Speeding up Slicing

Thomas Reps,[†] *Susan Horwitz,*[†]*, and Mooly Sagiv*[‡]
University of Copenhagen

Genevieve Rosay
University of Wisconsin-Madison

Program slicing is a fundamental operation for many software engineering tools. Currently, the most efficient algorithm for interprocedural slicing is one that uses a program representation called the system dependence graph. This paper defines a new algorithm for slicing with system dependence graphs that is asymptotically faster than the previous one. A preliminary experimental study indicates that the new algorithm is also significantly faster in practice, providing roughly a 6-fold speedup on examples of 348 to 757 lines.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques – *programmer workbench*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance – *enhancement, restructuring*; E.1 [**Data Structures**] *graphs*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: dynamic programming, dynamic transitive closure, flow-sensitive summary information, program debugging, program dependence graph, program slicing, realizable path

1. INTRODUCTION

Program slicing is a fundamental operation for many software engineering tools, including tools for program understanding, debugging, maintenance, testing, and integration [15,6,8,5,3,1]. Slicing was first defined by Mark Weiser [15], who gave algorithms for computing both intra- and inter-procedural slices. However, two aspects of Weiser's interprocedural-slicing algorithm can cause it to include "extra" program components in a slice:

1. A procedure call is treated like a multiple assignment statement "$v_1, v_2, \ldots, v_n := x_1, x_2, \ldots, x_m$", where the v_i are the set of variables that might be modified by the call, and the x_j are the set of variables that might be used by the call. Thus, the value of every v_i after the call is assumed to depend on the value of every x_j before the call. This may lead to an overly conservative slice (*i.e.*, one that includes extra components) as illustrated in Figure 1.
2. Whenever a procedure P is included in a slice, *all* calls to P (as well as the computations of the actual parameters) are included in the slice. An example in which this produces an overly conservative slice is given in Figure 2.

Interprocedural-slicing algorithms that solve the two problems illustrated above were given by Horwitz, Reps, and Binkley [7], and by Hwang, Du, and Chou [9]. Hwang, Du, and Chou give no analysis of their algorithm's complexity; however, as we show in Appendix A, in the worst case the time used by their algorithm is exponential in the size of the program. By contrast, the Horwitz-Reps-Binkley algorithm is a polynomial-time algorithm.

[†]On sabbatical leave from the University of Wisconsin, Madison, WI, USA.

[‡]On leave from IBM Israel, Haifa Research Laboratory.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants CCR-8958530 and CCR-9100424, by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937), by the Air Force Office of Scientific Research under grant AFOSR-91-0308, and by a grant from Xerox Corporate Research.

Authors' addresses: Datalogisk Institut, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, and Computer Sciences Department, University of Wisconsin, 1210 W. Dayton Street, Madison, WI, USA.
Electronic mail: {reps, horwitz, sagiv}@diku.dk; rosay@cs.wisc.edu.

example program	slice from "**output**(i)"	slice using Weiser's algorithm

procedure *Main*
  $sum := 0$
  $i := 1$
  while $i < 11$ **do**
  call $A(sum, i)$
  od
  output(sum)
  output(i)
end
 	
procedure $A(x, y)$
  $x := x + y$
  $y := y + 1$
return
	procedure *Main*
	
	  $i := 1$
	  while $i < 11$ **do**
	  call $A(i)$
	  od
	
	  output(i)
	end
	
	procedure $A(y)$
	
	  $y := y + 1$
	return
		procedure *Main*
		  $sum := 0$
		  $i := 1$
		  while $i < 11$ **do**
		  call $A(sum, i)$
		  od
		
		  output(i)
		end
		
		procedure $A(x, y)$
		
		  $y := y + 1$
		return

Figure 1. An example program, its slice with respect to "**output**(i)", and the slice computed using Weiser's algorithm.

example program	slice from "**output**(i)"	slice using Weiser's algorithm

procedure *Main*
  $sum := 0$
  $i := 1$
  while $i < 11$ **do**
  call $Add(sum, i)$
  call $Add(i, 1)$
  od
  output(sum)
  output(i)
end
 	
procedure $Add(x, y)$
  $x := x + y$
return
	procedure *Main*
	
	  $i := 1$
	  while $i < 11$ **do**
	
	  call $Add(i, 1)$
	  od
	
	  output(i)
	end
	
	procedure $Add(x, y)$
	  $x := x + y$
	return
		procedure *Main*
		  $sum := 0$
		  $i := 1$
		  while $i < 11$ **do**
		  call $Add(sum, i)$
		  call $Add(i, 1)$
		  od
		
		  output(i)
		end
		
		procedure $Add(x, y)$
		  $x := x + y$
		return

Figure 2. An example program, its slice with respect to "**output**(i)", and the slice computed using Weiser's algorithm.

The Horwitz-Reps-Binkley algorithm operates on a program representation called the system dependence graph (SDG). The algorithm involves two steps: first, the SDG is augmented with summary edges, which represent transitive dependences due to procedure calls; second, one or more slices are computed using the augmented SDG. The two steps of the algorithm (as well as the construction of the SDG) both require time polynomial in the size of the program. The cost of the first step—computing summary edges—dominates the cost of the second step.

In this paper we define a new algorithm for interprocedural slicing using SDGs that is asymptotically faster than the one given by Horwitz, Reps, and Binkley. In particular, we present an improved algorithm for computing summary edges. This not only leads to a faster interprocedural-slicing algorithm, but is also important for all other applications that use system dependence graphs augmented with summary edges [2,11,4].

The new algorithm's asymptotic complexity is discussed at the end of Section 3, and is compared to the previous algorithm's complexity in Section 4. Section 5 describes some experimental results that indicate how much

better the new slicing algorithm is than the old one: when implementations of the two algorithms were used to compute slices for three example programs (which ranged in size from 348 to 757 lines) the new algorithm exhibited roughly a 6-fold speedup.

2. BACKGROUND: INTERPROCEDURAL SLICING USING SYSTEM DEPENDENCE GRAPHS

2.1. System Dependence Graphs

System dependence graphs were defined in [7]. Due to space limitations we will not give a detailed definition here; the important ideas should be clear from the examples. A program's *system dependence graph* (SDG) is a collection of procedure dependence graphs (PDGs): one for each procedure. The vertices of a PDG represent the individual statements and predicates of the procedure. A call statement is represented by a call vertex and a collection of actual-in and actual-out vertices: there is an actual-in vertex for each actual parameter, and there is an actual-out vertex for each actual parameter that might be modified during the call. Similarly, procedure entry is represented by an entry vertex and a collection of formal-in and formal-out vertices. (Global variables are treated as "extra" parameters, and thus give rise to additional actual-in, actual-out, formal-in, and formal-out vertices.) The edges of a PDG represent the control and flow dependences among the procedure's statements and predicates.[1] The PDGs are connected together to form the SDG by *call* edges (which represent procedure calls, and run from a call vertex to an entry vertex) and by *parameter-in* and *parameter-out* edges (which represent parameter passing, and which run from an actual-in vertex to the corresponding formal-in vertex, and from a formal-out vertex to all corresponding actual-out vertices, respectively).

Example. Figure 3 shows the SDG for the program of Figure 2. □

2.2. Interprocedural Slicing

Ottenstein and Ottenstein showed that *intra*procedural slices can be obtained by solving a reachability problem on the PDG: to compute the slice with respect to PDG vertex v, find all PDG vertices from which there is a path to v along control and/or flow edges [13]. *Inter*procedural slices can also be obtained by solving a reachability problem on the SDG; however, the slices obtained using this approach will include the same "extra" components as illustrated in column 3 of Figure 2. This is because not all paths in the SDG correspond to possible execution paths. For example, there is a path in the SDG shown in Figure 3 from the vertex of procedure *Main* labeled "$sum := 0$" to the vertex of *Main* labeled "**output**(i)." However, this path corresponds to an "execution" in which procedure *Add* is called from the first call site in *Main*, but returns to the second call site in *Main*, which is not a legal call/return sequence. The final value of i in *Main* is independent of the value of *sum*, and so the vertex labeled "$sum := 0$" should not be included in the slice with respect to the vertex labeled "**output**(i)".

Instead of considering all paths in the SDG, the computation of a slice must consider only *realizable* paths: paths that reflect the fact that when a procedure call finishes, execution returns to the site of the most recently

---

[1] As defined in [7], procedure dependence graphs include four kinds of dependence edges: control, loop-independent flow, loop-carried flow, and def-order. However, for slicing the distinction between loop-independent and loop-carried flow edges is irrelevant, and def-order edges are not used. Therefore, in this paper we assume that PDGs include only control edges and a single kind of flow edge.

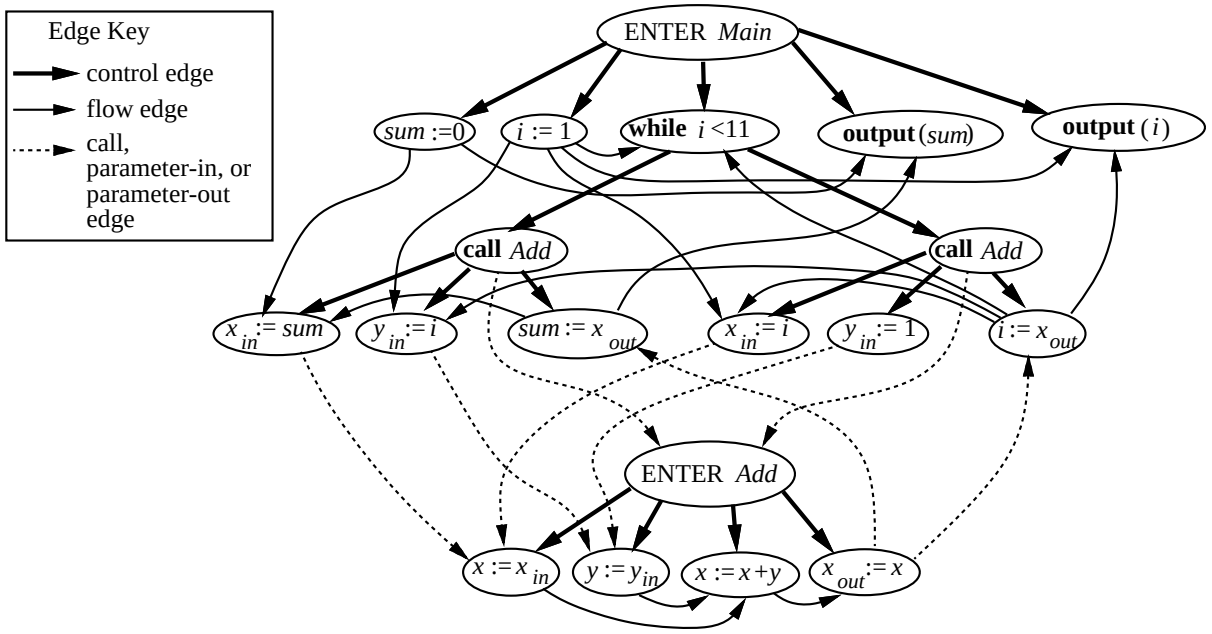


Figure 3. The SDG for the program of Figure 2.

executed call.[2]

Definition (realizable paths). Let each call 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 in the path is a string in the language of balanced parentheses generated from nonterminal *matched* by the following context-free grammar:

$$\begin{array}{lll} matched & \rightarrow & matched\ (_i\ matched\ )_i \qquad \text{for } 1 \le i \le k \\ & | & \varepsilon \end{array}$$

A path in G is a *realizable* path iff the sequence of symbols labeling the parameter-in, parameter-out, and call edges in the path is a string in the language generated from nonterminal *realizable* by the following context-free grammar (where *matched* is as defined above):

$$\begin{array}{lll} realizable & \rightarrow & realizable\ (_i\ matched \qquad \text{for } 1 \le i \le k \\ & | & matched \end{array}$$

□

Example. In Figure 3, the path

$$sum := 0 \rightarrow x_{in} := sum \rightarrow x := x_{in} \rightarrow x := x + y \rightarrow x_{out} := x \rightarrow sum := x_{out} \rightarrow \textbf{output}(sum)$$

is a (same-level) realizable path, while the path

[2] A similar goal of considering only paths that correspond to legal call/return sequences arises in the context of interprocedural dataflow analysis [14,12]. Several different terms have been used for these paths, including *valid paths*, *feasible paths*, and *realizable paths*.

$$sum := 0 \rightarrow x_{in} := sum \rightarrow x := x_{in} \rightarrow x := x + y \rightarrow x_{out} := x \rightarrow i := x_{out} \rightarrow \textbf{output}(i)$$

is not. □

An interprocedural-slicing algorithm is precise up to realizable paths if, for a given vertex v, it determines the set of vertices that lie on some realizable path from the entry vertex of the main procedure to v. To achieve this precision, the Horwitz-Reps-Binkley algorithm first augments the SDG with *summary edges*. A summary edge is added from actual-in vertex v (representing the value of actual parameter x before the call) to actual-out vertex w (representing the value of actual parameter y after the call) whenever there is a same-level realizable path from v to w. The summary edge represents the fact that the value of y after the call might depend on the value of x before the call. Note that a summary edge cannot be computed simply by determining whether there is a path in the SDG from v to w (*e.g.*, by taking the transitive closure of the SDG's edges). That approach would be imprecise for the same reason that transitive closure leads to imprecise interprocedural slicing, namely that not all paths in the SDG are realizable paths.

After adding summary edges, the Horwitz-Reps-Binkley slicing algorithm uses two passes over the augmented SDG; each pass traverses only certain kinds of edges. To slice an SDG with respect to vertex v, the traversal in Pass 1 starts from v and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but *not* along parameter-out edges. The traversal in Pass 2 starts from all actual-out vertices reached in Pass 1 and goes backwards along flow edges, control edges, summary edges, and parameter-out edges, but *not* along call or parameter-in edges. The result of an interprocedural slice consists of the set of vertices encountered during Pass 1 and Pass 2, and the edges induced by those vertices.[3]

Example. Figure 4 gives the SDG of Figure 3 augmented with summary edges, and shows the vertices and edges traversed during the two passes when slicing with respect to the vertex labeled "**output**(i)." □

3. AN IMPROVED ALGORITHM FOR COMPUTING SUMMARY EDGES

This section contains the main result of the paper: a new algorithm for computing summary edges that is asymptotically faster than the one defined by Horwitz, Reps, and Binkley. (We will henceforth refer to the latter as the *HRB-summary algorithm*.)

The new algorithm for computing summary edges is given in Figure 5 as function ComputeSummaryEdges. (Function *Proc* returns the procedure that contains the given SDG vertex; function *Callers* returns the set of procedures that call the given one; function *CorrespondingActualIn* (and *CorrespondingActualOut*) returns the actual-in (or actual-out) vertex associated with the given call site that corresponds to the given formal-in (or formal-out) vertex.) Figure 6 illustrates schematically the key steps of the algorithm. The basic idea is to find, for every procedure P, all same-level realizable paths that end at one of P's formal-out vertices. Those paths that start from one of P's formal-in vertices induce summary edges between the corresponding actual-in and actual-out vertices at all call sites that represent calls to P. (For example, if the algorithm were applied to the SDG shown in Figure 3, a path would be found from the formal-in vertex of procedure *Add* labeled "$x := x_{in}$" to the formal-out vertex labeled "$x_{out} := x$". This path would induce the summary edges from "$x_{in} := sum$" to "$sum := x_{out}$", and from "$x_{in} := i$" to "$i := x_{out}$", in *Main*, as shown in Figure 4.)

---

[3] The augmented SDG can also be used to compute a *forward* (interprocedural) slice using two edge-traversal passes, where each pass traverses only certain kinds of edges; however, in a forward slice edges are traversed from source to target. The first pass of a forward slice ignores parameter-in and call edges; the second pass ignores parameter-out edges.

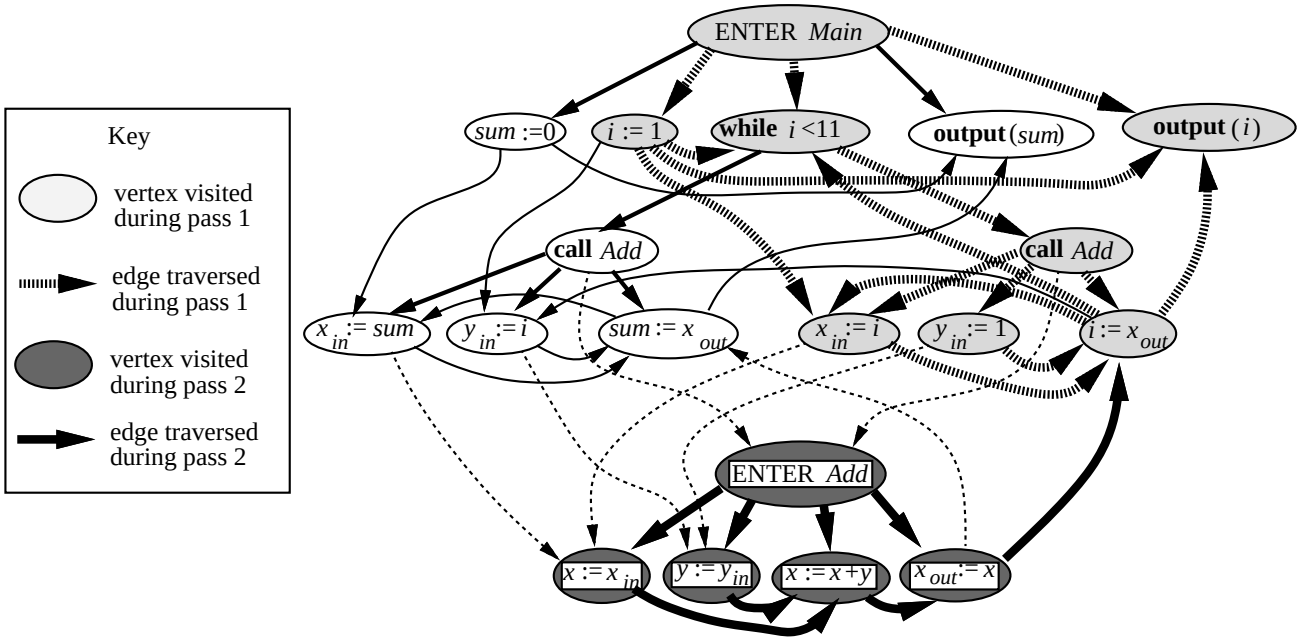


Figure 4. The SDG of Figure 3, augmented with summary edges and sliced with respect to "**output**(*i*)".

In the algorithm, same-level realizable paths are represented by "path edges", the edges that are inserted into the set called PathEdge. The algorithm starts by "asserting" that there is a same-level realizable path from every formal-out vertex to itself; these path edges are inserted into PathEdge, and also placed on the worklist. Then the algorithm finds new path edges by repeatedly choosing an edge from the worklist and extending (backwards) the path that it represents as appropriate depending on the type of the source vertex (this is illustrated in Figure 6). When a path edge is processed whose source is a formal-in vertex, the corresponding summary edges are inserted into the SummaryEdge set (lines [16]-[19]). These new summary edges may in turn induce new path edges: if there is a summary edge $x \longrightarrow y$, then there is a same-level realizable path $x \longrightarrow^{+} a$ for every formal-out vertex a such that there is a same-level realizable path $y \longrightarrow^{+} a$. Therefore, procedure Propagate is called with all appropriate $x \longrightarrow a$ edges (lines [20] - [22]).

The cost of the algorithm can be expressed in terms of the following parameters:

P	The number of procedures in the program.
$Sites_p$	The number of call sites in procedure p.
Sites	The maximum number of call sites in any procedure.
TotalSites	The total number of call sites in the program. (This is bounded by $P \times Sites$.)
E	The maximum number of control and flow edges in any procedure's PDG.
Params	The maximum number of formal-in vertices in any procedure's PDG.

The algorithm finds all same-level realizable paths that end at a formal-out vertex w. A new path $x \longrightarrow^{+} w$ is found by extending (backwards) a previously discovered path $v \longrightarrow^{*} w$ (taken from the worklist) along the edge $x \longrightarrow v$. Because vertex x can have out-degree greater than one, the same path can be discovered more than once (but it will only be put on the worklist once, due to the test in *Propagate*). In the worst case, the algorithm will "extend a path" along every PDG edge (lines [27] - [29]) and every summary edge (lines [11] - [13] and [20] -

```
     function ComputeSummaryEdges(G: SDG) returns set of edges
     declare PathEdge, SummaryEdge, WorkList: set of edges

     procedure Propagate(e: edge)
     begin
[1]    if e ∉ PathEdge then insert e into PathEdge; insert e into WorkList fi
     end

     begin
[2]    PathEdge := ∅; SummaryEdge := ∅; WorkList := ∅
[3]    for each w ∈ FormalOutVertices(G)
[4]      insert (w ⟶ w) into PathEdge
[5]      insert (w ⟶ w) into WorkList
[6]    od
[7]    while WorkList ≠ ∅ do
[8]      select and remove an edge v ⟶ w from WorkList
[9]      switch v
[10]       case v ∈ ActualOutVertices(G) :
[11]         for each x such that x ⟶ v ∈ (SummaryEdge ∪ ControlEdges(G)) do
[12]           Propagate(x ⟶ w)
[13]         od
[14]       end case
[15]       case v ∈ FormalInVertices(G) :
[16]         for each c ∈ Callers(Proc(w)) do
[17]           let x = CorrespondingActualIn(c, v)
[18]               y = CorrespondingActualOut(c, w) in
[19]             insert x ⟶ y into SummaryEdge
[20]             for each a such that y ⟶ a ∈ PathEdge do
[21]               Propagate(x ⟶ a)
[22]             od
[23]           end let
[24]         od
[25]       end case
[26]       default :
[27]         for each x such that x ⟶ v ∈ (FlowEdges(G) ∪ ControlEdges(G)) do
[28]           Propagate(x ⟶ w)
[29]         od
[30]       end case
[31]     end switch
[32]   od
[33]   return(SummaryEdge)
     end
```

Figure 5. Function ComputeSummaryEdges computes and returns the set of summary edges for the given system dependence graph G (see also Figure 6).

[22]) once for each formal-out vertex. Thus, the cost of computing summary edges for a single procedure is equal to the number of formal-out vertices (bounded by $Params$) times the number of PDG and summary edges in that procedure. In the worst case there is a summary edge from every actual-in vertex to every actual-out vertex associated with the same call site. Therefore, the number of summary edges in procedure p is bounded by $O(Sites_p \times Params^2)$, and the cost of computing summary edges for one procedure is bounded by $O(Params \times (E + (Sites_p \times Params^2)))$, which is equal to $O((Params \times E) + (Sites_p \times Params^3))$. Summing over all procedures in the program, the total cost of the algorithm is bounded by $O((P \times Params \times E) + (TotalSites \times Params^3))$.

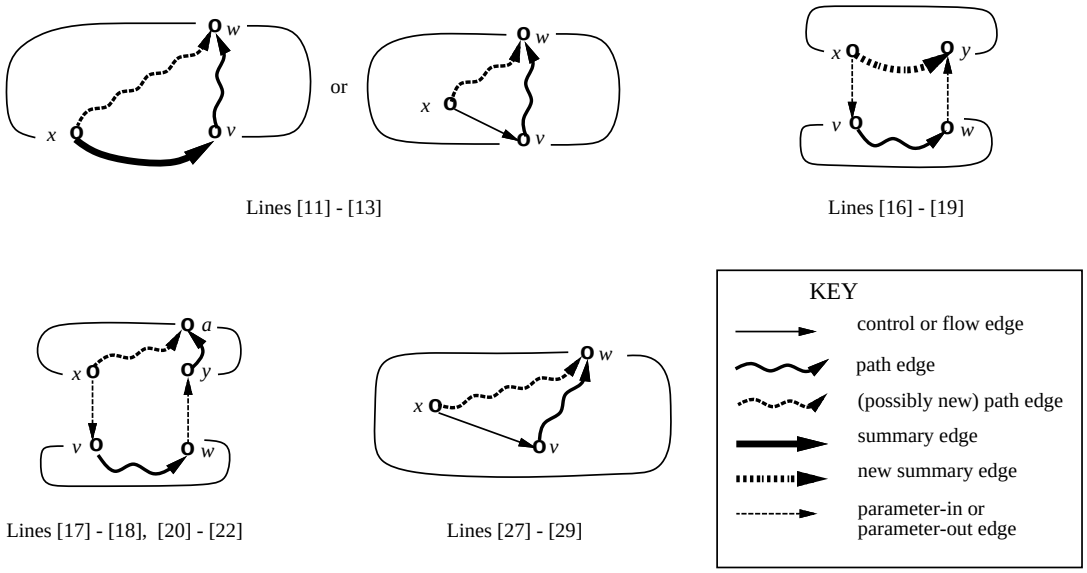


Figure 6. The above four diagrams show how the algorithm of Figure 5 extends same-level realizable paths, and installs summary edges.

4. COMPARISON WITH PREVIOUS WORK

The cost of interprocedural slicing using the algorithm of Horwitz, Reps, and Binkley is dominated by the cost of computing summary edges via the HRB-summary algorithm (see [7]):

$$O((TotalSites \times E \times Params) + (TotalSites \times Sites^2 \times Params^4)).$$

The main result of this paper is a new algorithm for computing summary edges whose cost is bounded by

$$O((P \times E \times Params) + (TotalSites \times Params^3)).$$

Under the reasonable assumption that the total number of call sites in a program is much greater than the number of procedures, each term of the cost of the new algorithm is asymptotically smaller than the corresponding term of the cost of the HRB-summary algorithm. Furthermore, because there is a family of examples on which the HRB-summary algorithm actually performs $\Omega((TotalSites \times E \times Params) + (TotalSites \times Sites^2 \times Params^4))$ steps, the new algorithm is asymptotically faster.

There are two main differences in the approaches taken by the two algorithms that lead to the differences in their costs:

1. The HRB-summary algorithm first creates a "compressed" form of the SDG that contains only formal-in, formal-out, actual-in, and actual-out vertices. The edges of the compressed graph represent (intraprocedural) paths in the original graph. The cost of compressing the SDG is $O(TotalSites \times E \times Params)$, the first term in the cost given above. The new algorithm uses the uncompressed SDG, so there is no compression cost.
2. After compressing the SDG, the HRB-summary algorithm repeatedly finds and installs summary edges, then closes the edge set of the PDG. This series of "install and close" steps is similar to the "extend a path" step that is repeated by the new algorithm. The difference is that the "close" step of the HRB-summary algorithm essentially replaces a 3-part path of the form "path:edge:path" with a single path edge, while the new

algorithm replaces a 2-part path of the form "edge:path" with a single path edge. The latter approach is a second reason for the superiority of the new algorithm. The total cost of the series of "install and close" steps performed by the HRB-summary algorithm is $O(TotalSites \times Sites^2 \times Params^4)$, the second term in the cost given above. This term is likely to be the dominant term in practice, and it is worse (by a factor of $Sites^2 \times Params$) than the second term in the new algorithm's cost.

To summarize: Both the cost of the HRB-summary algorithm and the cost of the new algorithm contain two terms. In the case of the former, the first term represents the cost of compression, and the second term represents the cost of finding summary edges using the compressed graph. In the case of the latter, both terms represent the cost of finding summary edges using the uncompressed graph. The cost of the new algorithm is asymptotically better than the cost of the HRB-summary algorithm.

5. EXPERIMENTAL RESULTS

This section describes the results of a preliminary performance study we carried out to measure how much faster interprocedural slicing is when function ComputeSummaryEdges is used in place of the HRB-summary algorithm. The slicing algorithms were implemented in C and tested on a Sun SPARCstation 10 Model 30 with 32 MB of RAM. Tests were carried out for three example programs (written in a small language that includes scalar variables, array variables, assignment statements, conditional statements, output statements, while loops, for loops, and procedures with value-result parameter passing): *recdes* is a recursive-descent parser for lists of assignment statements; *calculator* is a simple arithmetic calculator; and *format* is a text-formatting program taken from Kernighan and Plauger's book on software tools [10]. The following table gives some statistics about the SDGs of the three test programs:

Example	Lines of source code	SDG statistics						
		Vertices	Control and flow edges	*P*	*Sites*	*TotalSites*	*E*	*Params*
recdes	348	838	1465	15	13	60	255	8
calculator	433	841	1443	24	26	70	409	12
format	757	1844	3276	53	20	108	597	23

The comparison in Section 4 of the asymptotic worst-case running time of the HRB-summary algorithm with that of the new algorithm suggests that the new algorithm should lead to a significantly better slicing algorithm. However, formulas for asymptotic worst-case running time may not be good predictors of actual performance. For example, the formula for the running time of ComputeSummaryEdges was derived under the (worst-case) assumptions that there is a summary edge from every actual-in vertex to every actual-out vertex associated with the same call site, and that every call site has the same number of actual-in and actual-out vertices—both of which are bounded by *Params*. This yields $O(TotalSites \times Params^2)$ as the bound on the total number of summary edges. As shown in the following table, this overestimates the actual number of summary edges by one to two orders of magnitude:

Example	$TotalSites \times Params^2$	Actual number of summary edges
recdes	3840	157
calculator	10080	227
format	57132	413

Thus, although asymptotic worst-case analysis may be helpful in guiding algorithm design, tests are clearly needed to determine how well a slicing algorithm performs in practice.

For our study, we implemented three different slicing algorithms: (A) the Horwitz-Reps-Binkley slicing algorithm, (B) the slicing algorithm with the improved method for computing summary edges from Section 3, and (C) an algorithm that is essentially the "dual" of Algorithm B. Algorithm C is just like Algorithm B, except that the computation of summary edges involves finding all same-level realizable paths *from* formal-in vertices (rather than *to* formal-out vertices), and paths are extended forwards rather than backwards.

The following table gives statistics about the performance of the three algorithms for a representative slice of each of the three programs. In each case, the reported running time is the average of five executions. (The quantity "Time to slice" is "user cpu-time + system cpu-time".)

Example	Vertices in slice		Algorithm A	Algorithm B		Algorithm C	
			HRB algorithm	Summary edges computed by the algorithm of Section 3		Summary edges computed by the dual of the algorithm of Section 3	
	Number	Percent of total	Time to slice (seconds)	Time to slice (seconds)	Speedup (over HRB)	Time to slice (seconds)	Speedup (over HRB)
recdes	413	49%	2.08 + 0.04	0.35 + 0.04	5.4	0.39 + 0.05	4.8
calculator	484	58%	3.06 + 0.05	0.46 + 0.03	6.3	0.45 + 0.03	6.5
format	1327	72%	6.64 + 0.12	0.98 + 0.12	6.1	1.09 + 0.16	5.4

The time for the final step of computing slices—the two-pass traversal of the augmented SDG—is not shown as a separate entry in the table; this step is a relatively small portion of the time to slice: .03-.04 seconds (of total cpu-time) for both *recdes* and *calculator*; .20-.23 seconds for *format*.

As shown in columns 6 and 8 of the above table, Algorithms B and C are clearly superior to Algorithm A, exhibiting 4.8-fold to 6.5-fold speedup. Algorithm B appears to be marginally better than Algorithm C. We believe that this is because procedures have fewer formal-out vertices than formal-in vertices.

Because the bound derived for the series of "install-and-close" steps of Algorithms B and C is better than the bound for the Horwitz-Reps-Binkley algorithm by a factor of $Sites^2 \times Params$, the speedup factor may be greater on larger programs. As a preliminary test of this hypothesis, we gathered some statistics on versions of the above programs in which the number of parameters was artificially inflated (by adding additional global variables). On these examples, Algorithm C exhibited 10-fold speedup over the Horwitz-Reps-Binkley algorithm, and Algorithm B exhibited 13-fold to 23-fold speedup.

In summary: the conclusion that the algorithm presented in this paper is significantly better than the Horwitz-Reps-Binkley interprocedural-slicing algorithm is supported both by comparison of asymptotic worst-case running times (see Section 4) and preliminary experimental results.

APPENDIX A: Demonstration that the Algorithm of Hwang, Du, and Chou is Exponential

The Hwang-Du-Chou algorithm constructs a *sequence* of slices of the program—where each slice in the sequence essentially permits one additional level of recursion—until a fixed point is reached (*i.e.*, until no further elements are included in a slice). In essence, to compute a slice with respect to a point in procedure P, it is as if the algorithm performs the following sequence of steps:

1. Replace each call in procedure P with the body of the called procedure.

2. Compute the slice using the new version of P (and assume that there are no flow dependences across unexpanded calls).
3. Repeat steps 1 and 2 until no new vertices are included in the slice. (For the purposes of determining whether a new vertex is included in the slice, each vertex instance in the expanded program is identified with its "originating vertex" in the original, multi-procedure program.)

In fact, no actual in-line expansions are performed; instead they are simulated using a stack. On the k^{th} slice of the sequence, there is a bound of k on the depth of the stack. Because the stack is used to keep track of the calling context of a called procedure, only realizable paths are considered.

In this appendix, we present a family of examples on which the Hwang-Du-Chou algorithm takes exponential time. In order to simplify the presentation of this family of programs, we will streamline the diagrams of the SDGs we use by including only vertices related to procedure calls (enter, formal-in, formal-out, call, actual-in, and actual-out vertices) and the intraprocedural transitive dependences among them. (This streamlining does not affect our argument, and showing complete SDGs would make our diagrams unreadable.)

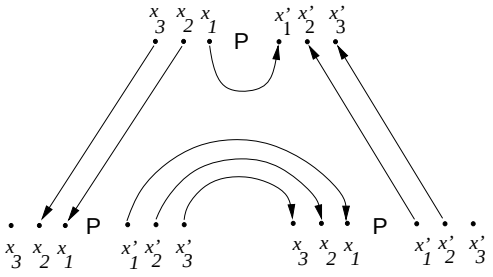
Theorem. *There is a family of programs on which the Hwang-Du-Chou algorithm uses time exponential in the size of the program.*

Proof. We construct a family of programs P^k that grows linearly in size with k but causes the Hwang-Du-Chou algorithm to use time exponential in the size of k (*i.e.*, the algorithm's running time is $\Omega(2^k)$).

A given program P^k in the family consists of just a single recursive procedure (also named P^k), defined as follows:

procedure $P^k(x_1, x_2, \cdots, x_{k-1}, x_k)$
  $t := 0$
  call $P^k(x_2, \cdots, x_{k-1}, x_k, t)$
  call $P^k(x_2, \cdots, x_{k-1}, x_k, t)$
  $x_1 := x_1 + 1$
end

To present the idea behind the construction, we first discuss the case of P^3. The SDG for program P^3 can be depicted as shown below. (We use the labels x_i and x_i', for $1 \le i \le 3$, to denote corresponding formal-in, formal-out, actual-in, and actual-out vertices. To enhance readability, formal-in and actual-in vertices are shown ordered right-to-left ($x_3\ x_2\ x_1$) rather than left-to-right ($x_1\ x_2\ x_3$).)



Now consider a slice of program P^3 with respect to the formal-out vertex for parameter x_3. To compute this slice, the Hwang-Du-Chou method performs actions that are equivalent to carrying out a traversal of an exponentially long path in a complete binary tree of height 3. The path traversed is shown in bold in Figure 7.

If we examine the tree of Figure 7 more closely, it becomes apparent that the original slicing problem spawns two additional slicing problems of very similar form. These two subsidiary problems involve performing slices

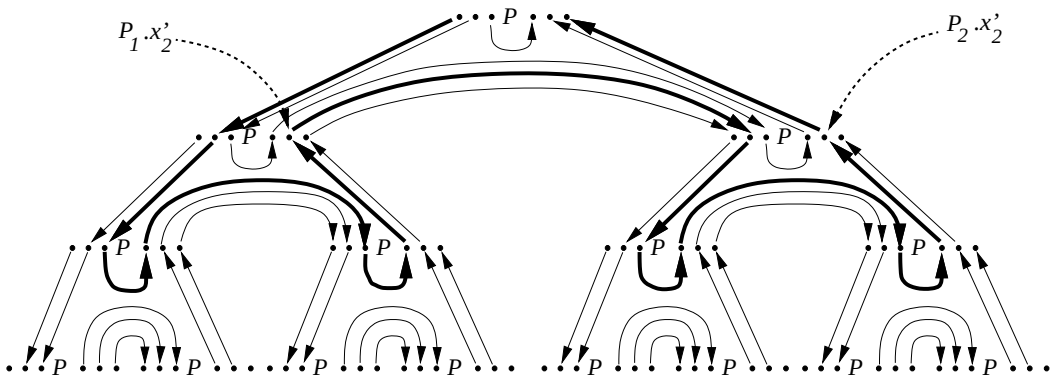


Figure 7. To compute the same-level slice with respect to $P.\,x_3'$, the Hwang-Du-Chou algorithm traverses the path highlighted in bold.

of the program with respect to $P_1.\,x_2'$ and $P_2.\,x_2'$, where P_1 and P_2 are the two children of the root of the tree. Each of these subsidiary slicing problems is equivalent to taking a slice with respect to the formal-out vertex $P^2.\,x_2'$ in program P^2.

In general, the Hwang-Du-Chou algorithm takes exponential time on the family of programs P^k. To perform a slice with respect to formal-out vertex $P^k.\,x_k'$, the algorithm performs actions that are equivalent to traversing an exponentially long path (*i.e.*, a path of length $\Omega(2^k)$) in a complete binary tree of height k. To perform the slice with respect to formal-out vertex $P^k.\,x_k'$, the algorithm spawns two subsidiary slicing problems that are equivalent to performing slices with respect to formal-out vertex $P^{k-1}.\,x_{k-1}'$ in program P^{k-1}. (In addition to the two subsidiary slices, three additional edges are traversed.) Thus, the time complexity of the Hwang-Du-Chou algorithm is described by the following recurrence relation:

$$T(k) = 2T(k-1) + 3$$
$$T(1) = 1$$

Therefore, $T(k) = 2^{k+1} - 3$. □

Acknowledgement

The *recdes* and *calculator* programs were supplied by Tommy Hoffner (Linköping University).

References

1. Bates, S. and Horwitz, S., "Incremental program testing using program dependence graphs," pp. 384-396 in *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages,* (Charleston, SC, January 10-13, 1993), ACM, New York, NY (1993).
2. Binkley, D., "Multi-procedure program integration," Ph.D. Thesis and Technical Report 1038, Department of Computer Sciences, University of Wisconsin, Madison, WI (August 1991).
3. Binkley, D., "Using semantic differencing to reduce the cost of regression testing," *Proceedings of the 1992 Conference on Software Maintenance* (Orlando, Flori da), pp. 41-50 (November 9-12, 1992).
4. Binkley, D., "Interprocedural constant propagation using dependence graphs and a data-flow model," pp. 374-388 in *Proceedings of the Fifth International Conference on Compiler Construction*, (Edinburgh, U.K., April 7-9, 1994)*, Lecture Notes in Computer Science,* Vol. 786, ed. P.A. Fritzson, Springer-Verlag, New York, NY (1994).
5. Gallagher, K.B. and Lyle, J.R., "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering* **17**(8) pp. 751-761 (August 1991).
6. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems* **11**(3) pp. 345-387 (July 1989).
7. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).
8. Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 1990), pp. 234-246 (June 1990).
9. Hwang, J.C., Du, M.W., and Chou, C.R., "Finding program slices for recursive procedures," in *Proceedings of IEEE COMPSAC 88,* (Chicago, IL, Oct. 3-7, 1988), IEEE Computer Society, Washington, DC (1988).
10. Kernighan, B. and Plauger, P., *Software Tools in Pascal,* Addison-Wesley, Reading, MA (1981).
11. Lakhotia, A., "Constructing call multigraphs using dependence graphs," pp. 273-284 in *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages,*(Charleston, SC, Jan. 11-13, 1993), ACM, New York, NY (1993).
12. Landi, W. and Ryder, B., "Pointer-induced aliasing: A problem classification," pp. 93-103 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages,* (Orlando, FL, January 21-23, 1991), ACM, New York, NY (1991).
13. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).
14. Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones, Prentice-Hall, Englewood Cliffs, NJ (1981).
15. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).