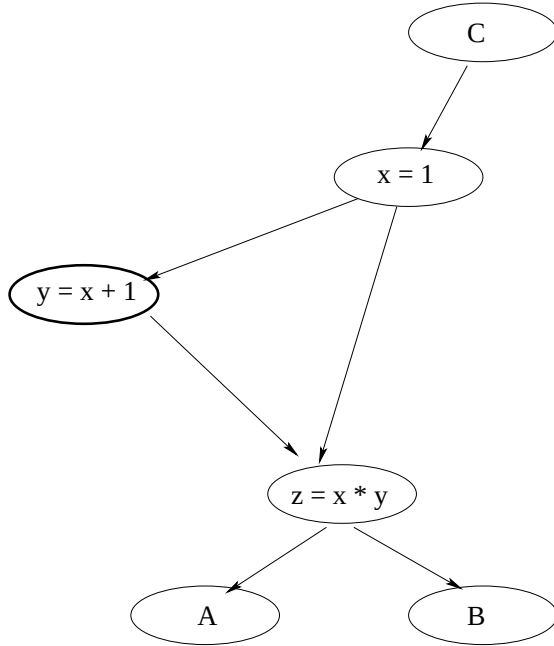


Figure 12. Simple Example of a vertex (highlighted in bold) that need not be considered as a potential linchpin.

for his comments on the prose. This work was supported by EPSRC grant EP/F01044.3.

REFERENCES

- [1] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In P. Fritzson, editor, *1st Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer. Also available as University of Wisconsin-Madison, technical report (in extended form), TR-1128, December, 1992.
- [2] F. Balmas. Using dependence graphs as a support to document programs. In *2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 145–154, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.
- [3] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, and J. Wegener. Dependence anti patterns. In *4th International ERCIM Workshop on Software Evolution and Evolvability (Evol’08)*, pages 25–34, L’Aquila, Italy, September 2008.
- [4] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *21st IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.
- [5] D. Binkley, M. Harman, Y. Hassoun, S. Islam, and Z. Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software*, 2009. To appear.
- [6] D. Binkley, M. Harman, and J. Krinke. Animated visualisation of static analysis: Characterising, explaining and exploiting the approximate nature of static analysis. In *6th International Workshop on Source Code Analysis and Manipulation (SCAM 06)*, pages 43–52, Philadelphia, Pennsylvania, USA, Sept. 2006.
- [7] S. E. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, 2001.
- [8] Y. Deng, S. Kothari, and Y. Namara. Program slice browser. In *9th IEEE International Workshop on Program Comprehension*, pages 50–59, Los Alamitos, California, USA, May 2001. IEEE Computer Society Press.
- [9] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3), 2003. Special issue on ICSM 2001.
- [10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [11] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.
- [12] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.
- [13] Grammatech Inc. The codesurfer slicing system, 2002.
- [14] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems*, 2009. to appear.
- [15] M. Harman, Y. Hassoun, K. Lakhota, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *ACM Symposium on the Foundations of Software Engineering (FSE ’07)*, pages 155–164, Dubrovnik, Croatia, September 2007. Association for Computer Machinery.
- [16] M. Harman, A. Lakhota, and D. W. Binkley. A framework for static slicers of unstructured programs. *Information and Software Technology*, 48(7):549–565, 2006.
- [17] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.
- [18] N. Jones and S. Muchnick, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [19] J. Krinke. Static slicing of threaded programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE’98)*, pages 35–42, June 1998.
- [20] S. Lin and H. Chlupsky. Unsupervised link discovery in multi-relational data via rarity analysis. In *IEEE ICDM Conference on Data Mining*, 2003.
- [21] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance*, pages 315–324, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.
- [22] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.
- [23] M. Rattigan and D. Jensen. The case for anomalous link discovery. *ACM SIGKDD Expl. News*, 7(2), 2005.
- [24] T. W. Reps. Solving demand versions of interprocedural analysis problems. In P. Fritzson, editor, *Compiler Construction, 5th International Conference*, volume 786 of *Lecture Notes in Computer Science*, pages 389–403, Edinburgh, U.K., 7–9 Apr. 1994. Springer.
- [25] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6), 2003.