Dependence Clusters Causes

David Binkley*
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. For small programs, programmers as well as static-analysis tools can overcome the negative effects of large dependence clusters. However, this ability diminished as program size increases. Thus, the existence of large dependence clusters presents a serious challenge to the scalability of modern software. Recent ongoing work into the existence and causes of dependence clusters is presented. A better understanding of clusters and their causes is a precursor to the construction of more informed analysis tools and ideally the eventual breaking or proactive avoidance of large dependence clusters.

1 Introduction

Within a program, dependence connects program components (*e.g.*, statements and predicates). Dependence has a bearing on many aspects of software engineering. For example, it has been linked to ease of program understanding [2, 5], used to delimit the changes that may be performed [7, 11], and to capture the impact of a change [4].

A *dependence cluster* is a maximal set of program components where each depends on the others. Dependence clusters have a negative impact on both programmer effectiveness and dependence-based static analysis tools. For example, consider a programmer trying to test or to debug part of the program. If this part overlaps with a dependence clutter, then the programmer must be aware of the entire cluster, which greatly reduces their productivity. As a static analysis example, consider a 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.

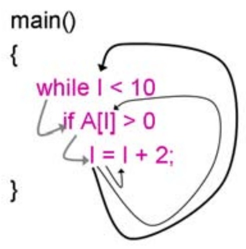
Two common dependences are the *data dependence* and *control dependence*. A *data dependence* connects a definition of a variable with each use of the variable reached by the definition [6]. A *control dependence* connects a predicate p to a component c when p has at least two control-flow-graph successors, one of which can lead to the exit node without encountering c and the other always leads eventually to c [1, 8]. For structured code, control dependence reflects the nesting structure of the program.

Figure 1 shows an example simple dependence cluster where the predicate $\mathsf{I} < 10$ data depends on the assignment to I, this assignment control depends on the predicate of the if statement, and the if control depends on the predicate $\mathsf{I} < 10$. Thus, the three statements are tied together into a cluster by their dependences. Recent empirical work has shown that dependence clusters are surprisingly prevalent [3].

On a small scale, programmers and tools can overcome the impact of programs that include large dependence clusters. However, this ability diminished as program size increases. Thus, a challenge in scaling the software development process is the dealing with large dependence clusters.

This paper considers the causes of large dependence clusters. It first summarizes prior work on the identification of dependence clusters. Then, the core of the paper, presented in Section 3, considers the initial search for dependence cluster causes. The paper then considers future work before concluding with a summary.

*On sabbatical leave from Loyola College in Maryland.



```
main()
{
   while I < 10
     if A[I] > 0
       I = I + 2;
}
```

Figure 1: A cluster caused by data (black) dependences and control (grey) dependences.

```
void f(int a)                 void g(int x)
{                             {
   a = 42                        x = 1968
   b = h(a)                      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 dependences from the assignment to variable a to the assignment of variable y that does not correspond to a legal execution of the program.

2 Dependence Cluster Identification

This section describes how dependence clusters are defined and then presents results from an empirical search for them. A naive definition of a dependence cluster would be based on transitive closure and thus would define a cluster as a strongly connected component. Unfortunately, for certain language features dependence is not transitive. Examples of such features include procedures [9] and threads [10]. 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 a sequence of data dependences 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 dependences 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, ignores calling-context. Therefore, forming strongly 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 calling-context information [9]. 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) [9].

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* [3]. 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.

This approximation also provides a useful visualization: the *Monotone Slice-size Graph* (MSG) [3]. 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, Figure 3 shows an example MSG where, reading along the horizontal axis, approximately the first 8%

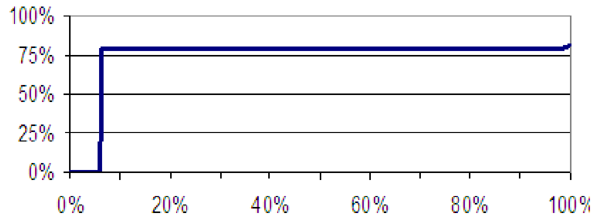


Figure 3: An example MSG showing a large dependence cluster.

of slices are very small, after which the MSG reveals a sharp increase to just over 50% of the program. Most of the remaining slices are essentially the same size thought there is a small up-tick at the far right.

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?).

Using 45 programs that contain just over one million lines of C code, a surprisingly large number of large clusters turned up [3]; although 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 range in size from 600 LoC to almost 180 KLoC and cover a range of application domains such as utilities, games, operating system code, etc.

While the definition of *large* is clearly subjective, Figure 4 attempts to quantify 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 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 of 94% of the program!

3 Dependence Cluster Causes

Having found that dependence clusters are common and noting that they have a negative impact on software construction, this paper considers the next ongoing phase in the investigation: the search for *dependence cluster causes*. 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 (or small set of components) that hold a dependence cluster together. If such components exist, then ignoring the dependences associated with them will cause the dependence cluster to disappear.

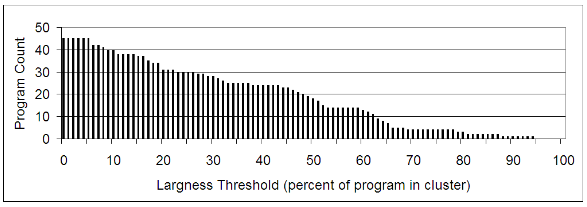


Figure 4: For *large* ranging from 0 to 100% of a program, the y-axis shows the number of the 45 programs with at least one large cluster.

While partially successful, an initial manual search proved to time consuming and too error prone; thus, a semi-automated search was employed. This search is based on 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 the cause of a cluster. 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 have 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 a breaking of a cluster, but does not imply the breaking.

Starting 'small', the search for *linchpin* components begins by considering individual vertices and edges from the program's *System Dependence Graph* (SDG) [9]. 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.

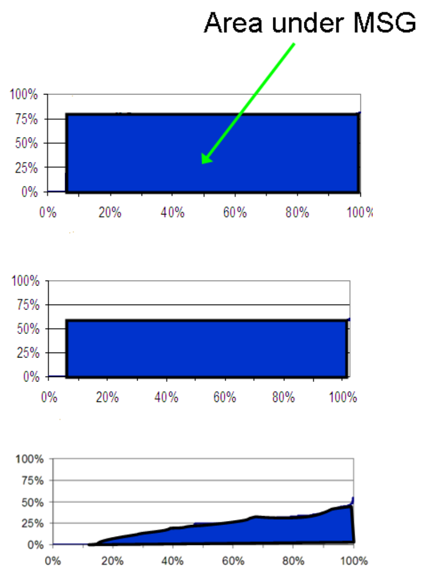


Figure 5: The area under the MSG drops under two conditions: the slices of the cluster get smaller (center MSG), or when the cluster breaks (lower MSG). Thus a reduction in area is a necessary, but not sufficient condition for a cluster breaking.

The search for dependence clusters is conducted by computing the MSG while *ignoring* the dependence associated with an individual component. Here, a single dependence edge represents the smallest components considered. If all of the dependence 'flows' through an 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. For a vertex, it is sufficient to ignore either the vertex's incoming *or* outgoing dependence edges. 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. By definition ignoring the dependence associated with a vertex ignores the dependence associated *all* its incoming edges. Thus, only those edges whose associated vertex has a significant impact, need be considered. Because recomputing the MSG while ignoring the dependence represented by an edge or a vertex is marginally expensive, all vertices are processed first and then only edges incident on a vertex associated with a significant reduction are considered. This avoids significant wasted computation by not considering unprofitable edges.

3.1 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 paths neck down to *one* 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 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 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 6. Each chart shows, in monotonically decreasing order, the reduction in area under the MSG obtained by ignoring the dependences of each program 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 6. Given that a vertex is a rather small part of an SDG, that such vertices exist is, in and of itself, interesting.

Considering all 279,992 vertices collectively, only 1611 (0.58%) cause a reduction greater than 1%. Those causing a 10% and a 20% reduction number 71 (0.025%) and 21 (0.0075%), respectively. The reduction caused by the 71 vertices having at least 10% reduction is shown in Figure 7. Finally, details of the 21 verities that cause a reduction of at least 20% are presented in Figure 8.

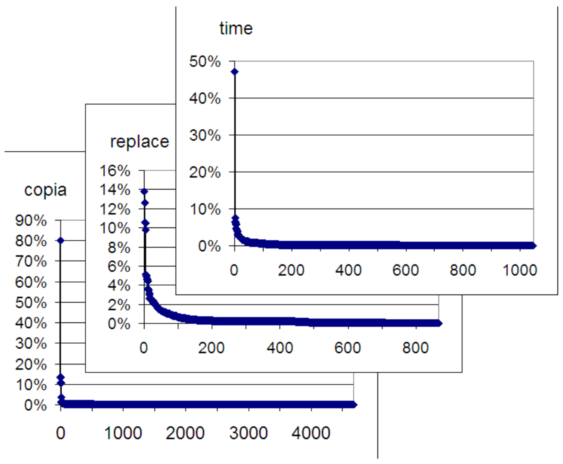


Figure 6: Fall-off in reduction for three example programs. Note that the scale on the y-axis is not the same in the three graphs.

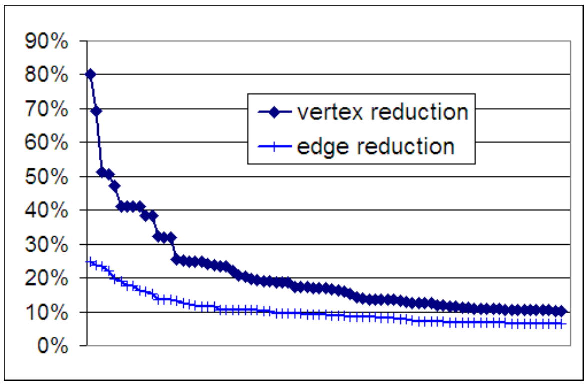


Figure 7: Top Reduction Causing Vertices and Edges

The most striking feature in this data is that the *linchpin vertices* causing the largest reductions are dominated by control-points. Only two non-control vertices (a function return value and the expression done++) occur in the top 21. Figure 8 also includes, a subset of the 50 vertices causing between 10% and 20% reduction (the selection favors the larger programs). In this range, it is clear that control has lost its dominance. This is true with the smaller programs as well. Thus 'control' statement appear to produce the largest reduction, but (almost) as large reduction is caused by non-control vertices.

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* in the MSG. Figure 9 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) is ignored. This vertex is in a loop that processes an output formal string for the current time. The loop and switch tie together 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 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 9 shows the MSGs for the top three reduction causing vertices of the program replace. While none of these provides significant evidence of cluster breaking, they are useful in illustrating two patterns seen in non-cluster busting 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 chart in Figure 10. Ignoring the vertex that represents the assignment of amatch's return value to variable m essentially *cuts off* slices that use the values 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 com-

Program	Percentage	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 the 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]=probtab[i].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 (i¡riga)
compress	10.73%	expression	text_buffer[bufindex]=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 8: Vertices causing a large reduction.

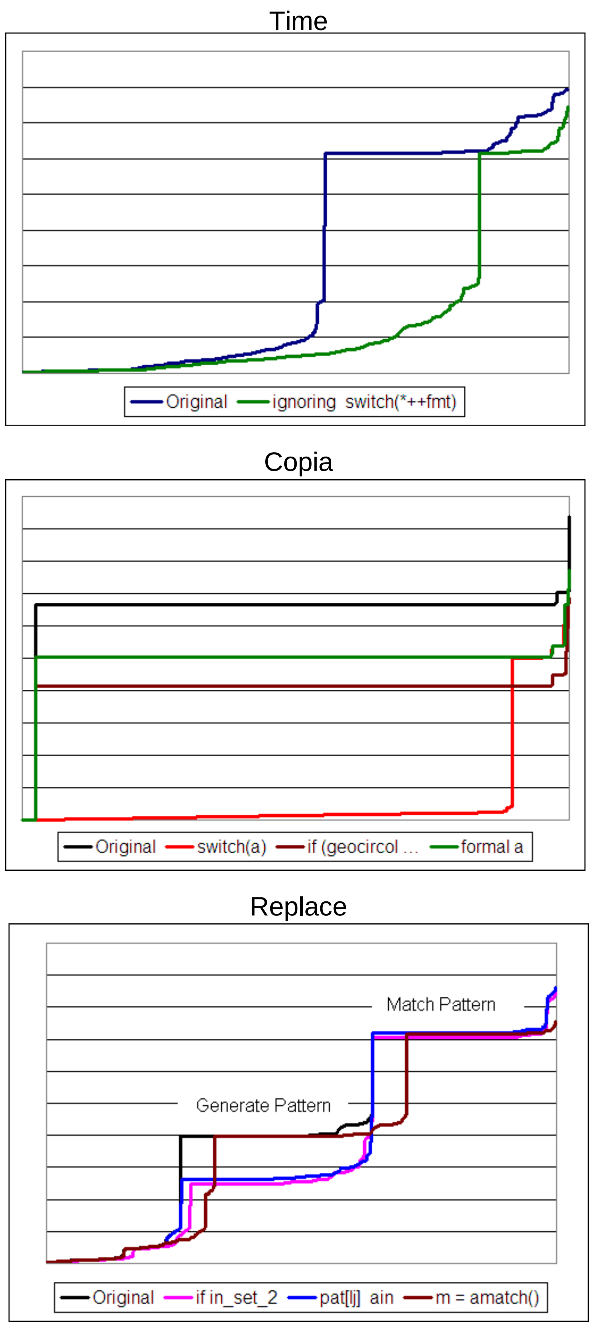
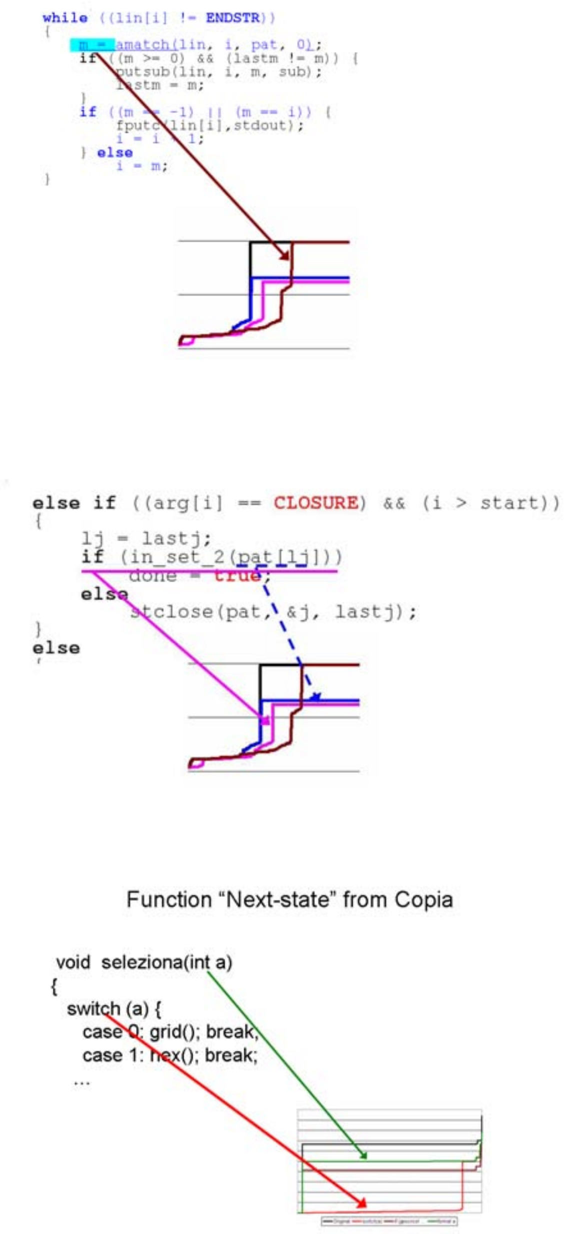


Figure 9: MSGs for high reduction vertices that show the different patterns seen in the search for dependence clusters.



```
while ((lin[i] != ENDSTR))
{
    m = amatch(lin, i, pat, 0);
    if ((m >= 0) && (lastm != m)) {
        putsub(lin, i, m, sub);
        lastm = m;
    }
    if ((m == -1) || (m == i)) {
        fputc(lin[i],stdout);
        i = i + 1;
    } else
        i = m;
}
```

```
else if ((arg[i] == CLOSURE) && (i > start))
{
    lj = lastj;
    if (in_set_2(pat[lj]))
        done = true;
    else
        stclose(pat, &j, lastj);
}
else
```

Function “Next-state” from Copia

```
void  seleziona(int a)
{
  switch (a) {
    case 0: grid(); break;
    case 1: hex(); break;
  ...
```

Figure 10: Source code patterns illustrating the MSG produced for several large reduction vertices on the MSGs.

putation 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 chart in the middle of Figure 10 shows two patterns: a *drop* and the combination of a shift and a drop. The drop comes from ignoring the actual parameter that represents the passing of actual parameter 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 chart 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 chart in Figure 10 includes the MSG shown at the center chart of Figure 9. 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 selezioa. 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 that are 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 8.

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 vertices. The remaining vertices produce 23 drops and 22 shifts. In identifying drops and shifts the dominant feature was used, so no combinations of drop and shift are not reported.

3.2 Linchpin Edges

The search of linchpin edges considers the 8678 incoming edges of the 1623 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 almost nine million. All the unconsidered edges, by definition, produce no more than a 1% reduction. Of the 8678 edges 5106 (59%) led to *no* reduction at all, 2682 (31%) led to between 0% and 1% reduction, 860 (10%) led to between 1% and 10% reduction, and 30 (0.34%) let to a reduction greater then 10% reduction. Figure 7 shows the reduction attained by the top 78 edges (the number of edges was chosen to match the number of 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 variable 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 of the seven edges targets copia's switch statement switch(a) (recall that a hold the finite-state machine's next state). These two edges are a data-dependence from the vertex representing a as a formal parameter to the function, and a control-dependence from the function entry vertex. In this case, part of the computation of a is separate from other parts of the program. Thus, ignoring the data-dependence reduces all the slices including 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.

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 busting, it is important to note that an edge is a rather small component and thus that any an edge has a significant impact is interesting.

3.3 Mutually Recursive Functions

The frequency of control vertices, procedure calls, and parameter vertices in Figure 8, and the importance that mutually recursive functions have in the copia example, suggest that mutually recursive functions might play a more general role as a linchpin component that holds a dependence cluster together. The search for mutually recursive functions is conducted by separately ignoring dependences associated with calls and procedure entry. Both of which include ignoring dependences of the associated parameter vertices.

To better understand the relation between these two, let *max_call* be the maximum reduction attained by ignoring the dependence associated with a single call and let *max_entry* be the maximum reduction attained by ignoring the dependence associated with a single procedure entry. For programs with a single *key* call to a *key* procedure, *max_call* and *max_entry* are essentially the same. Multiple calls to a key procedure dilute the reduction attained by ignoring the incoming dependence of any one call; thus, *max_entry* is larger. However, it is possible for *max_call* to exceed *max_entry*. The most common case where this happens involves an indirect call cite.

For most of the programs the reduction obtained is similar weather ignoring calls or entrys. For a few cases, one or the other produces a considerably larger reduction. The remainder of this section first considers programs for which ignoring the incoming dependences of a call produces a larger reduction than ignoring the incoming dependences of a procedure entry. It then considers the inverse. Fortuitously, these two illustrate the two main causes of *max_call* exceeding *max_entry*.

To begin with, in the program findutils ignoring dependence associated with an indirect call site causes the largest reduction. The function-pointer used can point to one of sixty different parse functions that each process the arguments for a particular search criteria. Thus, ignoring dependences associated with the call (*parse_function)(), has a significant impact on the area under the MSG. The MSGs with and with-out the dependences associated with this call are shown in Figure 11. The original (upper) MSG shows that findutils includes four clusters (the first two have similar slice size). Ignoring dependence associated with the indirect call (*parse_function)() breaks the first cluster (making it easier to tell apart from the second). It also reduces the size of the slices in the remaining three clusters (causing a drop in the latter part of the MSG).

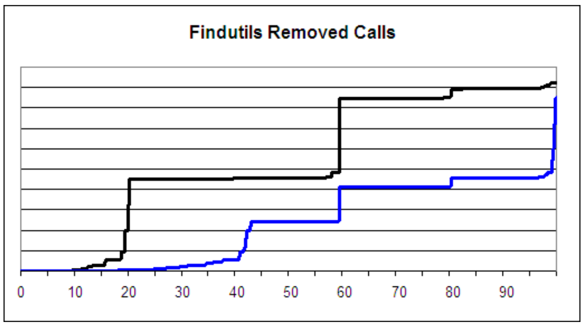


Figure 11: The MSG for findutils before and after the ignoring the dependences associated with an indirect call.

The second example, which illustrates the other main causes of *max_call* exceeding *max_entry*, occurs in the program sudoku where *max_call* and *max_entry* both involve the function submcom(). In this case *max_call* exceeds *max_entry* because of what might be termed the *summary edge effect*. Figure 12 illustrates this effect caused when the ignoring of the dependences associated with a call-site resulting in the ignoring of the associated SDG summary edges. First, consider the graph fragment shown in the left of Figure 12. In the SDG, parameter passing and return values are modeled as assignments through special variables labeled f_{in} and P_{out} for formal f and function P, respectively. (The graph fragment shown is a simplification of the actual SDG.) In this graph there is a path (an intraprocedural path) from source S to target T; thus, a slice that includes T will also include S.

The impact of ignoring the dependences associated with a call and procedure entry are shown in the center and right of Figure 12 where ignored dependences are shown in gray. The key difference is that the ignoring of dependences (incoming edges) of the vertex labeled “a = P_{out}” breaks the connection between S and T. Thus, unlike the right figure, in the center figure there is no path from S to T. Because of this, ignoring the dependences associated with a call can cause greater reduction than ignoring the edges associated with the entry to the called procedure.

The proceeding two examples illustrated the two main cases where *max_call* exceeds *max_entry*. At the other end of the spectrum, two illustrative examples of when *max_entry* exceeds *max_call* are found in the programs bc and gnuchess. With bc there are two calls to the function, dc_func, which processes the current input character. This character can denote part of number, an operation (*e.g.*, add, multiple, negate), etc. Each call is in a loop (that iterates over a string or a file, respectively). Together the call and the loop ‘connect’ the implementation of the various operations. Ignoring the dependences associated with one of the two calls still leaves the operations connected through the other. Ignoring the dependences associated with the entry, breaks these dependences. This effectively breaks the large dependence cluster in bc, as shown in Figure 13.

The second example is the program gnuchess. Ignoring dependences associated with the function SelectMove() generates a 44% reduction. This is the lowest line in the chart shown in Figure 14. In addition to the lowering the MSG, the chart shows evidence of several smaller clusters separating out. The middle MSG results from ignoring of dependence associated with the function search(). While it still shows the large central cluster, the beginning of this MSG shows the gradual increase that is evidence of a broken cluster. In both cases the associated call-sites cause no significant reduction.

Because more dependences (*i.e.*, those of a function entry or call, *and* its parameters) are being ignored, greater reduction is to be expected. This expectation is born out in the data generated from the 4275 calls and 2759 entry’s that were considered. Overall 3920 (68.3%) of the 4275 call cites and 2474 (89.6%) of the 2759 entrys caused less than a 10% reduction. Looking at the top 10% of each data set, the 42 calls include six that come from programs without clusters and thus produce drops by definition. From the programs that include clusters, there are nine breaks, 13 partial breaks, but only 13 drops and one shift. Thus, a higher percentage of breaks are seen than with the smaller linchpin vertices or edges.

Next considering the entry’s, 27 entry’s include three from programs without clusters and thus are drops by definition. From the programs that include clusters, there are eight breaks, 13 partial breaks and only 3 drops with no shifts. Thus, even more dramatic than the calls, a higher percentage of breaks are seen than with the smaller linchpins.

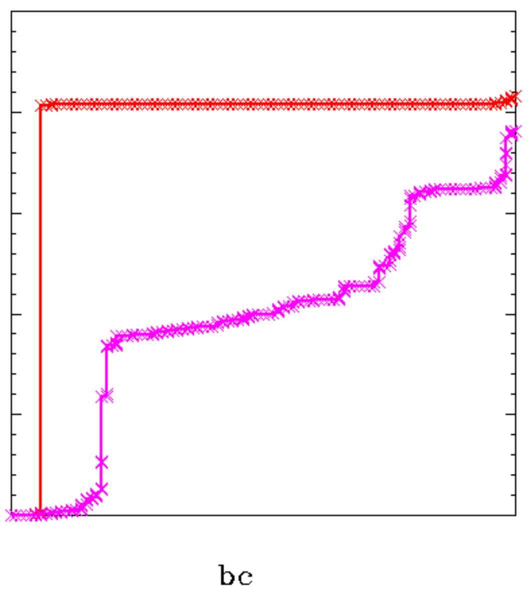


Figure 13: Reduction caused by ignoring dependences of the entry to the functions dc_func from bc.

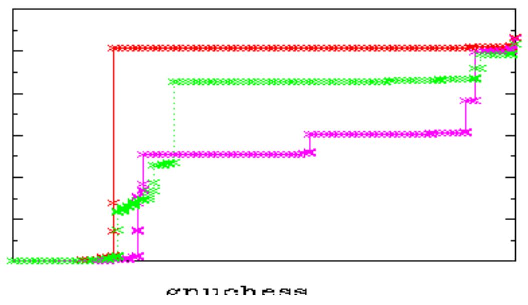


Figure 14: Ignoring dependence associated with the top 2 function entries in gnuchess.

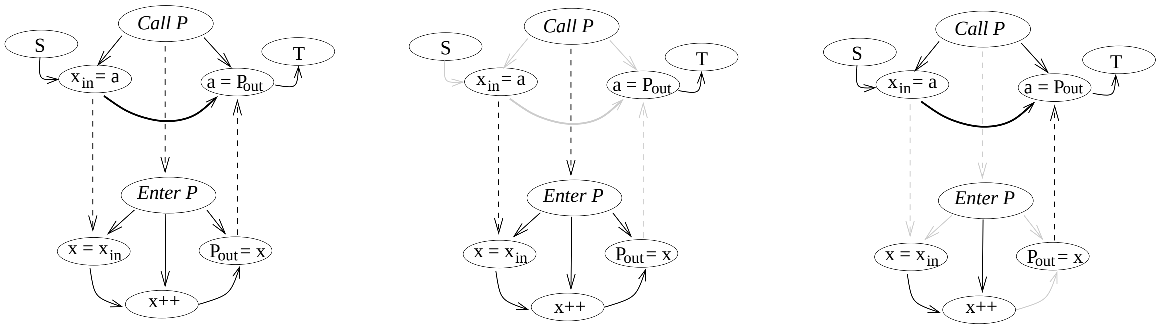


Figure 12: Illustration of the Summary-Edge Effect

3.4 Difference between source level and graph level analysis

While ignoring the dependences of an individual vertex has a clear parallel at the source level, the summary edge effect suggests that ignoring dependences of larger SDG structures might not always have a direct analog at the source level. This section illustrates how ignoring the incoming dependences of a call in the SDG differs from ignoring the incoming dependences of the same call in the source.

For example, the transitive effects of the call's summary edges cannot be (easily) accounted for from within the SDG (consider the case where Q calls P producing a summary edge at the call-site, and P calls R again producing a summary edge at the call-site). If the call (or entry) to R is ignored this might disconnect the path causing the summary edge at the call to P, but it might not. Re-computation is possible, but costly.

This suggests transformation at the source level. For example, replacing

```
int R(int x, long l)
{
   body
}
```

with

```
// stub generating no dependence
int R(int x, long l)
{
}
```

```
// body preserves dependence caused
// by calls in R
int R_PRIME(int x, long l)
{
  body
}
```

This rewriting effectively breaks the dependences generated through calls to R while preserving the dependences cause by calls from within the body of R (*i.e.*, from within body). A generalization of this observation suggests the study of other source-level transformations such as ignoring the dependence associated with globals variables.

4 Future Work

One of the features that is not easily visualized using the MSG is the sub-cluster relationships between clusters where the slices of one clusters includes the slices of another. For example, the lower chart in Figure 9 include two clusters (labeled Generate Pattern and Match Pattern). From the MSG, it is not clear that Generate Pattern is related to Match Pattern (beyond involving smaller slices). Figure 15 shows a sub-cluster containment graph for Replace. In the lower zoomed-in view, the two clusters (each of size 220) are labeled Generate Pattern and Match Pattern. From the sub-cluster diagram, it is clear that Generate Pattern is a sub-cluster of Match Pattern. Future challenges in presenting these sub-clusters are illustrated in Figure 16, where the number of sub-clusters and sub-cluster relations is visually overwhelming.

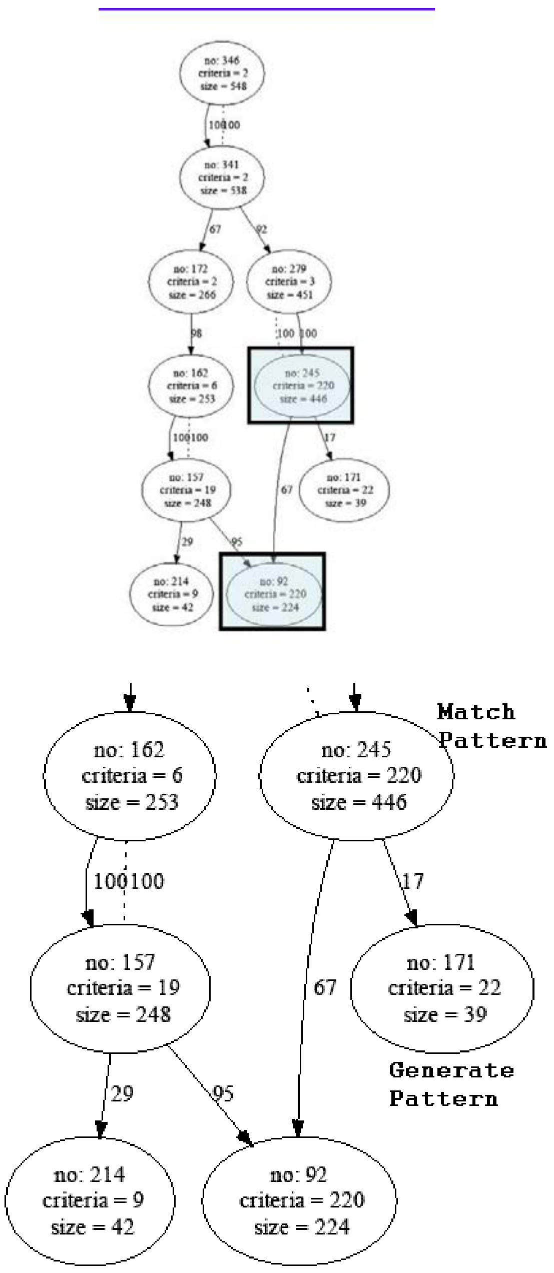

no: 346 criteria = 2 size = 548
100100
no: 341 criteria = 2 size = 538
67
92
no: 172 criteria = 2 size = 266
no: 279 criteria = 3 size = 451
98
100 100
no: 162 criteria = 6 size = 253
no: 245 criteria = 220 size = 446
100100
17
no: 157 criteria = 19 size = 248
67
no: 171 criteria = 22 size = 39
29
95
no: 214 criteria = 9 size = 42
no: 92 criteria = 220 size = 224
Match Pattern
Generate Pattern

Figure 15: The top figure shows all the sub-cluster relations withing the program Replace. The bottom figure shows a zoom of the lower portion.

5 Summary

This paper describes recent work on analyzing and understanding dependence clusters. These surprisingly common features in programs, negatively impact programmer performance and static-analysis tool performance. Previous work has identified the prevalence of dependence clusters. This paper presents results from an initial search for their causes. In starts by looking for 'low level' causes, whose presence is interesting in and of itself. The study finds that a handful of the vertices and edges are *linchpins* in that they hold together large dependence clusters. These vertices and edges suggest the search for *language-level* features that may act as *high-level linchpins*. Examples include procedure calls and global variables. Better understand of the graph-level and source-code 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.

6 Acknowledgement

Discussions and working with Mark Harman and Jens Krinke were essential in the running of the experiments and preparing of this paper. 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, 1^{st} *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 2^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 145–154, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

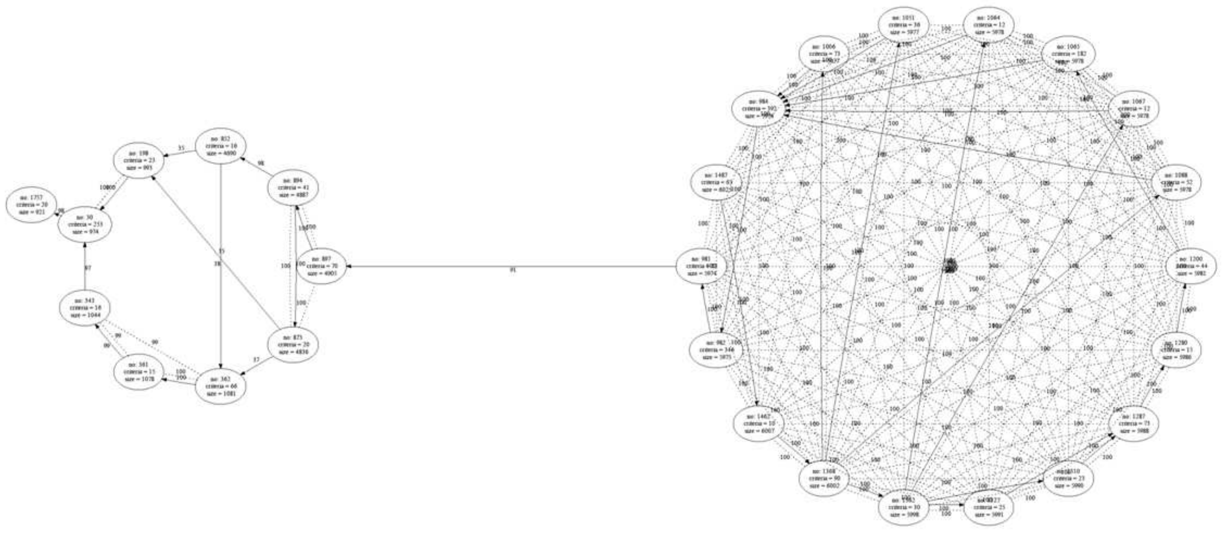


Figure 16: Sub-cluster Relations

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

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

[5] Y. Deng, S. Kothari, and Y. Namara. Program slice browser. In 9^{th} *IEEE International Workshop on Program Comprenhesion*, pages 50–59, Los Alamitos, California, USA, May 2001. IEEE Computer Society Press.

[6] 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.

[7] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[8] M. Harman, A. Lakhotia, and D. W. Binkley. A framework for static slicers of unstructured programs. *Information and Software Technology*, 48(7):549–565, 2006.

[9] 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.

[10] 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.

[11] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, 2003.