Efficient Identification of Linchpin Vertices in Dependence Clusters

David Binkley, Loyola University Maryland
Nicolas Gold, University College London
Mark Harman, University College London
Syed Islam, University College London
Jens Krinke, University College London
Zheng Li, Beijing University of Chemical Technology

Several authors have found evidence of large dependence clusters in the source code of a diverse range of systems, domains, and programming languages. This raises the question of how we might efficiently locate the fragments of code that give rise to large dependence clusters. We introduce an algorithm for the identification of *linchpin* vertices, which hold together large dependence clusters, and prove correctness properties for the algorithm's primary innovations. We also report the results of an empirical study concerning the reduction in analysis time that our algorithm yields over its predecessor using a collection of 38 programs containing almost half a million lines of code. Our empirical findings indicate improvements of almost two orders of magnitude, making it possible to process larger programs for which it would have previously been impractical.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments; E.1 [**Data Structures**]: Graphs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Slicing, Internal Representation, Performance Enhancement, Empirical Study

1. INTRODUCTION

A dependence cluster is a maximal set of program elements where each element depends on the others. Previous work has shown the widespread presence of very large dependence clusters in source code in a wide range of open source and commercial software [Harman et al. 2009]. The empirical observation of the prevalence of large dependence clusters, coupled with the belief that they are potential sources of problems [Binkley et al. 2008], motivates investigation into their causes.

Initial work on the causes of dependence clusters revealed evidence that global variables can be the source of so-called capillary data flows [Binkley et al. 2009], which lead to the formation of large dependence clusters. Global scope makes it hard to understand the effect of these variables in isolation and therefore difficult to take any action to ameliorate their potentially harmful effects without major restructuring. This motivated the study of *linchpin* edges and vertices [Binkley and Harman 2009]. A linchpin is a single edge or vertex in a program's dependence graph through which so much dependence flows that the linchpin holds together a large cluster.

One obvious and natural way to identify a linchpin is to remove it, re-construct the dependence graph, and then compare the 'before' and 'after' graphs to see if the large dependence cluster has either disappeared or reduced in size. This naïve approach was implemented as a proof of concept to demonstrate that such linchpins do indeed exist [Binkley and Harman 2009]. That is, there *are* single vertices and edges in real world systems, the removal of which causes large dependence clusters to essentially disappear.

This naïve algorithm is useful as a demonstration that linchpins exist, but it must consider all vertices as potential linchpins. Unfortunately, this limits its applicability as a useful research tool. That is, the computational resources required for even mid-sized programs are simply too great for the approach to be practical. In this paper we improve the applicability of the analysis from thousands of lines of code to tens of thousands of lines of code by developing a graph-pattern based theory that provides a foundation for more efficient linchpin detection. Finally, we introduce a new linchpin detection algorithm based on this theory and report on its performance with an empirical study of 38 programs containing a total of 494K lines of code.

The theory includes a ratio which we term the 'risk ratio'. If the risk ratio is sufficiently small then we know that the impact (as captured by the ratio) of a set of vertices on large dependence clusters will be negligible. On this basis we are able to define a predicate that guards whether or not we are able to prune vertices from linchpin consideration. The theory establishes the required properties of the risk ratio, but only an empirical study can answer whether or not the guarding predicate that uses this ratio is satisfied sufficiently often to be useful for performance improvement. We therefore complement the theoretical study with an empirical study that investigates this question. We find that the predicate is satisfied in all but four of over a million cases. Furthermore, these four all occur with the strictest configuration in very small programs. This provides empirical evidence to support the claim that the theory is highly applicable in practice. Our empirical study also reports the performance improvement obtained by the new algorithm. Finally, we introduce and empirically study a tuning parameter (the search depth in what we call 'fringe look ahead'). Our empirical study investigates the additional performance increase obtained using various values of the tuning parameter.

Using the new algorithm we were able to study several mid-sized programs ranging up to 66KLoC. This provides a relatively robust set of results on non-trivial systems upon which we draw evidence to support our empirical findings regarding the improved execution efficiency of the new algorithm. The results support our claim that the theoretical improvement of our algorithm is borne out in practice. For instance, to analyze the mid-sized program go, which has 29,246 lines of code, using the naïve approach takes 101 days. Using the tuned version of the new algorithm this time is reduced to just 8 days.

The primary contributions of the paper are as follows:

(1) Three theorems, proved in Section 3, identify situations in which it is possible to effectively exclude vertices from consideration as linchpins. These theoretical findings highlight opportunities for pruning the search for linchpins.
(2) Based on this theory, Section 4 introduces a more efficient linchpin search algorithm that exploits the pruning opportunities to reduce search time. Section 4 also proves the algorithm's correctness with respect to the theory introduced in Section 3.
(3) To empirically investigate the improvement achieved in practice using our new algorithm, Section 5 presents the results of an empirical study using a collection of 38 programs. The results from this study reveal that the basic algorithm can be used, with no tuning at all, to achieve at least an order of magnitude speedup in execution time. This means that offline linchpin identification becomes feasible where it was previously infeasible. We also present results that analyse

```
vertex_set reaching_vertices($G$, $V$, excluded_edge_kinds)
{
 work_list = $V$
 answer = $\emptyset$
 while work_list $!= \emptyset$
   select and remove a vertex $v$ from work_list
   mark $v$
   insert $v$ into answer
   foreach unmarked vertex $w$ such that there is an edge $w \rightarrow v$ whose kind is
       not in excluded_edge_kinds
     insert $w$ into work_list
   return answer
}

For SDG $G$,
  $b_1(v)$ = reaching_vertices($G$, $\{v\}$, {parameter-out})
  $b_2(v)$ = reaching_vertices($G$, $\{v\}$, {parameter-in, call})
```

Fig. 1. The function *reaching_vertices* [Binkley 1993] returns all vertices in SDG G from which there is a path to a vertex in V along edges whose edge-kind is something other than those in the set *excluded_edge_kinds*.

the further performance improvements that can be obtained by tuning the basic algorithm.

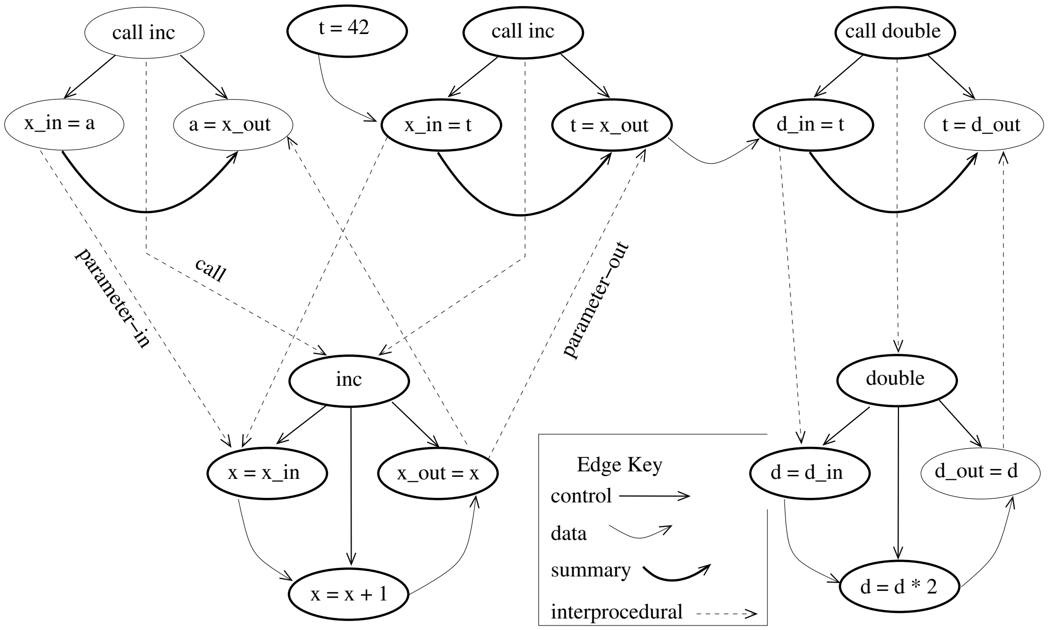
2. BACKGROUND: LARGE DEPENDENCE CLUSTERS AND THEIR CAUSES

Dependence clusters can be defined in terms of program slices [Binkley and Harman 2005]. This section briefly reviews the definitions of slice, dependence clusters, and dependence cluster causes, to motivate the study of dependence clusters in general and of improved techniques for finding linchpins in particular. More detailed accounts can be found in the literature [Harman et al. 2009].

A backward slice identifies the parts of a program that potentially affect a selected computation [Weiser 1984], while a forward slice identifies the parts of the program potentially affected by a selected computation [Horwitz et al. 1990; Reps and Yang 1988]. Both slices can be defined as the solution to graph reachability problems over a program's *System Dependence Graph* (SDG) [Horwitz et al. 1990] using two passes over the graph. The passes differ in their treatment of interprocedural edges. For example, the backward slice taken with respect to SDG vertex v, denoted $b(v)$, is computed by the first traversing only edges "up" into calling procedures while the second pass traverses only edges "down" into called procedures. Both passes exploit transitive dependence edges (summary edges) included at each call site to summarize the dependence paths through the called procedure. The two passes of a backward slice are referred to as b_1 and b_2; thus $b(v) = b_2(b_1(v))$ where the slice taken with respect to a set of vertices V is defined as the union of the slices taken with respect to each vertex $v \in V$. For a forward slice $f(v) = f_2(f_1(v))$ where f_1 traverses only edges "up" into calling procedures and f_2 traverses only edges "down" into called procedures.

Formalizing these slicing operators is done in terms of the interprocedural edges that enter and exit called procedures. Three such edge kinds exist in an SDG: a *parameter-in edge* represents the data dependence of a formal parameter on the value of the actual, a *parameter-out edge* represents the data dependence of an actual parameter on the final value of the formal as well as the data dependence from the returned value, and finally a *call edge* represents the control dependence of the called procedure on a call-site. Figure 1 provides the algorithm used to compute $b_1(v)$ and $b_2(v)$. The algorithm for forward slicing is the same except edges are traversed in the forward direction.

Example. Figure 2 shows a simplified SDG used to illustrate the key concepts in interprocedural slicing. Let vertex v be the vertex from procedure double labeled d = d * 2. The first backward-slicing pass, $b_1(v)$, ignores parameter-out edges, but traverses parameter-in and call edges to include vertices from procedure double and the rightmost two calls. Notice that the summary edges, which



```
{                              inc(x)                        double(d)
  a = inc(a)                   {                             {
  t = 42                         x = x + 1                     d = d * 2
  t = inc(t)                     return x                      return d
  t = double(t)                }                             }
}
```

Fig. 2. An example SDG showing with slice taken with respect to the vertex labeled d = d * 2. The vertices of the slice are shown bold.

summarize paths through the called procedure, allow Pass 1 to include the initialization of t (t = 42) without descending into procedure inc. The second pass, which excludes parameter-in and call edges, starts from all the vertices encountered in the first pass. In particular, when starting from the vertex labeled x = x_out the slice descends into procedure inc and thus includes the body of the procedure. Combined, the two passes respect calling context and thus correctly omit the first call on procedure inc. The vertices of the slice are shown in bold.

It is possible to compute the slice with respect to any SDG vertex. However, in the experiments only the vertices representing source code are considered as slice starting points. Furthermore, slice size is defined as the number of vertices representing source code encountered while slicing. Restricting attention to the vertices representing source code excludes several kinds of 'internal' vertices introduced by CodeSurfer [Grammatech Inc. 2002] (the tool used to build the SDGs). For example, an SDG includes pseudo-parameter vertices representing global variables potentially accessed by a called procedure.

While alternate definitions are possible, dependence clusters can be defined as maximal sets of SDG vertices that all have the same backward slice. That is, two vertices that have the same backward slice are deemed to reside in the same cluster. In practice, it turns out that *same backward slice* can be very closely approximated by *same backward slice size* [Binkley and Harman 2005]. This is a conservative approximation because two backward slices may differ, yet, coincidentally have the same size. However, two identical backward slices must have the same size. The 'same size' approximation has been empirically demonstrated to be over 99% accurate [Binkley and Harman 2005; Harman et al. 2009]. In the following definitions $|\cdots|$ is used to denote size. For a set S, $|S|$ denotes the number of elements in S, while for an SDG G, a slice $b(v)$, or an SDG path P, size is the number of vertices that represent source code in G, $b(v)$, or along P, respectively, Using slice size, dependence clusters can be defined as follows

Definition 1 (DEPENDENCE CLUSTER).
The dependence cluster *for vertex v of SDG G consists of all vertices that have the same slice size as v: $cluster(v) = \{u \in G\ s.t.\ |b(u)| = |b(v)|\}$.* □

Our previous work has demonstrated that large dependence clusters are surprisingly prevalent in traditional systems written in the C programming language, for both open and closed source systems [Harman et al. 2009; Binkley et al. 2008]. Other authors have subsequently replicated this finding in other languages and systems, both in open source and proprietary code [Beszédes et al. 2007; Szegedi et al. 2007; Acharya and Robinson 2011]. Though our work has focused on C programs, large dependence clusters have also been found by other authors in C++ and Java systems [Beszédes et al. 2007; Savernik 2007; Szegedi et al. 2007] and there is recent evidence that they are present in legacy Cobol systems [Hajnal and Forgács 2011].

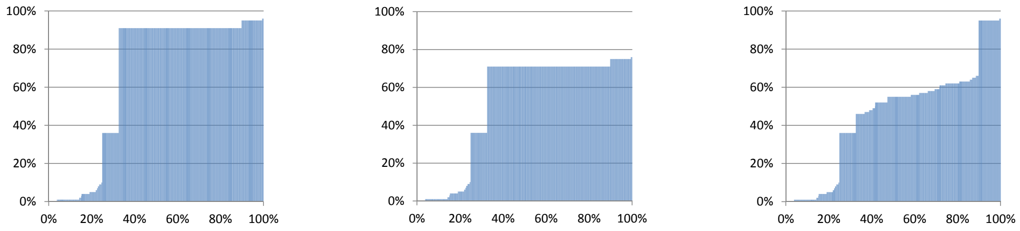
Large dependence clusters have been linked to dependence 'anti patterns' or bad smells that reflect possible problems for on-going software maintenance and evolution [Savernik 2007; Binkley et al. 2008; Acharya and Robinson 2011]. Other authors have studied the relationship between faults, program size, and dependence clusters [Black et al. 2006], and between impact analysis and dependence clusters [Acharya and Robinson 2011; Harman et al. 2009]. The presence of large dependence clusters has also been suggested as an opportunity for refactoring intervention [Black et al. 2009; Binkley and Harman 2005; Islam et al. 2010a].

Because dependence clusters are believed to raise potential problems for software maintenance, testing, and comprehension, and because they have been shown to be highly prevalent in real systems, a natural question arises: "What causes large dependence clusters?" Our previous work investigated the global variables that contribute to creating large clusters of dependence [Binkley et al. 2009]. For example, the global variable representing the board in a chess program creates a large cluster involving all the pieces. Finding such a global variable can be important for understanding the causes of a cluster. However, global variables, by their nature, permeate the entire program scope and so the ability to take action based on this knowledge is limited. This motivates the study of linchpins: small localized pieces of code that cause (in the sense that they hold the cluster together) the formation of large dependence clusters.

The search for linchpins considers the impact of removing each potential linchpin on the dependence connections in the program. In an SDG the component whose removal has the smallest dependence impact is a single dependence edge. A vertex, which can have multiple incident edges, is the next smallest component. Because a linchpin edge's target vertex must be a linchpin vertex, it is a quick process to identify linchpin edges once the linchpin vertices have been identified. This process simply considers each incoming edge of each linchpin vertex in turn [Binkley and Harman 2009]. Thus, in this paper, we characterize the answer to the question of what holds clusters together in terms of a search for *linchpin vertices.*

Ignoring the dependencies of a linchpin vertex will cause the dependence cluster to disappear. 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.

The search is largely automated by considering changes in the *Monotone Slice-size Graph* (MSG) [Binkley and Harman 2005]. An MSG is a graph of backward slice sizes plotted in monotonically increasing order on the x-axis. The y-axis measures backward slice size. That is, the backward 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, the MSGs shown in this



MSG (a) – original

MSG (b) – drop only

MSG (c) – broken cluster

Fig. 3. The area under the MSG drops under two conditions: the slices of the cluster get smaller (center MSG), or when the cluster breaks (rightmost MSG). Thus, while a reduction in area is necessary, it is not a sufficient condition for cluster breaking.

paper use the percentage of the backward slices taken on the x-axis and the percentage of the entire program on the y-axis.

In general the definitions laid out in the next section will work with an MSG constructed from any set of vertices. As mentioned above, for the empirical investigation presented in Section 5 the set of source-code representing vertices is used as both the slice starting points and when determining the size of a slice. Under this arrangement a cluster appears as a rectangle that is taller than it is wide.

The search considers changes in the *area under the MSG*, denoted A_{MSG}. This area is the sum of all the slice sizes that make up the MSG. Formally, if SC is the set of SDG vertices representing source code then

$$A_{MSG} = \sum_{v \,\in\, SC} |b(v)|$$

As illustrated in Figure 3, a *reduction* in area is a necessary but not a sufficient condition for identifying a *linchpin* vertex. This is because there are two possible outcomes: a *drop* and a *break*. These two are illustrated by the center and right-most MSGs shown in Figure 3. Both show a reduction in area; however, the center MSG reflects only a reduction in the size of the backward slices that makeup up a cluster. Only the right-most MSG shows a true breaking of the cluster. These two are clear-cut extreme examples meant to illustrate the concepts of a drop and a break. In reality there are reductions that incorporate both effects. In the end, the decision if a reduction represents a drop or a break is subjective.

The detection algorithm presented in Section 4 reports all cases in which the reduction is greater than a threshold. These must then be inspected to determine if the area reduction represents a true breaking of a cluster. From the three example MSGs shown in Figure 3, it is clear that a reduction in area must accompany the breaking of a cluster, but does not imply the breaking of a cluster. Thus, to test if a Vertex ℓ is a linchpin, the MSG for the program is constructed while ignoring ℓ's incoming dependence edges. If a significant reduction in area occurs, the resulting MSG can then be inspected to see if the cluster is broken.

The search for linchpins is thus conducted by computing the MSG while *ignoring* the dependence associated with a specific vertex. Previous work [Binkley and Harman 2009] has shown that ignoring dependence associated with vertices could identify linchpin vertices in real programs. However, this implementation is rather naïve.

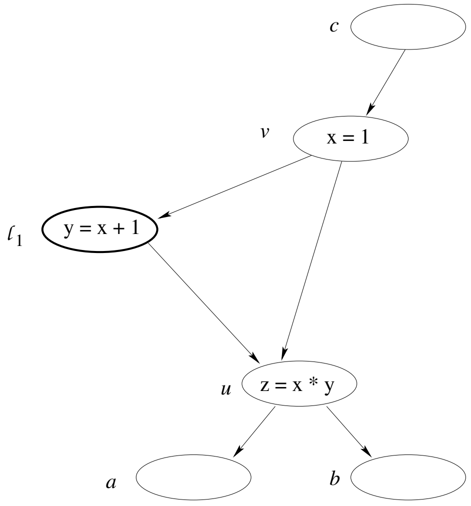


Fig. 4. Simple Example of a vertex (highlighted in bold) that need not be considered as a potential linchpin.

3. THEORETICAL INVESTIGATION OF PROPERTIES OF DEPENDENCE CLUSTER CAUSES

The naïve linchpin search algorithm simply recomputes the MSG while ignoring the incoming dependence edges of each vertex in turn. MSG construction is computationally non-trivial and the inspection becomes tedious when considering programs with thousands of vertices. Thus, this section considers the efficient search for linchpin vertices. The search centers around several graph patterns that identify vertices that *cannot* play the role of a linchpin vertex. The section begins with two examples that illustrate the key concepts.

Figure 4 shows a Vertex ℓ_1 that *cannot* be a linchpin. This is because there are two paths connecting Vertex v, labeled x = 1, to Vertex u, labeled z = x * y. Ignoring ℓ_1's incoming dependence edges does not disconnect v and u and thus the level of 'overall connectedness' does not diminish. Consequently, a backward slice taken with respect to any vertex other than ℓ_1 (*e.g.*, a or b) continues to include v and the vertices v depends on such as c; thus, the backward slice size of all slices except the one taken with respect to ℓ_1 is unchanged.

Figure 5 shows a slightly more involved example in which one of the paths from v to u includes two vertices ℓ_1 and ℓ_2 (labeled t = x + 1 and y = t + t). With this example, ignoring the incoming edges of Vertex ℓ_1 changes only the sizes of the backward slices taken with respect to ℓ_1 and ℓ_2, which have sizes one and two respectively when the incoming edges of ℓ_1 are ignored. Ignoring the incoming dependence edges of ℓ_2 has two effects. First, it reduces to one the size of the backward slice taken with respect to ℓ_2. Second, it reduces the size of all backward slices that include u by one because they no longer include ℓ_1; however, these backward slices continue to include v and the vertices that it depends on such as c, thus, the reduction is small as formalized in the next section.

Building on these examples, three graph patterns are considered and then proven correct. Each pattern bounds the reduction in the area under the MSG, A_{MSG}, that ignoring the incoming edges of a potential linchpin vertex may have. This reduction is formalized by the following two *small-impact properties*. Both properties, as well as the remainder of the paper, include a parameter, κ, that denotes a minimum percentage area reduction below which ignoring a vertex's incoming edges is deemed to have an insignificant impact on A_{MSG}. The selection of κ is subjective. In the empirical analysis of the next section, a range of values is considered.

When identifying vertices that have a small impact it is often useful to exclude from consideration the impact of a certain (small) set of vertices, V, on the area under the MSG, denoted $A_{MSG} \backslash V$. For SDG G, $A_{MSG} \backslash V$ is the sum of the slice sizes for each vertex of G that represents source code

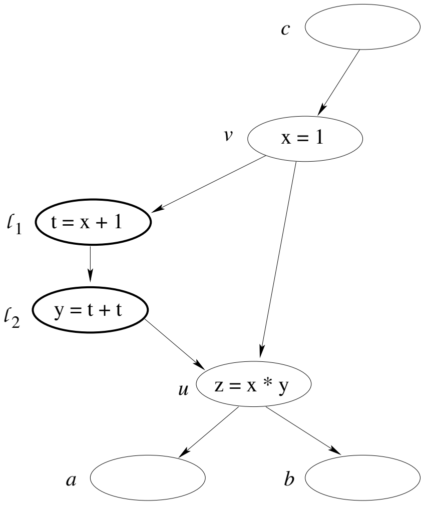


Fig. 5. A more complex example involving a path of two vertices. Again the bold vertices need not be considered as potential linchpins.

except those vertices in V. Formally, if SC is the set of vertices representing source code in an SDG then

$$A_{MSG} \backslash V = \sum_{v \,\in\, SC - V} |b(v)|$$

For example, in a graph where all vertices have exactly one edge targeting a common vertex, v, ignoring v's incoming edges reduces the area under the MSG by almost 50%, but has no effect on the area under $A_{MSG} \backslash \{v\}$, which ignores the area attributed to v. Such a reduction never corresponds to the breaking of a cluster and thus is uninteresting. The impact on $A_{MSG} \backslash V$ and $A_{MSG} \backslash \emptyset$ (i.e., with and without ignoring any vertices) is formalized by the following two definitions:

Definition 2 (STRONG SMALL-IMPACT PROPERTY).
Vertex v satisfies the strong small-impact property *iff ignoring the incoming dependence edges of v can reduce A_{MSG} (equivalently $A_{MSG} \backslash \emptyset$) by at most κ percent.*
□

Definition 3 (WEAK SMALL-IMPACT PROPERTY).
Given a (small) set for vertices V, Vertex v satisfies the weak small-impact property *iff ignoring the incoming dependence edges of v can reduce the area under $A_{MSG} \backslash V$ by at most κ percent.*
□

A Vertex v that satisfies the strong small-impact property also satisfies the weak small-impact property, but not vice versa. Thus the strong version is preferred. Both are introduced because sometimes the strong version cannot be proven to hold.

Before presenting the three theorems that prove the correctness of the three graph patterns, a more formal understanding of the SDG and interprocedural slicing is necessary. This begins with the definition of *same-level realizable path* (Definition 4) [Reps and Rosay 1995; Reps et al. 1995; Sharir and Pnueli 1981]. A same-level path begins and ends in the same procedure and corresponds to an execution where the call stack may temporarily grow deeper, but never shallower than its original depth, before eventually returning to its original depth. A realizable path respects calling context by matching returns with the correct call site. Several different terms have been used for paths that respect calling context, including feasible paths and realizable paths [Reps and Rosay 1995].

Definition 4 (SAME-LEVEL REALIZABLE PATH [REPS AND ROSAY 1995]).
Let each call-site vertex in SDG G be given a unique index from 1 to k. For each call site c_i, label the outgoing parameter-in edges and the incoming parameter-out edges with the symbols "$(_i$" and "$)_i$", respectively; label the outgoing call edge with "$(_i$". A path in G is a same-level realizable path *iff the sequence of symbols labeling the parameter-in, parameter-out, and call edges on the path is a string in the language of balanced parentheses generated from the nonterminal matched of the following grammar.*

$$\begin{aligned} \textit{matched} &\rightarrow \textit{matched}\ (_i\ \textit{matched}\ )_i \quad \textit{for}\ 1 \leq i \leq k \\ &\mid\ \epsilon \end{aligned}$$

□

The formalization next describes *valid paths* (Definition 5), the paths traversed while slicing: a vertex u in $b(v)$ is connected to v by a valid path. In the general case u and v are in different procedures called by a common ancestor. For example in Figure 2 if u is the vertex labeled x = x + 1 and v is the vertex labeled d = d * 2 then the path from u to v is a valid path. A valid path includes two parts. The first connects u to a vertex in the common ancestor (e.g., the vertex labeled t = x_out), while the second connects this vertex to v. Valid paths and their two parts are used to formally define six slicing operators (Definition 6). Finally, a set of path composition rules is introduced.

Definition 5 (VALID PATH [REPS AND ROSAY 1995]).
A path in SDG G is a (context) valid path iff the sequence of symbols labeling the parameter-in, parameter-out, and call edges on the path is a string in the language generated from nonterminal valid-path *given by the following context-free grammar where the non-terminals* b_1f_2-valid-path *and* b_2f_1-valid-path *take their names from the two slicing passes used in the implementation of interprocedural slicing.*

$$\begin{aligned} &\textit{valid-path} \rightarrow b_2f_1\textit{-valid-path}\quad b_1f_2\textit{-valid-path} \\ &b_2f_1\textit{-valid-path} \\ &\quad \rightarrow b_2f_1\textit{-valid-path matched}\ )_i \qquad \textit{for}\ 1 \leq i \leq k \\ &\quad\ \mid \textit{matched} \\ &b_1f_2\textit{-valid-path} \\ &\quad \rightarrow \textit{matched}\ (_i\ b_1f_2\textit{-valid-path} \qquad \textit{for}\ 1 \leq i \leq k \\ &\quad\ \mid \textit{matched} \end{aligned}$$

□

Example. For example, in Figure 2 there are two calls on inc, inc(a) and inc(t). In the SDG there are interprocedural parameter-in edges from each actual parameter to the vertex labeled x = x_in that represent the transfer of the actual to the formal. Symmetrically there are interprocedural parameter-out edges that represent the transfer of the returned value back to each caller. In terms of the grammar, the edges into inc are labeled $(_1$ and $(_2$ while the edges back to the call sites are labeled $)_1$ and $)_2$. Paths that match $(_1\ )_1$ represent calls through the first call site, inc(a) and those that match $(_2\ )_2$ represent calls through the second call site, inc(t). However any path that includes $(_1$ $)_2$ is not a valid path as it represents entering inc from the first call site but returning to the second. One such path connects the vertex labeled x_in = a to the vertex labeled d = d * 2

The interprocedural backward slice of an SDG taken with respect to Vertex v, $b(v)$, includes the program components whose vertices are connected to v via a *valid* path. The interprocedural forward slice of an SDG taken with respect to Vertex v, $f(v)$, includes the components whose vertices are reachable from v via a *valid path*. Both slices are computed using two passes. This leads to the following six slicing operators for slicing SDG G.

Definition 6 (INTERPROCEDURAL SLICING OPERATORS [HORWITZ ET AL. 1990; BINKLEY 1993]).
Let $u \rightarrow^* v$ denote a path of SDG edges. For vertex v in SDG G,

$$\begin{aligned}
b(v) &= \{u \in G \mid u \to^* v \textit{ is a valid path}\}\\
b_1(v) &= \{u \in G \mid u \to^* v \textit{ is a } b_1 f_2\textit{-valid path}\}\\
b_2(v) &= \{u \in G \mid u \to^* v \textit{ is a } b_2 f_1\textit{-valid path}\}\\
f(v) &= \{u \in G \mid v \to^* u \textit{ is a valid path}\}\\
f_1(v) &= \{u \in G \mid v \to^* u \textit{ is a } b_2 f_1\textit{-valid path}\}\\
f_2(v) &= \{u \in G \mid v \to^* u \textit{ is a } b_1 f_2\textit{-valid path}\}
\end{aligned}$$

□

Example. In Figure 2 let the v be the vertex labeled d = d * 2, u be the vertex labeled x = x + 1, and w be the vertex labeled t = x_out. In the SDG there is a $b_1 f_2$-valid path from w to v and a $b_2 f_1$-valid path from u to w. This places $w \in b_1(v)$, $u \in b_2(w)$, and $u \in b(v)$. Symmetrically, $w \in f_1(u)$, $v \in f_2(w)$, and $v \in f(u)$.

As noted before, the notation is overloaded such that each of the above slicing operators can be applied to a set of vertices V. The result is the union of the slices taken with respect to each vertex of V. For example, $b(V) = \cup_{v \in V} b(v)$; thus $f(v) = f_2(f_1(v))$ and $b(v) = b_2(b_1(v))$.

Finally, the search for linchpin vertices makes use of path composition, denoted $p_1 \circ p_2$, where path p_1's final vertex is the same as path p_2's first vertex. Some path compositions yield invalid paths. The following table describes the legal and illegal compositions.

	Path Combinations				
1	$b_1 f_2$-valid path	$\circ$	$b_1 f_2$-valid path	$\to$	$b_1 f_2$-valid path
2	$b_1 f_2$-valid path	$\circ$	$b_2 f_1$-valid path	$\to$	invalid path
3	$b_1 f_2$-valid path	$\circ$	valid path	$\to$	invalid path
4	$b_2 f_1$-valid path	$\circ$	$b_1 f_2$-valid path	$\to$	valid path
5	$b_2 f_1$-valid path	$\circ$	$b_2 f_1$-valid path	$\to$	$b_2 f_1$-valid path
6	$b_2 f_1$-valid path	$\circ$	valid path	$\to$	valid path
7	valid path	$\circ$	$b_1 f_2$-valid path	$\to$	valid path
8	valid path	$\circ$	$b_2 f_1$-valid path	$\to$	invalid path
9	valid path	$\circ$	valid path	$\to$	invalid path

Example. A valid path has two sections: the first (matching $b_2 f_1$-valid path) includes only unmatched $)_i$'s, while the second (matching $b_1 f_2$-valid path) includes only unmatched $(_i$'s. The first composition rule notes that composing two paths with only unmatched $(_i$'s leaves a path with only unmatched $(_i$'s. The second and third rules observe that the result of appending a path that includes unmatched $(_i$'s to a path that includes unmatched $)_i$'s is not a valid path. For example, in Figure 2 composing the $b_1 f_2$-valid path that connects the vertices labeled x_in = a and x = x + 1 with the $b_2 f_1$-valid path that connects the vertices labeled x = x + 1 and t = x_out results in a path that enters inc through one call site but exists through the other; this path is not a valid path. However, as seen in the table, it is always legal to prefix a path with a path that contains unmatched $)_i$'s (rules 4 and 6) and it is always legal to suffix a path with a path that contains unmatched $(_i$'s (rules 4 and 7).

Building on these definitions, Theorem 1 identifies a condition in which the strong small-impact property holds.

Theorem 1 (SMALL SLICE).
Let ℓ be a vertex from SDG G. If $|b(\ell)| \leq \kappa A_{MSG}/|G|$ or $|f(\ell)| \leq \kappa A_{MSG}/|G|$ then ℓ satisfies the strong small-impact property.

PROOF. For both cases, the worst case situation (where there are no other connections between the vertices of $b(\ell)$ and $f(\ell)$ except through ℓ) is illustrated in the left of Figure 6. Typically there are other connections between these vertices and thus the theorem is a conservative over-approximation of the actual reduction.

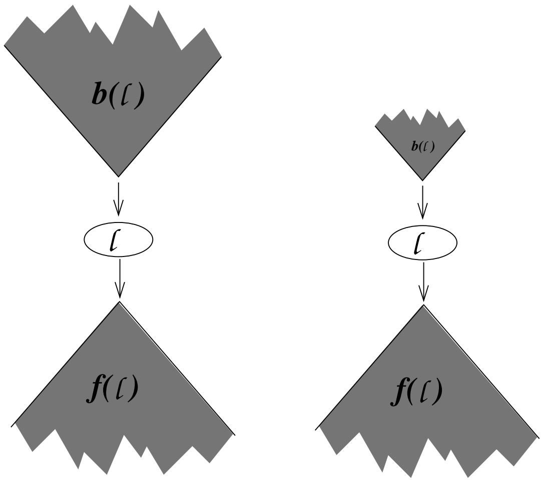


Fig. 6. Illustration of the cases in Theorem 1. The worst-case illustrated on the left involves an SDG's vertices being partitioned into three sets: $\{\ell\}$, $b(\ell)$, and $f(\ell)$. The two cases of the proof are illustrated in the middle and on the right.

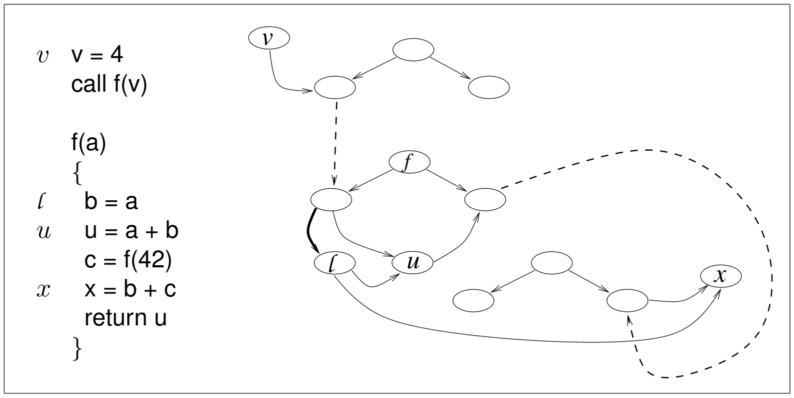
First, when $b(\ell)$ is small, the worst case area reduction is bounded by assuming that the vertices of $b(\ell)$ are not reachable from any vertex other than ℓ (illustrated in the center of Figure 6). In this case, ignoring the incoming dependence edges of ℓ reduces each backward slice that includes ℓ by $|b(\ell)|$. In the worst case ℓ is in every backward slice, which produces a maximal reduction of $|b(\ell)| \times |G|$. However, the assumption that $|b(\ell)| \leq \kappa A_{MSG}/|G|$ implies that the total reduction is at most κA_{MSG} and consequently the total area reduction is bound by κ.

Second, when $f(\ell)$ is small (illustrated in the right of Figure 6), note that the backward slices affected by ignoring the incoming edges of ℓ are those taken with respect to the vertices in $f(\ell)$. Thus if $|f(\ell)|$ is no more than $\kappa A_{MSG}/|G|$ then no more than $\kappa A_{MSG}/|G|$ backward slices are affected. Because the maximal reduction for a slice is to be reduced to size zero (a reduction of at most $|G|$), the total area reduction is bounded by $(\kappa A_{MSG}/|G|) \times |G|$, which is again bound by κ; therefore, if ℓ has a small backward or a small forward slice then it satisfies the strong small-impact property. □

As born out in the empirical investigation, the area reduction is often much smaller. For example it is unlikely that ℓ will be in every backward slice. Furthermore, the vertices of $b(\ell)$ often have connections to other parts of the SDG that do not go through ℓ. This is illustrated in the example shown in Figure 4 where ignoring the incoming edges of ℓ_1 does *not* remove v or its predecessors from backward slices that contain u. Formalizing this observation is done in the *dual-path property*, which is built on top of the definition for valid paths (Definition 5). The dual-path property holds for two vertices when they are connected by two valid paths where one includes the selected vertex ℓ and the other does not.

Definition 7 (DUAL-PATH PROPERTY).
Vertices ℓ, v, and u satisfy the Dual-Path Property*, written $dpp(\ell, v, u)$, iff there are valid paths $v\gamma u$ and $v\beta u$ such that $\ell \in \gamma$ and $\ell \notin \beta$.*
□

The value of the dual-path property is that ignoring the incoming dependence edges of ℓ leaves the endpoints connected via the other path. This observation expands the set of vertices for which one of the small-impact properties can be shown to hold. For example, in Figure 4, $dpp(\ell_1, v, u)$; thus, ignoring the incoming edges of ℓ_1 does not disconnect v and u. In this example, ignoring ℓ's incoming edges actually has no effect on backward slice sizes other than the obvious effect on the



```
v   v = 4
    call f(v)

    f(a)
    {
ℓ     b = a
u     u = a + b
      c = f(42)
x     x = b + c
      return u
    }
```

Fig. 7. An illustration of the the need for dpp_2. Only a subgraph of the SDG is shown.

backward slice taken with respect to ℓ_1. This is because $\forall v \in b(\ell)$, $dpp(\ell, v, u)$ and thus all backward slices that include ℓ also include u except the backward slice taken with respect to ℓ. The value of the dual-path property is that this reduction can often be shown to be much smaller than $|b(\ell_1)|$.

However, because interprocedural dependence is not transitive, an additional property is necessary. The issue arises when a backward slice s includes ℓ, u, and a vertex v' where $dpp(\ell, v', u)$, but the path from v' to u is a $b1f2$-valid path while u is encountered during the second pass of s. In this case excluding vertices v for which $dpp(\ell, v, u)$ overestimates the reduction (e.g., it errantly excludes v').

Example. This situation is illustrated in Figure 7 where the slice $b(x)$ includes Vertices ℓ, u, and v, and there are paths $v\gamma u$ and $v\beta u$ where $\ell \in \gamma$ and $\ell \notin \beta$. Consider the situation when the incoming edge of ℓ (shown in bold) is ignored. During Pass 2 of $b(x)$ the slice descends into f along the parameter-out edge through the recursive call (i.e., c = f(42)). Because u is returned by f, the assignment u = a + b is encountered while slicing; however, Pass 2 does not ascend to calling procedures and thus the slice does not ascend to the call f(v) and consequently does not reach v.

To correct for the over estimation, a dpp property is introduced to cover encountering u during a slice's second pass:

Definition 8 (SECOND PASS DUAL-PATH PROPERTY).
Vertices ℓ, v, and u satisfy the second-pass Dual-Path Property, written $dpp_2(\ell, v, u)$, iff there are two paths: valid path $v\gamma u$ and $b_2 f_1$-valid path $v\beta u$ such that $\ell \in \gamma$ and $\ell \notin \beta$.
□

Both dpp and dpp_2 are used in the following theorem to prove that when certain dual paths exist, ℓ satisfies the weak small-impact property (Definition 3). The weak version is used because the reduction for a backward slice taken with respect to certain vertices (e.g., ℓ) cannot be tightly bound. The first corollary to the theorem proves that under certain circumstances, the strong small-impact property also holds. For a vertex v in the slice $b(x)$, the proof in essence splits the paths connecting v and x at ℓ and u. The "top half" of these paths is captured by a dual-path property. while the "bottom half" is captured by the following vertex partitions based on a vertex's backward slice's inclusion of the two vertices ℓ and u.

Set 1 - vertices x where $\ell \notin b(x)$
Set 2 - vertices x where $\ell \in b(x)$ and $u \in b(x)$ where there is a path from u to x that does not include ℓ.
Set 3 - (the remaining vertices) vertices x where $\ell \in b(x)$ and $u \notin b(x)$ or $u \in b(x)$ but all paths from u to x include ℓ.

Example. These three sets can be illustrated using Figure 5 where ℓ is Vertex ℓ_1. In this example, Set 1 is $\{v, c\}$ because $b(v)$ and $b(c)$ do not include ℓ. For the remaining vertices (e.g., Vertex a) $b(x)$ includes ℓ. Set 2 is $\{u, a, b\}$ because the slice taken with respect to each of these vertices includes u. Finally, Set 3 is $\{\ell_2\}$ because u, a, and b are not in $b(\ell_2)$ and while c, v, ℓ_1, and ℓ_2 are in $b(\ell_2)$, all paths connecting them to ℓ_2 include ℓ_2.

Similar to the need for both *dpp* and *dpp*$_2$, Set 2 is further partitioned based on the slicing pass in which ℓ and u are encountered.

Set 2.11 - ℓ and u encountered during Pass 1
Set 2.12 - ℓ encountered during Pass 1 and u during Pass 2 (but not Pass 1)
Set 2.21 - ℓ encountered during Pass 2 (but not Pass 1) and u during Pass 1
Set 2.22 - ℓ and u encountered during Pass 2 (but not Pass 1)

Example. Vertices u, a, and b from Figure 5 are all in Set 2.11. Vertex x shown in Figure 7 is in Set 2.12 because ℓ is encountered during Pass 1 but u is not encountered until Pass 2.

Theorem 2 (DUAL-PATH WEAK IMPACT THEOREM).
Given an SDG G having V vertices, if there exists a Vertex u such that
for Set 2.11 $|b(\ell) - \{v \in G \mid dpp(\ell, v, u)\}| \leq \kappa A_{MSG}/V$,
for Set 2.12 $|b(\ell) - \{v \in G \mid dpp_2(\ell, v, u)\}| \leq \kappa A_{MSG}/V$,
for Set 2.21 $|b_2(\ell) - \{v \in G \mid dpp(\ell, v, u)\}| \leq \kappa A_{MSG}/V$,
and
for Set 2.22 $|b_2(\ell) - \{v \in G \mid dpp_2(\ell, v, u)\}| \leq \kappa A_{MSG}/V$
then ℓ satisfies the weak small-impact property. In particular, the area reduction under $A_{MSG}\backslash$Set 3 is bound by κ.

PROOF. The proof is a case analysis using the above partitions. For the first partition, Set 1, slices without ℓ are unchanged when ignoring ℓ's incoming edges. Next consider Set 2.11, which includes slices where ℓ and u are encountered during Pass 1. Assume that $b(x)$ is such a slice. Thus there are b_1f_2-valid paths from ℓ and u to x. Furthermore, reaching ℓ during Pass 1 means that $b(\ell) \subseteq b(x)$. Let v be a vertex in $b(\ell)$; thus v is also in $b(x)$. If $dpp(\ell, v, u)$ then there is a valid path from v to u that when composed with the b_1f_2-valid path from u to x produces a valid path from v to x; thus $v \in b(x)$ even when ℓ's incoming edges are ignored. Finally, $dpp(\ell, v, u)$ is true for most v's. In particular because $|b(\ell) - \{v \in G \mid dpp(\ell, v, u)\}| \leq \kappa A_{MSG}/V$, the reduction for vertices in Set 2.11 is bounded by κ.

Next, the argument for Set 2.12 is similar to that of Set 2.11 except that the path from u to x is a valid path rather than a b_1f_2-valid path. However the use of dpp_2 in the assumption that $|b(\ell) - \{v \in G \mid dpp_2(\ell, v, u)\}| \leq \kappa A_{MSG}/V$, means that the path from v to u is a b_2f_1-valid path and thus the composition again places $v \in b(x)$ even when ℓ's incoming edges are ignored. Similar to the case for Set 2.11, in this case because $|b(\ell) - \{v \in G \mid dpp_2(\ell, v, u)\}| \leq \kappa A_{MSG}/V$, the reduction for vertices in Set 2.12 is bounded by κ.

The argument for Set 2.21 is also similar to that for Set 2.11. The difference being that there is a valid path from ℓ to x rather than a b_1f_2-valid path and thus only v's in $b_2(\ell)$ need be considered. In other words, the path from v to ℓ is a b_2f_1-valid path. The remainder of the argument is the same as that for Set 2.11 except the assumption that $|b_2(\ell) - \{v \in G \mid dpp(\ell, v, u)\}| \leq \kappa A_{MSG}/V$ implies that the reduction for vertices in Set 2.21 is bounded by κ.

The argument for Set 2.22 parallels the above three arguments except that it uses the assumption that $|b_2(\ell) - \{v \in G \mid dpp_2(\ell, v, u)\}| \leq \kappa A_{MSG}/V$ to conclude that the reduction for vertices in Set 2.21 is bounded by κ.

Finally, Set 3 slices, which include ℓ but not u, are excluded because there is no way to bound their size change based on u. Combining the six cases, ℓ satisfies the weak small-impact property because the percent reduction in $A_{MSG}\backslash$Set 3 is bound by κ. □

In the preceding theorem the area reduction caused by slices from Set 3 is not tightly bound. To establish a bound the following corollary makes use of average reduction by balancing vertices whose backward slices cannot be tightly bound with backward slices that do not change (i.e., those of Set 1). This average reduction is formalized in the first of three corollaries.

Corollary 2.1 (STRONG IMPACT COROLLARY TO THEOREM 2). *Let $S_i = |\textit{Set } i|/|V|$ denote the proportion of slices in Set i. If $S_3 \times (1 - \kappa)/\kappa \leq S_1$ then the reduction in the area under $A_{MSG} \backslash \emptyset$ is bound by κ and ℓ satisfies the strong small-impact property.*

This is a foundational result that underpins the algorithm's performance improvement. If the guarding predicate ($S_3 \times (1 - \kappa)/\kappa \leq S_1$) holds, then the strong small impact property holds and, therefore, all vertices ignored in the search for linchpins will have little impact on dependence clusters. The term $S_3 \times (1 - \kappa)/\kappa$, the guarding predicate for Corollary 2.1, is referred to as the 'risk ratio,' because when the ratio is sufficiently small (thereby satisfying the guarding predicate) there is no risk in ignoring the associated vertices in the linchpin search.

The proof establishes when the corollary holds, but empirical research is needed to determine how often the guard is satisfied, indicating that the risk ratio is sufficiently low. If this does not happen sufficiently often then the performance improvements will be purely theoretical. This empirical question therefore forms the first research question addressed in Section 5.

PROOF. The strong small-impact property requires the total area reduction to be less than κ percent. To show that the reduction is at most κ, consider the largest reduction possible for each partition. Each reduction is given as a percentage of V. This yields the inequality $0 \times S_1 + \kappa \times S_2 + 1 \times S_3 \leq \kappa$ (because Set 1 slices are unchanged, from Theorem 2 slices in Set 2 are bound by κ, and slices from Set 3 can, in the worst case, include (no more than) the entire graph). The corollary follows from simplifying and rearranging this inequality as follows

$$\begin{array}{ll} \kappa S_2 + S_3 \leq \kappa & \\ S_3 \leq \kappa - \kappa S_2 & \\ S_3 \leq \kappa(1 - S_2) = \kappa(S_1 + S_3) & \text{as } 1 = S_1 + S_2 + S_3 \\ S_3 \leq \kappa S_1 + \kappa S_3 & \\ S_3 - \kappa S_3 \leq \kappa S_1 & \\ S_3(1 - \kappa)/\kappa \leq S_1 & \end{array}$$

□

Thus, for every vertex in Set 3 there needs to be $(1 - \kappa)/\kappa$ vertices in Set 1. For example, if κ = 5 must be 19 times larger than Set 3. In this case having 19 backward slices showing zero reduction and one backward slice showing (potentially) 100 reduction. Empirically, if Set 3 is kept below a size of about 20, then the corollary holds for all but the smallest of programs.

The statement of Theorem 2 requires the existence of a single Vertex u. It is useful to extend this definition from a single vertex u to a set of vertices U. Figure 8 shows an SDG fragment where $dpp(\ell, v1, u1)$ and $dpp(\ell, v2, u2)$ but not $dpp(\ell, v1, u2)$ and $dpp(\ell, v2, u1)$. Thus slices $b(x)$ that include a but not b require using $u1$, while those that include b but not a require $u2$ (those that include both a and b can use either). However, as the following corollary shows, it is possible to use the set $U = \{u1, u2\}$ in place of a single vertex u.

The following corollary generalizes this requirement from a single vertex u to a collection of vertices, U. The proof makes use of the following subsets of the backward slices of G. Again Set 2 is expanded, this time to take a particular $u \in U$ into account. Note that the original subsets were partitions. This was observed to simplify the presentation of Theorem 2 and its proof. It is not strictly necessary. When considering vertex x, the sets make use of the following subset of U: $U'(x) = \{u \in U \mid u \in b(x)$ and there is a valid path from u to x that does not include $\ell\}$.

Set 1 - vertices x where $\ell \notin b(x)$
Set 2 - vertices x where $\ell \in b(x)$ and $U'(x) \neq \emptyset$

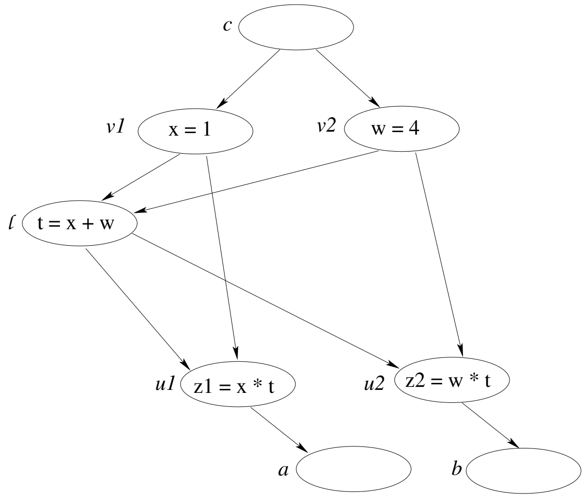


Fig. 8. Illustration using a set of vertices $U = \{u1, u2\}$.

Set 2.11 - ℓ encountered during Pass 1 and $\exists u \in U'(x)$ encountered during Pass 1
Set 2.12 - ℓ encountered during Pass 1, $\exists u \in U'(x)$ encountered during Pass 2, and $\nexists u \in U'(x)$ encountered during Pass 1
Set 2.21 - ℓ encountered during Pass 2 (but not Pass 1) and $\exists u \in U'(x)$ encountered during Pass 1
Set 2.22 - ℓ encountered during Pass 2 (but not Pass 1), $\exists u \in U'(x)$ encountered during Pass 2, and $\nexists u \in U'(x)$ encountered during Pass 1
Set 3 - vertices x where $\ell \in b(x)$ and $U'(x) = \emptyset$

The proof uses the following extensions of the definitions for *dpp* and *dpp*$_2$ to a set of vertices U:
$dpp(\ell, v, U) = \exists u \in U$ s.t. $dpp(\ell, v, u)$
$dpp_2(\ell, v, U) = \exists u \in U$ s.t. $dpp_2(\ell, v, u)$

Corollary 2.2 (MULTI-PATH IMPACT).
If there exists a collection of vertices U such that
for Set 2.11 $|b(\ell) - \{v \in G \mid dpp(\ell, v, U)\}| \leq \kappa A_{MSG}/V$,
for Set 2.12 $|b(\ell) - \{v \in G \mid dpp_2(\ell, v, U)\}| \leq \kappa A_{MSG}/V$,
for Set 2.21 $|b_2(\ell) - \{v \in G \mid dpp(\ell, v, U)\}| \leq \kappa A_{MSG}/V$,
and
for Set 2.22 $|b_2(\ell) - \{v \in G \mid dpp_2(\ell, v, U)\}| \leq \kappa A_{MSG}/V$
then ℓ satisfies the weak small-impact property. In particular, the area reduction under $A_{MSG} \backslash$Set 3 is bound by κ. Furthermore, if $|$Set 3$| \times (1 - \kappa)/\kappa \leq |$Set 1$|$ then ℓ satisfies the strong small-impact property.

PROOF. As with the proof of Theorem 2, Set 1 slices are unchanged when ignoring the incoming edges of ℓ. For each subset of Set 2 the proof is the same as in the theorem using one of the $u \in U$. Thus for $A_{MSG} \backslash$Set 3, the area reduction is bound by κ and the weak small-impact property holds. Finally, when $|$Set 3$| \times (1 - \kappa)/\kappa \leq |$Set 1$|$, then, following Corollary 2.1, the area under $A_{MSG} \backslash \emptyset$ is also bound by κ; thus, the strong small-impact property holds. □

The final corollary observes that it is not strictly necessary for each set to produce a reduction less than κ as one set being above this bound can be compensated for by another being below the bound. This observation is formalized in the final corollary of Theorem 2.

Corollary 2.3 (AVERAGE IMPACT).
To bound the area reduction for each set required in Corollary 2.2 is not strictly necessary. Rather it is sufficient to bound the weighted average reduction.

PROOF. Assume the number of vertices in Sets 2.11 and 2.12 are the same. If $|b(\ell) - \{v \in G \mid dpp(\ell, v, U)\}|$ is greater than $\kappa A_{MSG}/V$ by the same amount that $|b(\ell) - \{v \in G \mid dpp_2(\ell, v, U)\}|$ is less than $\kappa A_{MSG}/V$, then the total reduction is bounded by κ. In the general case, when the sets are not the same size, a weighted average is required. □

The final theorem exploits a property of the construction of the SDG vertices that represent variable declarations.

Theorem 3 (DECLARATION IMPACT).
All declaration vertices satisfy the weak small-impact property. And if Set 1 includes more than $(1 - \kappa)/\kappa$ vertices then the strong small-impact property holds as well.

PROOF. Consider declaration vertex d as ℓ. By construction, there is a single incoming edge to d, $p \rightarrow d$ from the procedure entry vertex, p, an edge $d \rightarrow o$ to each vertex, o, representing an occurrence (use or definition) of the declared variable, and a path of control edges $p\beta o$ where $d \notin \beta$. The proof follows from Corollary 2.2 of Theorem 2 where $U = \{o \mid o \text{ is an occurrence vertex}\}$ because there is a path (of control edges) β from p to every vertex $u \in U$ and this path does not include d; thus, omitting d's single incoming edge disconnects no vertices and consequently leaves Set 1 and Set 2 unaffected. Therefore the weak small-impact property holds for a declaration vertex because the area change for $A_{MSG}\backslash$Set 3 is zero and thus bound by κ. Finally, because Set 3 is the singleton set $\{d\}$, |Set 3| is one, and thus |Set 3|$(1-\kappa)/\kappa$, simplifies to $(1-\kappa)/\kappa$. Consequently, by Corollary 2.1 the strong small-impact property holds, assuming that Set 1 includes at least $(1-\kappa)/\kappa$ vertices. □

4. AN EFFICIENT LINCHPIN SEARCH ALGORITHM

This section presents an efficient algorithm for finding potential linchpins. The algorithm is based on the three theorems from Section 3 that remove vertices from linchpin consideration. A preprocessing step to the algorithm removes vertices with no incoming edges. Such vertices cannot be a part of a cluster. Initially these are entry vertices of uncalled procedures. Removal of these entry vertices may leave other vertices with no incoming edges; thus the removal is applied recursively. Unlike the simple deletion of uncalled procedures, this edge removal allows clusters in (presently) uncalled procedures to be considered.

The search for linchpins needs to slice while avoiding the (incoming edges of a) potential linchpin. This is supported in the implementation by marking the potential linchpin as *poison* and then using slicing operators that stop when they reach a poison vertex.

Definition 9 (POISON AVOIDING SLICE).
The slice $pb(v)$ is the same as $b(v)$ except that slicing stops at vertices marked as poison. *In other words, only valid paths free from poison vertices are considered. The remaining slicing operators, b_1, b_2, f, f_1, and f_2 have poison-vertex-avoiding variants pb_1, pb_2, pf, pf_1, and pf_2, respectively. As with b and f, $pb(v) = pb_2(pb_1(v))$ and $pf(v) = pf_2(pf_1(v))$.*
□

The algorithm for vertex exclusion, given in function *exclude* of Figure 9, takes three inputs: the vertex to test, ℓ, a fringe search depth k, and a threshold κ percent. It returns true if ℓ can be excluded from consideration as a linchpin. The second parameter k, investigated in Section 5.3, is used to tune the algorithm to speculatively search forward for alternative paths around a potential linchpin. The algorithm first checks if ℓ has a small backward or a small forward slice, or is a declaration vertex. As shown in Theorems 1 and 3 such vertices cannot be linchpins. For remaining vertices the search for dual paths is made. This is done by identifying three sets of vertices: *poison*, *core*, and *fringe*. The sole vertex in the set *poison* is ℓ. Being marked as poison causes the poison avoiding slices to

```
boolean fringe_search(Vertex $\ell$, Vertex $v$,
                      depth $k$, percent $\kappa$)
{
  let success = true and fail = false
  if $v$ marked as core
    return success          // already processed $v$
  mark $v$ is core
  foreach edge $v \to u$
    if area_reduction_bound($\ell$, $u$) $> \kappa * A_{MSG}/V$
      if $k$ == 0
        return fail
      else
        if fringe_search($\ell$, $u$, $k-1$, $\kappa$) == fail
          return fail
  return success
}

boolean exclude(Vertex $\ell$, depth $k$, percent $\kappa$)
{
  if $\ell$ is a declaration vertex
     or $|b(\ell)| < \kappa * A_{MSG}/V$
     or $|f(\ell)| < \kappa * A_{MSG}/V$
    return true
  clear_all_marks()
  Mark $\ell$ poison
  return fringe_search$(\ell, \ell, k, \kappa) ==$ success
}
```

Fig. 9. The linchpin exclusion algorithm.

stop at ℓ. Vertices in the set *core* are reachable from ℓ along paths that contain no more than k edges (k is the function's second parameter). Finally, the vertices of the set *fringe* have an incoming edge from a *core* vertex but are not *core* vertices. The intent is that the *fringe* vertices play the role of U from Corollary 2.2 of Theorem 2.

Function *area_reduction_bound* shown in Figure 10 computes an upper bound on the area reduction for Vertex ℓ where Vertex u is one of the vertices from the fringe (the set U in Corollary 2.2 of Theorem 2). In the computation, the size of Set i measures the width of the MSG impacted (i.e., the number of backward slices impacted). This is multiplied by a bound on the height (slice size) of the impact (the second multiplicand of each product). To ensure the strong version of the small-impact property, the impact of Set 3 must be included (the last list of the function). The impact of this set is ignored when using the weak small-impact property and thus the contribution of Set 3 is ignored.

Two examples are used to illustrate the algorithm. First, consider ℓ_1 from Figure 4 with depth $k = 0$. This makes ℓ_1 the only *core* vertex and u the only *fringe* vertex. In this case, Set 1 = $\{c, v\}$, Set 2.11 = $\{u, a, b\}$, Sets 2.12, 2.21, and 2.22 are all empty, and Set 3 = $\{\ell_1\}$. Furthermore $b(\ell) = \{\ell, v, c\}$ as does $pb(u)$. Thus $|b(\ell) - pb(u)|$ is zero. Function *area_reduction_bound* returns $2 \times 0 + 3 \times 0 + 0 \times 0 + 0 \times 0 + 0 \times 0$ when considering $A_{MSG}\backslash$Set 3. And adds 1×3 when considering A_{MSG}. Thus the change in $A_{MSG}\backslash$Set 3 is 0 vertices and the change in $A_{MSG}\backslash\emptyset$ is 3 vertices (15% of A_{MSG}). The 15% reduction for $A_{MSG}\backslash\emptyset$ is comparatively large because the SDG is very small.

As a second example, consider ℓ_1 from Figure 5 with depth $k = 0$. This makes ℓ_1 the only *core* vertex and ℓ_2 the only fringe vertex. In this case, there are no dual paths connecting the fringe to v and c causing a potentially large reduction. However using depth $k = 1$, places ℓ_1 and ℓ_2 in the

```
int area_reduction_bound(Vertex $\ell$, Vertex $u$)
{
 Let
   Set 1 = $V - f(\ell)$
   Set 2.11 = $f_2(\ell) \cap pf_2(u)$
   Set 2.12 = $f_2(\ell) \cap (pf(u) - pf_2(u))$
   Set 2.21 = $(f(\ell) - f_2(\ell)) \cap pf_2(u)$
   Set 2.22 = $(f(\ell) - f_2(\ell)) \cap (pf(u) - pf_2(u))$
   Set 3 = $f(\ell) - \cup_i$ Set 2.$i$
 in
   return |Set 1| $\times$ 0
         $+$ |Set 2.11| $\times$ $|b(\ell) - pb(u)|$
         $+$ |Set 2.12| $\times$ $|b(\ell) - pb_2(u)|$
         $+$ |Set 2.21| $\times$ $|b_2(\ell) - pb(u)|$
         $+$ |Set 2.22| $\times$ $|b_2(\ell) - pb_2(u)|$
         // for the strong version include
         $+$ |Set 3| $\times$ $| \cup_{x \in core} b(x)|$
 end
}
```

Fig. 10. Computation of the bound on the area reduction arising from ignoring the incoming edges of ℓ.

core and u on the fringe. Because there are dual paths connecting u to v and c, the area reduction is smaller. Further increasing k makes no difference because once a fringe vertex is found along a path the recursive search stops. Finally the need for a set U with $k = 0$ is illustrated in Figure 8.

The complexity of *area_reduction_bound* is given in terms of the number of vertices V and edges E, the maximal number of edges incident on a vertex, e, and the search depth k. Note that in the worst case e is $O(V)$, but in practice is much smaller; thus it is retained in the statement of the complexity. The implementation of *area_reduction_bound* involves eight slices each of which take $O(E)$ time. The slicing algorithm marks each vertex encountered with a sequence number. This makes it possible to compute various set operation while slicing. For example, when computing the size of Set 2.11, assuming that the current sequence number is n, computing $f_2(\ell)$ leaves the vertices of this slice marked n. Subsequently, while computing $pf_2(u)$, which marks vertices with sequence number $n + 1$, if a vertex's mark goes from n to $n + 1$ then the count of vertices in the intersection $f_2(u) \cap pf_2(u)$ is incremented.

The complexity of the recursive function *fringe_search* involves at most $k + 1$ recursive calls. During each call, the foreach loop executes e times and, from the body of the loop, the call to *area_reduction_bound*'s complexity of O(E) dominates. Thus the complexity of a call to *fringe_search* is $O((Ee)^{k+1})$. For the untuned version $k + 1$ is the constant 1 and thus the complexity simplifies to is $O(Ee)$.

The complexities of the naïve algorithm and the untuned algorithm are the same as in the worst case it is possible that no vertices are excluded. In theory the tuning can be more expensive (when $(Ee)^{k+1}$ is greater than E^2. Empirically, this occurs only once in the range of k's considered (see the speedup for program flex-2-5-4 shown in Figure 20).

For a vertex v, the complexity of the three steps of *exclude* and the MSG construction are $O(1)$ for the declaration check, $O(E)$ for the small slice check, $O((Ee)^{k+1})$ for the fringe search, and $O(VE)$ to construct the MSG. This simplifies to $O((Ee)^{k+1} + VE)$. For untuned algorithm where $k = 0$, $O((Ee)^{k+1})$ simplifies to $O(Ee)$ and, because e is $O(V)$, the overall complexity simplifies to $O(VE)$, the same as that of the naïve algorithm.

The correctness of *exclude* is shown in Theorem 4, which proves that the algorithm satisfies the weak small-impact property, and when the last line of function *area_reduction_bound* is included, the strong small-impact property. Theorem 4 establishes that *exclude* is a conservative approximation to the weak small-impact property. Thus all excluded vertices are guaranteed to have a small impact and are consequently not linchpins.

Theorem 4 (ALGORITHM CORRECTNESS).
If function exclude *from Figure 9 returns* true *for Vertex ℓ, then ℓ satisfies the weak small-impact property.*

PROOF. There are two steps. The first shows that $\{v \in G \mid dpp(\ell, v, u)\} \subseteq pb(u)$ and that $\{v \in G \mid dpp_2(\ell, v, u)\} \subseteq pb_2(u)$. Then the remainder of the proof shows that the fringe satisfies the requirements of the set U from the Multi-Path Impact Corollary (Corollary 2.2 of Theorem 2).

To begin with, observe that $dpp(\ell, v, u)$ requires a valid path from v to u that excludes ℓ. By Definition 6 this valid path implies that $v \in b(u)$ and furthermore, because the path excludes ℓ, $v \in pb(u)$. Thus, $\{v \in G \mid dpp(\ell, v, u)\} \subseteq pb(u)$. The argument that $\{v \in G \mid dpp_2(\ell, v, u)\} \subseteq pb_2(u)$ is the same except that $b_2 f_1$-valid paths are used in place of (full) valid paths. These two subset containments imply that

$$\begin{aligned}
&|b(\ell) - pb(u)| \leq |b(\ell) - \{v \in G \mid dpp(\ell, v, U)\}| \\
&|b(\ell) - pb_2(u)| \leq |b(\ell) - \{v \in G \mid dpp_2(\ell, v, U)\}| \\
&|b_2(\ell) - pb(u)| \leq |b_2(\ell) - \{v \in G \mid dpp(\ell, v, U)\}| \\
&|b_2(\ell) - pb_2(u)| \leq |b_2(\ell) - \{v \in G \mid dpp_2(\ell, v, U)\}|
\end{aligned}$$

The second step of the proof establishes that the six sets (Set 1, Set 2.11, Set 2.12, Set 2.21, Set 2.22, and Set 3) used in Theorem 2 are equivalent to those computed at the top of function *area_reduction_bound* of Figure 10. For each set the argument centers on the observation that when v is in $b(x)$ then x is in $f(v)$.

For Set 1, observe that backward slices *with* ℓ (i.e., those that include ℓ) are those in $f(\ell)$; thus backward slices *without* ℓ are those not in $f(\ell)$, which is the set of vertices $V - f(\ell)$. For Set 2.11, first observe that f_2 is the dual of b_1; thus if $v \in b_1(u)$ then $u \in f_2(v)$. This means that all vertices whose slices include ℓ and u during Pass 1 are in the forward Pass 2 slice of both ℓ and u and thus in $f_2(\ell) \cap pf_2(u)$.

As with Set 2.11, for Set 2.12 all vertices whose backward slices include ℓ during Pass 1 are in the forward Pass 2 slice of ℓ. Set 2.12 also includes backward slices where u is included during Pass 2 but not Pass 1. These are backward slices taken with respect to the vertices in $pf(u) - pf_2(u)$; thus Set 2.12 includes the vertices in $f_2(\ell) \cap (pf(u) - pf_2(u))$. The arguments for Set 2.21 and 2.22 are similar.

Finally, for a Set 3 vertex, x, the backward slice $b(x)$ includes ℓ but not u. The vertices that include ℓ in their slice are those of $f(\ell)$. Those that also include u are in Set 2; thus Set 3 is efficiently computed as $f(\ell) - \cup_i \text{Set } 2.i$.

The final step in the proof is to observe that by construction the function *fringe_search* identifies a set of fringe vertices that fulfill the role of the set U from the multi-path impact corollary (Corollary 2.2 of Theorem 2). Thus the average impact corollary (Corollary 2.3) of Theorem 2 implies that the average reduction for $u \in U$ from ignoring the incoming edges of ℓ is bounded by κ. □

Corollary 4.1 (STRONG ALGORITHM).
If $|core| \times (1 - \kappa)/\kappa \leq |Set\ 1|$ and $exclude(\ell)$ then ℓ satisfies the strong small-impact property.

PROOF. Theorem 4 proves that the weak small-impact property holds for $A_{MSG} \backslash$Set 3. Thus only Set 3 need be considered. By construction all backward slices that encounter a *core* vertex, except those taken with respect to *core* vertices, also encounter a *fringe* vertex. This implies that Set 3 includes at most the *core* vertices. Consequently, under the assumption that $|core|$ is bound by $|core| \times (1 - \kappa)/\kappa \leq |\text{Set } 1|$, the strong small-impact property holds. □

5. EMPIRICAL STUDY OF PERFORMANCE IMPROVEMENT

To empirically investigate the improved search for linchpin vertices, four research questions are considered and a study designed and executed for each. The design includes considering κ set to 1%, 10%, and 20%. Based on visual inspection of hundreds of MSGs, a 1% or smaller reduction is never associated with the breaking of a cluster. The 1% reduction is thus included as a conservative bound on the search. At the other end, while, 20% might seem too liberal, it was chosen as an optimistic bound to investigate the speed advantages that come from the (potential) exclusion of a larger number of vertices. Finally, the 10% limit represents a balance point between the likelihood that no linchpin vertices are missed and the hope the only linchpin vertices are considered.

Three experiments were designed to empirically investigate the following four research questions. The experiments involve almost half a million lines of code from 38 subject programs written predominantly in C with some C++. Summary statistics concerning the programs can be found in Figure 11.

- RQ1: For sufficiently large programs, is the predicate of the Strong Impact Corollary (Corollary 2.1 of Theorem 2) satisfied?
  This is an important validation question. Recall that the guarding predicate of Corollary 2.1 determines whether the risk ratio is sufficiently low that we can be certain that the associated vertex set has no effect in dependence clusters (and can therefore be safely ignored). If this predicate is satisfied in most cases, then the theoretical performance improvements defined in Section 3 will become achievable in practice. Therefore, this is a natural first question to study.
- RQ2: Does the new algorithm significantly improve the linchpin-vertex search?
  This research question goes to the heart of the empirical results in the paper. It asks whether the basic algorithm (with no tuning) is able to achieve significant performance enhancements. If this is the case, then there is evidence to support the claim that the algorithm is practically useful: it can achieve significant performance enhancements with no tuning required.
- RQ3: What is the effect of tuning the fringe search depth on the performance of the algorithm?
  This research question decomposes into two related subquestions:
  - RQ3.1: What is the impact of the tuning parameter (the fringe search depth) on the search? This includes identifying general trends and the specific optimal depth for each of the three values for κ.
  - RQ3.2: Using the empirically chosen best depth, what is the performance improvement that tuning brings over the naïve search? This is measured in both vertices excluded and time saved.

  The aim of RQ3 is to provide empirical evidence concerning the effects of tuning. This may be useful to the software engineer who seeks to get the best performance from the algorithm. For those software engineers who would prefer to simply identify linchpins using an algorithm ‘out of the box’, the basic (untuned algorithm) should be used. For these ‘end users’ the answer to RQ2 is sufficient. The answer to RQ3 may also be relevant to researchers interested in finding ways to further improve and develop fast linchpin search algorithms or those working on related dependence analysis techniques.

The experiments were run on five identical Linux machines running Ubuntu 10.04.3 LTS and kernel version 2.6.32-42. Each experimental run is a single process executed on a 3.2GHz Intel 6-Core CPU. To help stabilize the timing, only five of the processor’s six cores were used for the experimental runs.

The SDGs used in this study were built using CodeSurfer 1.9p3 [Grammatech Inc. 2002]. CodeSurfer can build SDGs for the complete C and C++ languages. For example, it precisely treats structure fields [Yong et al. 1999] and performs extensive pointer analysis using the algorithm proposed by Fahndrich et al. [Fahndrich et al. 1998], which implements a variant of Andersen’s points-to algorithm [Andersen 1994] (this includes parameter aliasing).

Program	LoC	SLoC	Vertices	Edges	SVertices
fass	1,140	978	4,980	12,230	922
interpreter	1,560	1,192	3,921	9,463	947
lottery	1,365	1,249	5,456	13,678	1,004
time-1.7	6,965	4,185	4,943	12,315	1,044
compress	1,937	1,431	5,561	13,311	1,085
which	5,407	3,618	5,247	12,015	1,163
pc2c	1,238	938	7,971	11,185	1,749
wdiff.0.5	6,256	4,112	8,291	17,095	2,421
termutils	7,006	4,908	10,382	23,866	3,113
barcode	5,926	3,975	13,424	35,919	3,909
copia	1,170	1,112	43,975	128,116	4,686
bc	16,763	11,173	20,917	65,084	5,133
indent	6,724	4,834	23,558	107,446	6,748
acct-6.3	10,182	6,764	21,365	41,795	7,250
gcc.cpp	6,399	5,731	26,886	96,316	7,460
gnubg-0.0	10,316	6,988	36,023	104,711	9,556
byacc	6,626	5,501	41,075	80,410	10,151
flex2-4-7	15,813	10,654	49,580	105,954	11,104
space	9,564	6,200	26,841	74,690	11,277
prepro	14,814	8,334	27,415	75,901	11,745
oracolo2	14,864	8,333	27,494	76,085	11,812
tile-forth-2.1	4,510	2,986	90,135	365,467	12,076
EPWIC-1	9,597	5,719	26,734	56,068	12,492
userv-0.95.0	8,009	6,132	71,856	192,649	12,517
flex2-5-4	21,543	15,283	55,161	234,024	14,114
findutils	18,558	11,843	38,033	174,162	14,445
gnuchess	17,775	14,584	56,265	165,933	15,069
cadp	12,930	10,620	45,495	122,792	15,672
ed	13,579	9,046	69,791	108,470	16,533
diffutils	19,811	12,705	52,132	104,252	17,092
ctags	18,663	14,298	188,856	405,383	20,578
wpst	20,499	13,438	140,084	382,603	20,889
ijpeg	30,505	18,585	289,758	822,198	24,029
ftpd	19,470	15,361	72,906	138,630	25,018
espresso	22,050	21,780	157,828	420,576	29,362
go	29,246	25,665	144,299	321,015	35,863
ntpd	47,936	30,773	285,464	1,160,625	40,199
csurf-pkgs	66,109	38,507	564,677	1,821,811	43,044
sum	494,025	342,949	2,694,603	7,953,166	465,914

Fig. 11. Characteristics of the subject programs studied. LoC and SLoC (non-blank - non-comment Lines of Code) are source code line counts as reported by the linux utilities wc and sloc. Vertices and Edges are counts from the resulting SDG while SVertices is a count of the source-code-representing vertices. (In this and the remaining figures, programs are shown ordered by size based on SVertices.)

The subset of the vertices in the SDG that represent source code are considered as potential linchpins and counted when determining slice size. An SDG includes “pseudo” vertices that do not directly represent source code. For example, when a call to procedure P may reference a global variable, the SDG includes vertices representing the passing of the global to P. Finally, vertices are used in place of a source-level artifact such as lines of code because vertex count is more consistent across programming styles.

5.1. RQ1: Empirical Validation of Strong Small-Impact Property

This section empirically investigates how often the predicate of the Strong Impact Corollary (Corollary 2.1 of Theorem 2) is satisfied. To do so, the linchpin search was configured to produce the MSG regardless of the value returned by *exclude* and then verify that the reduction was less than κ percent of A_{MSG}. Therefore, the function *area_reduction_bound* omits the final term for Set 3. Vertices that produce a reduction greater than κ were then inspected by hand to determine whether the vertices of Set 3 were the cause.

Because of the execution time involved, six of the larger programs were not considered (the largest would take a year to complete). The remaining programs include 272,839 source-code representing vertices. These were considered for $\kappa = 1\%, 10\%$, and 20%. Of the resulting 818,517 executions, no violations were uncovered. Looking for *near miss* violations uncovered a strong trend between near misses and program size with violations growing less likely as program size increased. Given this relationship several very small programs were considered. This uncovered four violations all for $\kappa = 1\%$; thus empirically for $\kappa = 10\%$ and $\kappa = 20\%$ the Strong-Impact Property always held. The four violations, two each from programs with 428 and 723 SLoC, were inspected and it was confirmed that they came from vertices of the *core* (i.e., those from Set 3); thus validating the implementation of *exclude*. Basically violations only occur with very small programs where a small number of vertices can have a large percentage impact.

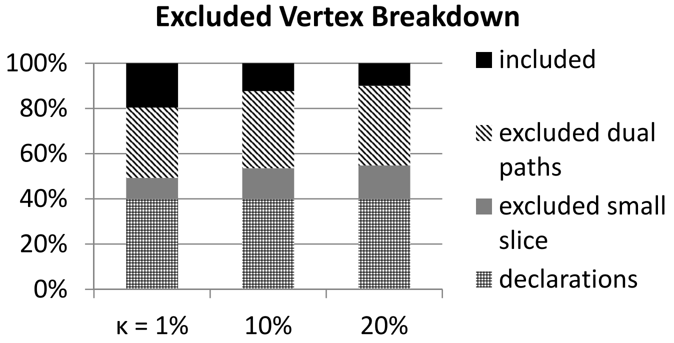
In summary, the Strong Impact Corollary (Corollary 2.1 of Theorem 2) holds for 100% of the 272,839 vertices considered. This result provides an affirmative answer to Research Question RQ1: for sufficiently large programs, the predicate of the Strong Impact Corollary (Corollary 2.1 of Theorem 2) is satisfied.

5.2. RQ2: Time Improvement

For each value of κ considered, the average impact is broken down in Figure 12. Over all 38 programs, the figure shows the weighted average percentage of vertices in each of four categories: declarations, small slice, dual paths, and included. The area reduction computation and thus the production of the MSG need only be performed for the last category. The first category, declarations, is the same regardless of κ. In the empirical study this category also includes vertices matching two other trivial patterns: those with no incoming edges and those with no outgoing edges. Approximately 40% of the vertices in this category are edge-less and 60% are true declarations. Category one is shown in Figure 12 using a grid pattern. Next, the light-gray section shows how an increase in κ leads to an increase in the percentage of vertices that can be excluded because they have a small slice. This increase is expected, but what is interesting here is how little change occurs when κ goes from 10% to 20%. The next category, shown with diagonal lines, is the percentage excluded because dual-paths were found in the graph. The values are largely independent of κ suggesting that when dual paths exist, they show a small reduction or a large reduction, but rarely something in between. The final category contains the vertices for which the MSG must be produced. As detailed in the table, 80 to 90% of the MSGs need not be produced. Because the execution time savings is the same regardless of the reason for excluding a vertex, the time saving attributed to the discovery of dual-paths is the percentage of the vertices excluded because of dual paths, which is approximately 30-35% of the excluded vertices.

Taking 10% of the area under the MSG as a cutoff for a cluster being a large cluster (the middle choice), almost 88% of the vertices are removed from consideration as linchpins. Furthermore, increasing this cutoff to 20% provides only a modest 2 percentage point increase; thus, there is empirically little benefit in the increase.

To provide a closer look at the exclusion, Figure 13 shows six example programs. Each triplet of bars represents κ of 1%, 10%, and 20%. Five of the six programs show an increase in the vertices excluded because they have a small slice. The exception, ed, is a program that includes a single very



κ

partition \ κ	1%	10%	20%
included	19.6%	12.3%	10.0%
excluded dual paths	31.2%	34.1%	35.1%
excluded small slice	9.4%	13.7%	15.0%
declarations	39.9%	39.9%	39.9%
total excluded	80.4%	87.7%	90.0%

Fig. 12. Weighted average percentage of vertices in four categories for the three values of κ. The figures do not sum to 100% due to rounding errors.

large dependence cluster [Harman et al. 2009]. Because of this large cluster, ed's slices are either large (for those vertices in the cluster) or small; thus, there is no benefit to increasing κ within a reasonable range. Perhaps because there are so few vertices with small slices, the search for vertices with dual paths finds the most success in ed, which also shows the most improvement with an increase in κ.

Finally, Figures 14 and 15 show the runtime speedup. Figure 14 shows the speedups for all programs with the averages shown on the far right. The averages are shown alone in Figure 15. The average speedup for κ of 1%, 10%, and 20% is 8x, 14x, and 18x, respectively. This speedup directly parallels the reduction in the number of vertices that must be considered. Most programs show a similar pattern, where increasing the value of κ brings a clear benefit. The program pc2c (a Pascal to C converter) is an outlier as it does not gain much speedup for larger values of κ. Looking at the source code for this program, 70% of the code is in main and the remaining functions have few parameters or local variables. The implication of this is that declaration vertices account of only 10% of the excluded vertices. This is dramatically less than the average of 40% and accounts for the overall difference.

Statistically all three versions provide a significant improvement in runtime over the naïve search ($p < 0.0001$ for all three tests). Because the runtimes are not normally distributed, the non-parametric Friedman's paired test is used. A paired test is appropriate because the same population of programs is used with each algorithm. Head to head the runtimes for κ of 10% and 20% are significantly less than that for κ of 1% ($p = 0.0003$ and $p < 0.0001$, respectively), and finally, the

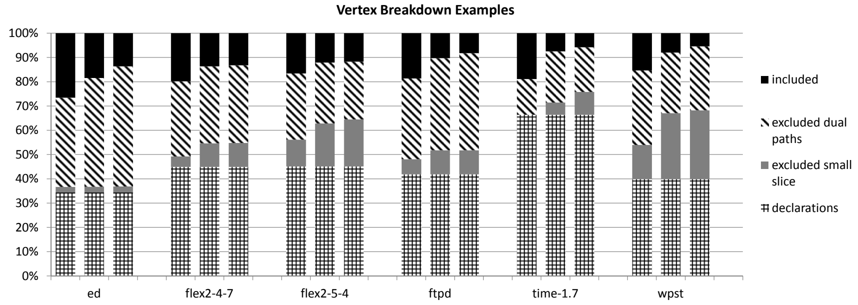


Fig. 13. Excluded vertex breakdown for six sample programs designed to show the variation of impact that κ has at the program level. Each triplet includes the breakdown for κ of 1%, 10%, and 20%.

runtime for κ of 10% is statistically higher than κ of 20% ($p = 0.0066$). The weakening strength of these p scores serves to underscore that increasing κ brings a diminishing performance improvement (along with the increased risk of missing a linchpin vertex). In summary, the data and statistics provide an affirmative answer to research question RQ2: the new algorithm significantly improves the linchpin-vertex search.

5.3. RQ3.1: Depth Study

With an increase in search depth comes an increased potential to find a fringe. However, this is done at the cost of searching a greater portion of the graph. Because the search for a fringe vertex stops when an appropriate vertex is found, increasing depth can only increase the number of excluded vertices. As this study shows, initially greater depth is beneficial, but this benefit wanes as *depth* (and thus search cost) increases.

The third study considers the impact of tuning the exclusion's search depth. To begin with, Figure 16 shows the percentage increase in excluded vertices when compared to the number excluded at depth zero. For all three values of κ the curves show an initial rapid increase that then tapers off. While it is not feasible to run these experiments for larger depths with all programs because the execution times become excessive, collecting data for a subset shows that the curves continue this taper as depth increases.

The increase in excluded vertices has a cost. For the three values of κ, Figure 17 shows the impact of search depth on the cost of the search. From the upper chart it is clear that for smaller search depths, MSG generation dominates the execution time and thus excluding additional vertices is worth the additional search effort. However, as depth increases above five, the time spent conducting

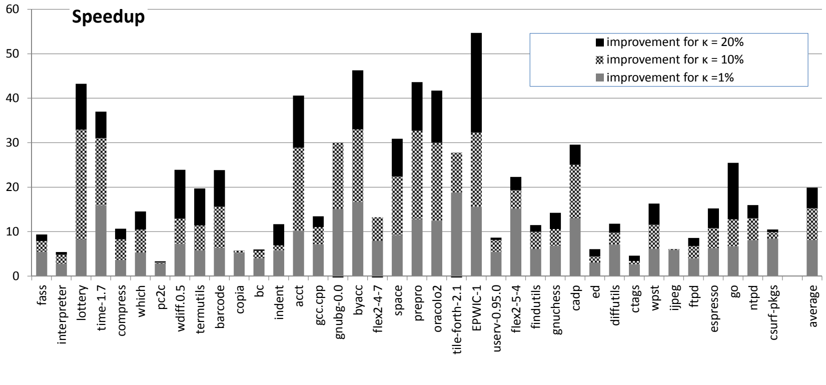


Fig. 14. Speedup from vertex exclusion. Each stacked bar shows the speedup when κ is 1% (solid gray) and then the increase in the speedup achieved by going to κ of 10% (gray checkerboard) and then 20% (black). For example, the average speedup for κ of 1%, 10%, and 20% are 8, 14, and 18, respectively. The programs are sorted based on program size.

the linchpin search dominates. This was confirmed using the gprof profiler where the functions involved in the search for the fringe dominate the execution time once depth exceeds five.

The lower chart shows a magnified view of the viable range of depths. The three curves, for the three values of κ, show very similar trends where increases in depth initially bring improvement that later wanes as search costs mount. For κ of 1% there is a clear spike at depth three. While less pronounced, for κ of 10% there is a spike at depth four. For κ of 20% the rise is more gradual with the best value occurring at depth five. Based on these results, when addressing research question RQ3.2, the search depths are fixed at three for $\kappa = 1\%$, four for $\kappa = 10\%$, and five for $\kappa = 20\%$.

5.4. RQ3.2: Performance Improvement

This section considers the impact of greater fringe search depth on vertex exclusion. Based on the depth study used to answer RQ3.1, this study fixes the search depth at three for $\kappa = 1\%$, four for $\kappa = 10\%$, and five for $\kappa = 20\%$. For each value of κ, the average impact is broken down in Figure 18. Over all 38 programs, the general pattern in the weighted average percentage of vertices in each of four categories: declarations, small slice, dual paths, and included, mirrors that seen in Figure 12.

Taking 10% of the area under the MSG as a cutoff for a cluster being a large cluster (the middle choice), the percentage of vertices removed from consideration increases from 87.7% using depth zero to 89.8% using depth four. Furthermore, increasing the cutoff to 20% does not notably improve the number of vertices excluded; thus, there is minimal benefit from the increase.

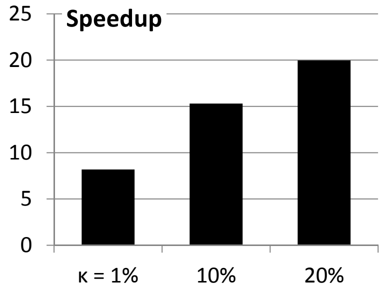


Fig. 15. Average Speedup for the three values of κ.

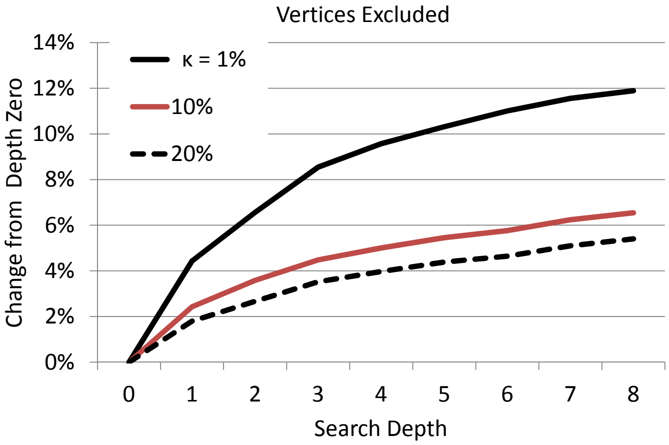
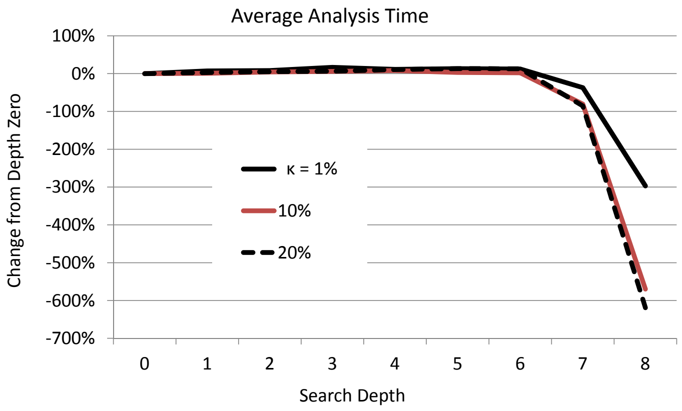


Fig. 16. Percentage change in excluded vertices for the three values of κ and depth ranging from 0 to 8.

Figure 19 shows the overall runtime speedup with the impact of greater search depth shown by the grey portion of each bar. The average speedup for κ of 1%, 10%, and 20% is 10x, 18x, and almost 25x, respectively. The speedups directly parallel the reduction in the number of vertices that must be considered. In more detail, Figure 20 shows the runtime speedup for each program



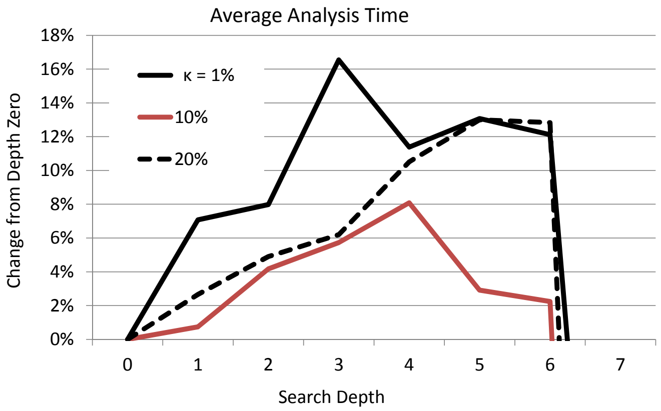
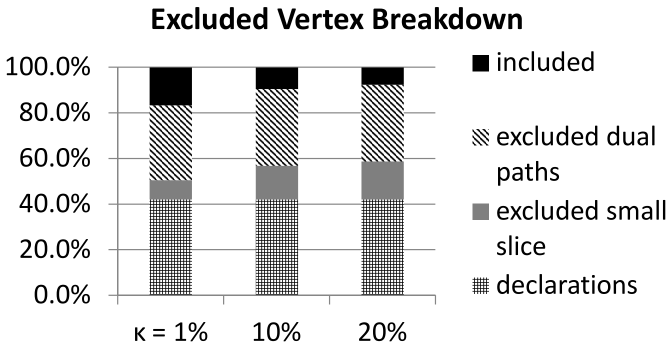


Fig. 17. The cumulative improvement in analysis time for three values of κ and depth ranging from 0 to 7. The lower chart shows a magnified view of the range 0 to 5.

with the averages shown on the far right. While most programs follow the general trend of an approximate 25% improvement, a few diverge from the pattern. For example, with flex2-5-4 the cost of the fringe search for $\kappa = 20$ far exceeds the benefit gained from excluding an additional 230 potential linchpin vertices. This leads to the negative speedup seen in Figure 20. At the other end of the spectrum, programs such as tile-forth show no improvement at depth zero when moving from $\kappa = 10\%$ to $\kappa = 20\%$. In contrast, with increased search depth the move from $\kappa = 10\%$ to



κ

partition $\backslash$ κ	1%	10%	20%
included	16.8%	10.2%	8.0%
excluded dual paths	34.0%	36.2%	37.2%
excluded small slice	9.4%	13.7%	15.0%
declarations	39.9%	39.9%	39.9%
total excluded	83.2%	89.8%	92.0%

Fig. 18. Weighted average percentage of vertices in four categories for the three values of κ. (The figures do not sum to 100% due to rounding errors.)

$\kappa = 20\%$ is accompanied by a notable speedup.

Finally, to better understand the per-program impact of *exclude*, Figure 21 shows the runtimes for the naïve, untuned, and tuned search for $\kappa = 10\%$. It also breaks out the time taken by the two simple checks (the declaration and small-slices tests) and the dual-path test. It is clear from this data that the MSG generation time dominates in all three algorithms. On average *exclude* takes 1% (untuned) and 2% (tuned) of the respective runtimes where the increase reflects the cost of the increased search depth. In the untuned and tuned algorithms the average time taken by the algorithm's declaration and small slice tests accounts for only 0.1% of the run time.

Statistically, again using a Friedman's paired test, all three versions provide a significant improvement in runtime over the naïve search ($p = 0.0005$ for κ of 1% and $p < 0.0001$ for the other two). Head to head, the runtimes for κ of 10% and 20% continue to be significantly less than that for κ of 1% ($p = 0.0005$ and $p < 0.0001$) respectively), and the runtime for κ of 10% is statistically higher than κ of 20% ($p = 0.003$). Again, the higher p value in the last test suggests that greater search depth was of more benefit going from κ of 1% to 10% than from 10% to 20%. In summary, the data and statistics show that on average the effect of tuning the fringe search depth is to improve the performance of the algorithm; however, for some programs the additional search costs exceed the savings attained by considering fewer potential linchpins.

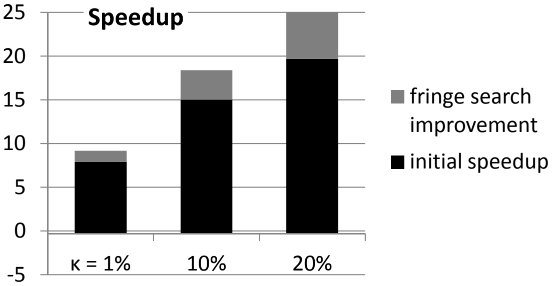


Fig. 19. Improvement in speedup from tuning. The grey tops shows the impact of the more aggressive search on the average speedup for the three values of κ.

5.5. Threats to Validity

This section concludes by considering threats to the external, internal, construct, and statistical 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 implication 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. They also include procedural C programs and object-oriented C++ programs. 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. Furthermore, all the programs studied are small to medium in size. It is possible that properties of larger programs differ from those studied. This difference would express itself through differences in the slices as the other parts of the algorithm are local to at most few procedures and larger programs tend to differ from smaller programs primarily in the number of procedures not the complexity of their procedures.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. The validation experiment directly tests that *exclude* finds only vertices that do not hold together large clusters. Other threats, such as maturation, are not

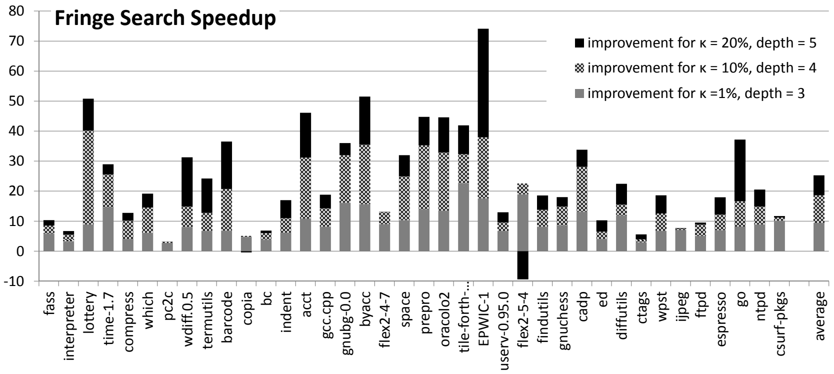


Fig. 20. Speedup from by vertex exclusion. Each bar shows the speedup when κ is 1% and then the increase in speedup achieved by going to κ of 10% and then to 20%. The programs are sorted based on program size.

a concern in the absence of human subjects.

Statistical conclusion validity considers the appropriateness of the statistical tests used. Friedman's paired test is a well known statistical test that is an appropriate substitute for a t-test when the latter's normality assumption is violated.

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; thus 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 [Grammatech Inc. 2002]) was used to mitigate this concern. In addition, the MSG generation tools have been extensively tested.

6. RELATED WORK

Section 2 described the historical development of work on the specific topic of large dependence clusters and their causes in source code. This section briefly reviews the wider context of work on dependence analysis and on dependence clusters at higher levels of abstraction.

Dependence analysis plays a key role in many forms of source code analysis and manipulation [Binkley 2007]. For example, it has been studied as a driver of program comprehension [Balmas 2002; Deng et al. 2001]. Dependence analysis has also been used to control the kinds of maintenance change that can be performed with minimal impact [Gallagher and Lyle 1991; Tonella 2003] as well as to measure the impact of such changes [Black 2001]. Finally, a recently proposed impact analysis framework [Acharya and Robinson 2011] reports that impact sets are often part of large

	(all times	untuned				tuned		
	in sec.)		D and		percent			percent
Program	naïve	total	SS	*exclude*	*exclude*	total	*exclude*	*exclude*
fass	399	50	0	1	3%	47	2	5%
interpreter	264	55	0	2	3%	48	3	6%
lottery	29	1	0	0	27%	1	0	36%
time-1.7	17	1	0	0	25%	1	0	43%
compress	22	3	0	1	19%	2	1	34%
which	58	6	0	1	9%	4	1	22%
pc2c	4,327	1,393	4	23	2%	1,379	32	2%
wdiff.0.5	251	19	0	1	5%	17	2	10%
termutils	1,099	96	1	3	4%	86	5	6%
barcode	3,636	233	2	9	4%	175	12	7%
copia	7,742	1,353	10	63	5%	1,525	117	8%
bc	15,217	2,666	8	65	2%	2,508	87	3%
indent	28,144	4,068	10	95	2%	2,551	181	7%
acct	5,164	179	1	8	5%	166	11	7%
gcc.cpp	49,786	4,510	13	90	2%	3,484	128	4%
gnubg-0.0	97,856	3,256	16	59	2%	3,055	134	4%
byacc	86,906	2,634	10	63	2%	2,449	67	3%
flex2-4-7	143,446	10,837	27	163	2%	10,999	223	2%
space	16,374	731	4	12	2%	656	22	3%
prepro	22,119	677	4	10	2%	627	18	3%
oracolo2	22,418	747	5	11	1%	681	18	3%
tile-forth	585,760	21,075	110	1,212	6%	18,104	2,285	13%
epwic	20,606	638	3	15	2%	543	21	4%
userv0	307,494	37,707	51	245	1%	32,028	461	1%
flex2-5-4	284,640	14,743	32	204	1%	12,625	316	3%
findutils	125,863	12,596	24	255	2%	9,138	777	8%
gnuchess	416,828	39,361	63	521	1%	27,931	478	2%
cadp	53,907	2,152	9	36	2%	1,917	40	2%
ed	601,486	135,616	149	1,203	1%	92,547	1,623	2%
diffutils	213,874	21,883	33	179	1%	13,771	778	6%
ctags	2,821,019	506,659	539	2,416	0%	437,066	6,250	1%
wpst	1,010,265	87,230	113	1,226	1%	80,236	1,701	2%
ijpeg	9,241,923	1,517,283	240	28,045	2%	1,219,897	55,124	5%
ftpd	1,036,271	153,065	227	1,167	1%	116,427	1,281	1%
espresso	4,265,666	394,800	464	3,365	1%	348,330	4,181	1%
go	8,711,660	681,586	1,035	7,307	1%	522,449	6,648	1%
ntpd	19,451,129	1,492,954	1,589	15,021	1%	1,303,025	21,146	2%
csurf-pkgs	29,914,749	3,055,701	2,756	21,855	1%	2,698,046	31,735	1%
average	217,094	28,537	39	256	1%	23,636	494	2%

Fig. 21. Runtimes for the naïve, untuned, and tuned algorithms for $\kappa = 10\%$. ("D and SS" abbreviates the algorithm's declaration and small slice tests, which take the same time for both the untuned and tuned variants. On average this account for 0.1% of the runtime.)

dependence clusters.

This paper is concerned with dependence clusters at the statement level, where the clusters bring together vertices of the program's dependence graph. However, other work has studied dependence and clustering of dependence at higher levels of abstraction, such as whole functions, modules, and files [Eisenbarth et al. 2003; Praditwong et al. 2011; Mitchell and Mancoridis 2006].

The study of clustering is not restricted to dependence nor to programs. There is also work on clustering of test cases [Yoo et al. 2009], but this work uses clustering to find commonality and reduce test effort. In such work, clustering is a choice and it has positive benefits. In the present paper, clustering is considered to be potentially harmful and it is not constructed through choice, but emerges from a program's dependence structure.

A specialized form of mutually dependent clusters, coherent dependence clusters was recently introduced [Islam et al. 2010b]. Such clusters extend dependence clusters to include not only internal dependence (each statements of a cluster must depend on all the other statements of the cluster) but also external dependence. Analysis of 16 open-source programs found that 15 of them had a coherent cluster that was over 5% of the program. Visualization of coherent clusters [Islam et al. 2010a] has also been used to locate structural problems within programs.

Looking beyond programs, dependence analysis and dependence clusters are also interesting to researchers studying other dependence networks (as construed in the broadest sense). The search for linchpins is akin to rarity measurements and anomaly detection used, for example, in social networking research [Madey et al. 2003]. 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.

Eberle and Holder [Eberle and Holder 2009] describe two related approaches. In the first, Lin and Chalupsky used *rarity measurements* to discover unusual links within a graph [Lin and Chlupsky 2003]. This approach assumes that a graph is built 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 be used to filter from consideration elements that *cannot* play the linchpin role. This is a topic for future investigation.

In the second approach, 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 [Rattigan and Jensen 2005]. 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 links represent coauthored papers, 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.

7. FUTURE WORK

Future work will consider the following: criteria that help separate refactorable clusters from unavoidable clusters, techniques for aiding a programmer break dependence clusters into smaller more manageable clusters, empirical assessment of dependence cluster's impact 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.

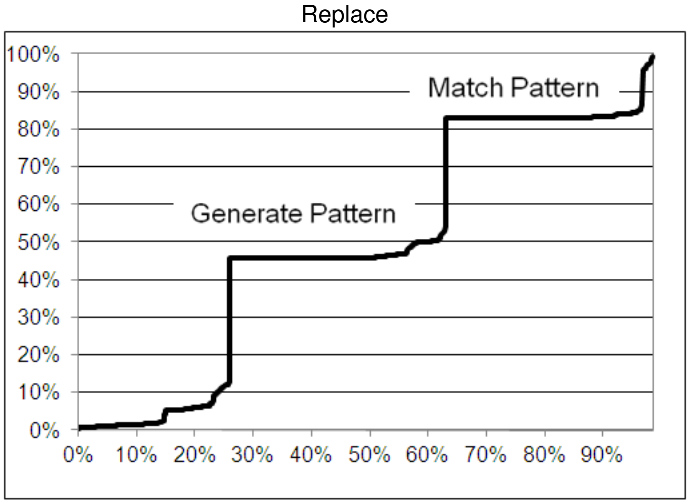


Fig. 22. The MSG for Replace showing two clusters. The containment relationship between these two clusters is unclear from the MSG.

Future work will also consider inter-cluster dependence relationships. For example, where the slices of vertices of one cluster include the vertices of another. Such relationships are an example of a feature not easily visualized using the MSG. For example, the MSG shown in Figure 22 includes two clusters (labeled Generate Pattern and Match Pattern). From the (unlabeled) MSG, it is not clear that the two clusters are related when in fact there is a containment relation between the two: every slice for an vertex in the Match Pattern cluster contains all the vertices of the Generate Pattern cluster. Better visualization and further study of such relationships will allow for 'cluster folding,' where related clusters can be merged together. This may lead to the identification of larger dependence structures, making a stronger case for identifying and removing linchpins.

Static analysis suffers from conservative dependence analysis collateral damage. This often leads to overly conservative, large, slices which can result in large clusters. Future work will consider how the size of dependence clusters varies with the size of static slices in programs. Intuitively, the size of a dependence cluster is bounded by the size of the slices taken with respect to its vertices. It will thus be interesting to see how a program's dependence clusters are affected by the use of different slicing techniques. For example, the use of dynamic slicing [Korel and Laski 1988] will help ignore debugging and error handling code resulting in smaller and more precise dependence clusters.

8. SUMMARY

This paper introduced a set of related theoretical results concerning the analysis of linchpins, which hold large dependence clusters together. The removal of a linchpin leads to the disappearance of source code dependence clusters. Using the theory we developed an algorithm for finding linchpins that is much faster than existing approaches. We provided empirical evidence from a large scale study of C and C++ code to support the claim that the assumptions made in the theoretical section of the paper are borne out in practice. Our empirical findings indicated that our algorithm can achieve orders of magnitude reductions in the analysis time. We also empirically studied the effects of tuning the algorithm's search depth and the effect that tuning has on algorithm performance.

REFERENCES

ACHARYA, M. AND ROBINSON, B. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering, (ICSE 2011)*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds. ACM, Waikiki, Honolulu, HI, USA, 746–755.

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

BALMAS, F. 2002. Using dependence graphs as a support to document programs. In 2^{nd} *IEEE International Workshop on Source Code Analysis and Manipulation* (Montreal, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 145–154.

BESZÉDES, Á., GERGELY, T., JÁSZ, J., TOTH, G., GYIMÓTHY, T., AND RAJLICH, V. 2007. Computation of static execute after relation with applications to software maintenance. In 23^{rd} *IEEE International Conference on Software Maintenance (ICSM 2007)* (Paris, France). IEEE Computer Society Press, Los Alamitos, California, USA, 295–304.

BINKLEY, D., GOLD, N., HARMAN, M., LI, Z., MAHDAVI, K., AND WEGENER, J. 2008. Dependence anti patterns. In 4^{th} *International ERCIM Workshop on Software Evolution and Evolvability (Evol'08)*. L'Aquila, Italy, 25–34.

BINKLEY, D. AND HARMAN, M. 2005. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance* (Budapest, Hungary, September 30th-October 1st 2005). IEEE Computer Society Press, Los Alamitos, California, USA, 177–186.

BINKLEY, D. AND HARMAN, M. 2009. Identifying 'linchpin vertices' that cause large dependence clusters. In *9th International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*. IEEE Computer Society, Edmonton, Canada.

BINKLEY, D., HARMAN, M., HASSOUN, Y., ISLAM, S., AND LI, Z. 2009. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software 83,* 1, 96–107.

BINKLEY, D. W. 1993. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems 3,* 1-4, 31–45.

BINKLEY, D. W. 2007. Source code analysis: A road map. In *Future of Software Engineering 2007*, L. Briand and A. Wolf, Eds. IEEE Computer Society Press, Los Alamitos, California, USA, 104–119.

BLACK, S., COUNSELL, S., HALL, T., AND BOWES, D. 2009. Fault analysis in OSS based on program slicing metrics. In *EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 3–10.

BLACK, S., COUNSELL, S., HALL, T., AND WERNICK, P. 2006. Using program slicing to identify faults in software. In *Beyond Program Slicing*, D. W. Binkley, M. Harman, and J. Krinke, Eds. Number 05451 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany.

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

DENG, Y., KOTHARI, S., AND NAMARA, Y. 2001. Program slice browser. In 9^{th} *IEEE International Workshop on Program Comprenhesion* (Toronto, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 50–59.

EBERLE, W. AND HOLDER, L. 2009. Graph-based approaches to insider threat detection. *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research Cyber Security and Information Intelligence Challenges and Strategies CSIIRW 09*.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2003. Locating features in source code. *IEEE Transactions on Software Engineering 29,* 3. Special issue on ICSM 2001.

FAHNDRICH, M., FOSTER, J. S., SU, Z., AND AIKEN, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montréal, Canada). Association for Computer Machinery, 85–96.

GALLAGHER, K. B. AND LYLE, J. R. 1991. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17,* 8, 751–761.

GRAMMATECH INC. 2002. The codesurfer slicing system.

HAJNAL, Á. AND FORGÁCS, I. 2011. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software Maintenance and Evolution: Research and Practice*. Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.533.

HARMAN, M., BINKLEY, D., GALLAGHER, K., GOLD, N., AND KRINKE, J. 2009. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems 32,* 1. Article 1.

HORWITZ, S., REPS, T., AND BINKLEY, D. W. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12,* 1, 26–61.

ISLAM, S., KRINKE, J., AND BINKLEY, D. 2010a. Dependence cluster visualization. In *SOFTVIS '10: Proceedings of the 5th international symposium on Software visualization*. ACM, Salt Lake City, Utah, USA.

ISLAM, S., KRINKE, J., BINKLEY, D., AND HARMAN, M. 2010b. Coherent dependence clusters. In *PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, Toronto, Canada.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Information Processing Letters 29,* 3, 155–163.

LIN, S. AND CHLUPSKY, H. 2003. Unsupervised link discovery in multi-relational data via rarity analysis. In *IEEE ICDM Conference on Data Mining*.

MADEY, G., FREEH, V., TYNAN, R., AND HOFFMAN, C. 2003. An analysis of open source software development using social network theory and agent-based modeling. In *Arrowhead Conference on Human Complex Systems*. Lake Arrowhead, CA, USA.

MITCHELL, B. S. AND MANCORIDIS, S. 2006. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering 32,* 3, 193–208.

PRADITWONG, K., HARMAN, M., AND YAO, X. 2011. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering 37,* 2, 264–282.

RATTIGAN, M. AND JENSEN, D. 2005. The case for anomalous link discovery. *ACM SIGKDD Expl. News 7,* 2.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*. San Francisco, CA, Jan. 23-25.

REPS, T. AND ROSAY, G. 1995. Precise interprocedural chopping. In *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, G. E. Kaiser, Ed. ACM Press, 41–52.

REPS, T. AND YANG, W. 1988. The semantics of program slicing. Tech. Rep. Technical Report 777, University of Wisconsin.

SAVERNIK, L. 2007. Entwicklung eines automatischen Verfahrens zur Auflösung statischer zyklischer Abhängigkeiten in Softwaresystemen (in German). In *Software Engineering 2007 - Beiträge zu den Workshops, Fachtagung des GI-Fachbereichs Softwaretechnik, 27.-30.3.2007 in Hamburg*, W.-G. Bleek, H. Schwentner, and H. Züllighoven, Eds. LNI Series, vol. 106. GI, 357–360.

SHARIR, M. AND PNUELI, A. 1981. *Two approaches to interprocedural data flow analysis*. Prentice-Hall, Englewood Cliffs, NJ.

SZEGEDI, A., GERGELY, T., BESZÉDES, Á., GYIMÓTHY, T., AND TÓTH, G. 2007. Verifying the concept of union slices on Java programs. In 11^{th} *European Conference on Software Maintenance and Reengineering (CSMR '07).* 233 – 242.

TONELLA, P. 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering 29,* 6, 495–509.

WEISER, M. 1984. Program slicing. *IEEE Transactions on Software Engineering 10,* 4, 352–357.

YONG, S. H., HORWITZ, S., AND REPS, T. May 1999. Pointer analysis for programs with structures and casting. *In Proceedings of the SIGPLAN 99 Conference on Programming Language Design and Implementation (Atlanta, GA)*, 91–103.

YOO, S., HARMAN, M., TONELLA, P., AND SUSI, A. 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ACM International Conference on Software Testing and Analysis (ISSTA 09)*. Chicago, Illinois, USA, 201–212.