February 25, 1993

Improved interprocedural slicing algorithm

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 231-6766, -5791 (Fax)
arun@cacs.usl.edu

Available as CACS TR-92-5-8

Abstract

Horwitz, Reps, and Binkley (TOPLAS, 90) present an algorithm for interprocedural program slicing using a system dependence graph (SDG) representation of programs. In order to identify the set of statements in a slice their algorithm makes two traversals over the SDG; effectively traversing some edges twice. This paper presents a one pass algorithm which traverses each edge in the slice at most once. In scenarios requiring *on-line* union of interprocedural slices, the algorithm provides significant improvement by permitting the construction of union incrementally.

1 Introduction

A slice of a program with respect to a program point *p* consists of all statements of the program that might affect the behavior of the program observed at *p*; the program point *p* is said to be the *slicing criterion*[*]. Horwitz, Reps, and Binkley [1] present an algorithm (henceforth referred to as HRB algorithm) for computing slices that cross boundaries formed by procedure calls. They introduced *system dependence graph* (SDG), a graph with typed edges and nodes encoding the data, control, and call dependencies in a program. In order to slice a program it is first converted into its SDG representation. The slice at a particular vertex (corresponding to a program point) using their algorithm is obtained in two passes. Both passes perform the transitive closure of vertices that can reach a given set of vertices in the SDG; the passes differ in the type of dependences they use to perform the closure.

This paper presents an algorithm (henceforth referred to as AL algorithm) equivalent to HRB algorithm. The AL algorithm uses the SDG representation for slicing and may be implemented to perform only one traversal of each edge in the slice. This improves upon the HRB algorithm which in the worst case traverses an edge twice. The AL traversal algorithm therefore is optimal.

The SDG representation of a program may be used to perform interprocedural analyses other than slicing. Most of the analyses algorithms would require forward or backward traversal of dependences represented in the SDG. The traversal algorithm used for interprocedural forward and backward slicing could be used as the backbone of these algorithms. One such application reported in [3] is the problem of constructing call multigraph of a program with procedure valued variables. The algorithm reported in [3]

[*] This definition is slightly different from the original definition of program slice introduced by Weiser [10].

combines Horwitz et. al.'s forward slicing algorithm's traversal mechanism with Wegman and Zadeck's constant propagation algorithm [9] to construct the call multigraph. In order to do so the algorithm requires several traversals of the SDG. Even an improvement in the constant factor provided by the AL algorithm could lead to a substantial improvement in the running time of this and other algorithms requiring such traversals.

Our algorithm provides substantial improvement in situations requiring *on-line* slicing of the same program. Consider, a debugging scenario where a programmer performs a slice on a statement, finds that it is inadequate, and would like the union of this slice with that over another statement. Using the HRB algorithm, the second request for slicing will require computing a fresh slice on the union of the two statements. However, the AL algorithm may be used to construct the slice incrementally starting from the slice for the first statement. The HRB algorithm uses the two passes to correctly account for the calling context of a procedure call. It maintains a two valued tag to distinguish statements in the slice from those not in it. The AL algorithm maintains a three valued tag that helps in distinguishing the calling context as well as identifying the vertices in the slice. Since the calling context of the previous slice is available in the tags, the AL algorithm may be used to incrementally build on it. This is not possible in the HRB algorithm because of the lack of the context information.

For the sake of brevity this paper does not contain a detailed discussion of system dependence graph (SDG); the reader is referred to [1] for that. The salient features of the graph are sketched in Section 2. Section 3 outlines HRB forward slicing algorithm and Section 4 the AL slicing algorithm. The proof of equivalence of the two algorithms is given in Section 5. Section 6 analyses the complexity of the AL algorithm and presents empirical results comparing the two algorithms. Section 7 contain our conclusions.

It may be emphasized that this paper only gives the AL algorithm for backward slicing. The concepts presented however are sufficient to construct the forward slicing algorithm.

2 System dependence graph

An SDG encodes the data, control, and call dependence relations between statements[†] of a program in a simple procedural language consisting of assignment, if-then-else, while-do, procedure call, entry, and return statements. The parameters to a procedure call are simple variables passed by value-reference. There is a special procedure *main* from which execution is initiated.

The SDG consists of a collection of procedure dependence graph (PDG) (a variation of program dependence graph [2, 5]). There is one PDG per procedure in the program encoding the control and data dependence relations within the procedure. These graphs contain vertices representing *call-sites* and procedure *entry* points. The SDG has *call edges* connecting the call-sites in a PDG to the entry point in the PDG of the procedure called at that site.

For each call-site, the PDG also contains two vertices for every actual parameter in the procedure call. An *actual-in* vertex to represent the transfer of value of the actual parameter to an intermediate variable used to send the input to the procedure. An *actual-out* vertex to represent transfer of the final value of the parameter from an intermediate variable to the actual parameter.

Analogously, for every entry point, the PDGs contain two vertices for every formal parameter. A *formal-in* vertex representing the transfer of value to the formal parameter from the intermediate variable

[†] It also has another dependence called the def-order dependence which is not relevant for slicing. This dependence is therefore ignored in this paper.

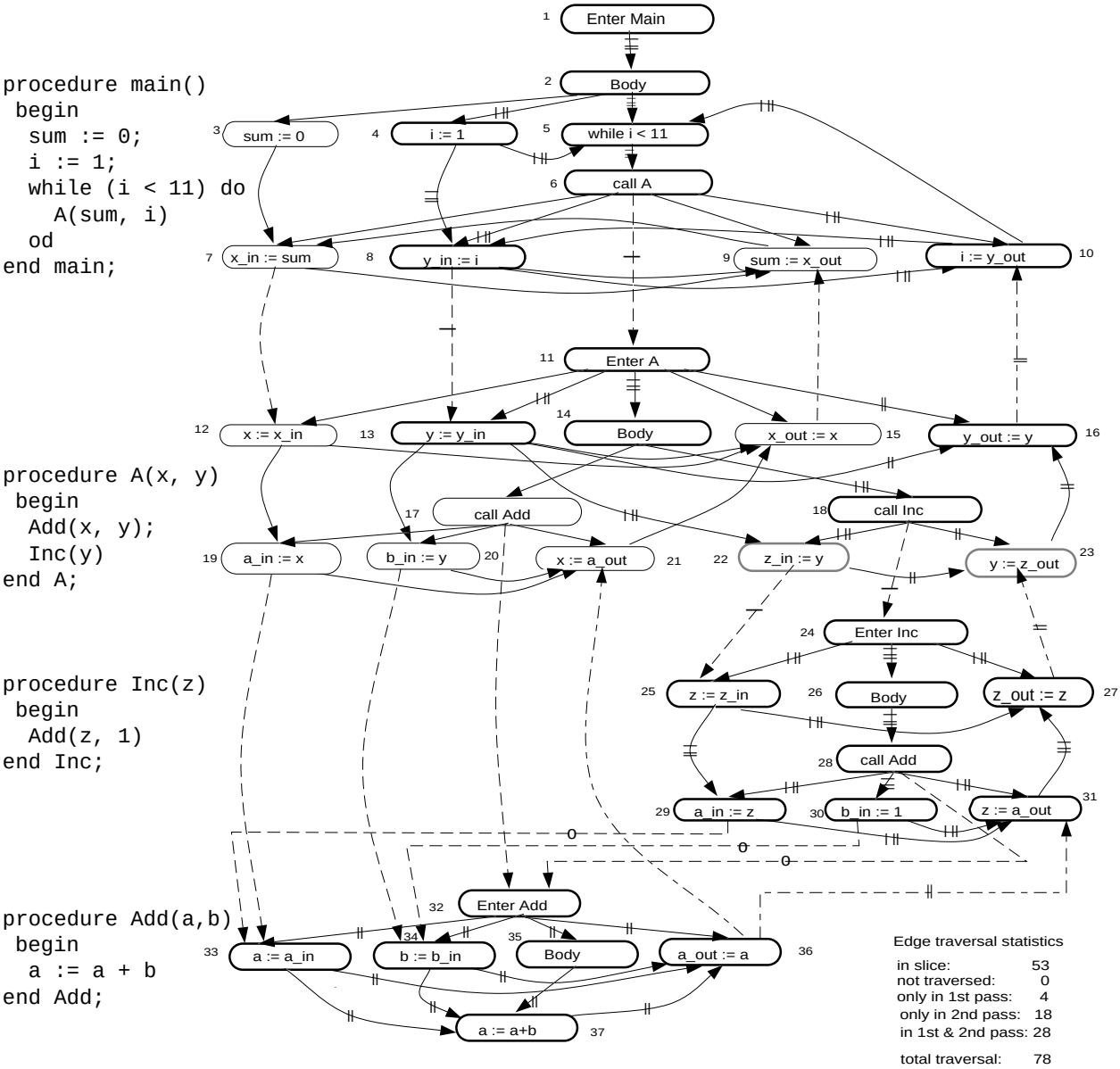

Figure 1 A program taken from [1] and its system dependence graph. Vertices are numbered for later reference. Intra edges are indicated by solid lines and From-To and To-From edges by (slightly different) dotted lines.
Vertices in the slice over the vertex set {25, 26} appear in boxes with thicker boundaries. The edges in the slice have an overstrike $|, ||, | \, ||$ depending on if they are traversed in the first pass, second pass, or both first and second pass of the HRB algorithm. Edges in the slice that are not traversed in either pass of the HRB algorithm are overstriked with an o.

assigned to in the *actual-in* vertex. A *formal-out* vertex representing the transfer of value from the formal parameter to the intermediate variable used at the *actual-out* vertex.

The pairs of intermediate variables used to communicate the initial and final values of a parameter to/from the procedure entry are unique. The SDG contains two edges, *parameter-in* and *parameter-out*, to represent the data dependence between the *actual-in* to *formal-in* and *formal-out* to *actual-out* vertices.

Since there is one PDG per procedure in the SDG, the actual-in (actual-out) vertices for the same parameter from different calls to the same procedure are depended upon by (depend on) the same formal-

procedure HRB_slicing_algorithm(G, S)
declare

G: a system dependence graph
S: sets of vertices in G

begin

First pass: Mark all vertices of G which can (reflexive, transitively) reach any vertex in S using *Intra* and *From-To* edges. Let the vertices marked be called S'.

Second pass: Mark all vertices of G which can transitively reach any vertex S using only *Intra* and *To-From* edges. The marked vertices are in the slice of G wrt S.

end

Figure 2 Summary of HRB interprocedural slicing algorithm

in (formal-out) vertex. The SDG also contains edges between the actual parameter vertices of the same procedure call and between the formal parameter vertices at the procedure entry. These edges, termed *summary edges*, summarize the dependence between the arguments of the procedure call as a result of executing the procedure.

In this paper we classify the edges in an SDG (excluding def-order dependence) in the following three categories:

Intra: Within vertices of a PDG - *control*, *flow*, and *summary*
From-To: From a call site to an entry site - *call* and *parameter-in*
To-From: To a call site from an entry site - *parameter-out*

3 HRB Interprocedural slicing algorithm

Ottenstein & Ottenstein [5] outline an algorithm for slicing a procedure-less program using PDG representation: it is the set of all statements that can reach any statement in the slicing criteria using control or data dependence edges.

When extending this algorithm for interprocedural slicing using SDG representation Horwitz et. al. note that traversing an SDG without discriminating between the type of edges traversed may create dependence paths between vertices of two procedures even when none exists. This happens because a procedure entry vertex may be connected to multiple call sites. A traversal may use parameter-in edge coming out of one call site and parameter-out vertex returning to another call site to create the incorrect linkage. Horwitz et. al. termed this the *calling context problem*. They developed a two pass traversal algorithm that correctly accounted for the calling context of a procedure [1]. The two passes differed in the edges used for traversal. In the first pass of their backward slicing algorithm, summarized in Figure 2, edges were chosen such that the traversal did not descend into a called procedure. Similarly, the choice of edges in the second pass limited the traversal from ascending into a procedure call. A similar approach is used by their forward slicing algorithm to correctly account for calling contexts in the forward traversal.

It may be noted that, earlier Myers' had provided a solution of the calling context problem in the framework of a set of data flow analysis problems. His solution required the maintenance of special markers representing the call site to keep track of the calling context. Horwitz et. al.'s solution to the calling context problem improved upon Myers' solution [4]. Our solution, presented in the next section, may be seen as presenting Horwitz et. al.'s solution in lattice theoretic framework.

```
procedure AL_slicing_algorithm(G, S)
declare
    G: a system dependence graph
    S: set of vertices
    Worklist: set of vertices
begin
    Worklist := S;
    mark tags of all vertices in S to ⊤;
    mark tags of all other vertices ⊥;
    while Worklist ≠ φ do
        Select and remove a vertex v from Worklist;
        for each vertex w such that edge e =w → v in G do
            case edge type of e do
            Intra: (* propagate v.tag to w *)
                newtag := w.tag ⊓ v.tag;
            To-From: (* propagate β to w *)
                newtag := w.tag ⊓ β;
            From-To: (* if v.tag is ⊤ propagate ⊤ else ignore w *)
                if v.tag = ⊤
                    then newtag := w.tag ⊓ ⊤
                    else newtag := w.tag;
            esac
            (* put w in the list only if its new tag value has increased *)
            if w.tag ⊏ newtag then
                w.tag := newtag;
                put w in the Worklist;
            fi
        rof
    elihw
    (* Vertices with tag values β and ⊤ are in the slice *)
end
```

Figure 3 One pass backward slicing algorithm. The statement blocks *if, for, case,* and *while* are terminated by *fi, rof, esac,* and *elihw*, respectively. Comments are enclosed in *(** and ***)*. After execution of statements for a *case* control is transferred to the statement following *esac* (as in PASCAL). The lattice $(\{\bot, \beta, \top\}, \sqcap)$ is defined so that $\bot \sqsubset \beta \sqsubset \top$.

4 AL interprocedural slicing algorithm

The AL algorithm for interprocedural slicing is given in Figure 3. Like HRB algorithm it maintains a $Worklist$ of vertices that have been marked so far and as vertices are traversed they are tagged. The AL algorithm maintains at each vertex a tag which can assume the values from the set $\{\bot, \beta, \top\}$. This set along with the meet operator $\sqcap$ forms a lattice where:

$\bot \sqcap \beta = \beta \sqcap \bot = \beta$,
$x \sqcap \top = \top \sqcap x = \top$, and
$x \sqcap x = x$.

Contrast this with the HRB algorithm's use of tags with two values: $marked$ and $unmarked$.

In the beginning of the AL algorithm all vertices in the slicing criterion are placed in the $Worklist$ and are assigned the tag $\top$. All other vertices are assigned the tag $\bot$. At the end of the algorithm all vertices whose tags are β or $\top$ are in the slice. The traversal requires picking each edge $e = w \rightarrow v$ in the SDG corresponding to a vertex v in the $Worklist$ and deciding:

1. whether w should be put in the $Worklist$ and, if so,
2. what the value of its tag should be.

When traversing Intra edges (i.e. with in the PDG) the tag at the target vertex of the edge is propagated to the vertex at the source. This is not however the case when From-To and To-From edges are traversed. A From-To edge (called to caller) is not traversed if the tag of the target vertex is not $\top$. When it is traversed the tag $\top$ is propagated to the source vertex. A To-From edge (caller to called) is always traversed. Irrespective of the tag of its target vertex a β tag is propagated to its source vertex. The source vertex is put in the *Worklist* only if its tag changes.

Table 1 enumerates the application of AL algorithm for the same example used for HRB slicing in Figure 1.

On-line union of slices implies creating a union of slices where a request for slice should be processed before the next one is received. The AL algorithm may be used to process on-line requests by making the following modifications:

1. move the statement "mark tags of all other vertices $\bot$" out of the algorithm into an initialization procedure that is performed before the first request is received and
2. the $Worklist$ may be initialized only by those vertices in S whose tag values at entry differ from $\top$.

5 Proof of equivalence

Definition: Let H denote the set of vertices in the slice of SDG G wrt vertex set S using HRB algorithm (Figure 2). Let H_1 and H_2 be the set of vertices marked by the algorithm in the first pass and second pass, respectively. Notice that $H_1 \cap H_2 = \phi$, $H_1 \cup H_2 = H$, and $S \subseteq H_1$.

Definition: Let L denote the set of vertices in the slice of SDG G wrt vertex set S using AL algorithm (Figure 3). Let L_β and $L_\top$ be the set of vertices whose with tag values β and $\top$, respectively. $L_\beta \cap L_\top = \phi$, $L_\beta \cup L_\top = L$, and $S \subseteq L_\top$.

We will show that $L_\top = H_1$ and $L_\beta = H_2$ thereby implying that $L = H$.

Definition: The edge type of a path in an SDG is the set of the type of edges in that path. If the edge type of a path is {*Intra*} or {} it is called an *in-pdg* path. If it is {*Intra, From-To*} it is called an *ascending path.* If it is {*Intra, To-From*} it is called a *descending path.*

The edge type of a path will be {} if it has only 0 or 1 vertex. Hence an *in-pdg* path consists of vertices in the same PDG. An ascending path traverses more than one PDG; always from the PDG of a procedure to that of a procedure calling it. A descending path also traverses more than one PDG; always from the PDG of a procedure to one it calls.

In the HRB algorithm the first pass traverses only *Intra* and *From-To* edges and the second pass only *Intra* and *To-From* edges. Hence, the first pass traverses only in-pdg and ascending paths and the second pass only in-pdg and descending paths. Notice that the paths are traversed in the reverse order.

Table 1 Application of AL algorithm for slicing the SDG in Figure 1 at vertex set {25, 27}. The first column shows the $Worklist$. At each iteration the first element from this list is selected. Vertices with tag $\top$ are added to the front of the list and those with tag β to the end. A $\diamond$ separates the two types of vertices. The last column gives the number of edges traversed in the previous iteration. The *Vertices marked* columns show the vertices marked with the respective tag in the beginning of that iteration. A ".." indicates that vertices from the previous iteration retain their respective tag.

	Vertices marked		*No. of Edges*
$Worklist$	$\top$	β	*Traversed*
25,27	25,27		
24,22,27	..,24,22		2
18,22,27	..,18		1
14,22,27	..,14		1
11,22,27	..,11		1
6,22,27	..,6		1
5,22,27	..,5		1
2,4,10,22,27	..,2,4,10		3
1,4,10,22,27	..,1		1
4,10,22,27	...		0
10,22,27	...		1
8,22,27$\diamond$ 16	..,8	16	3
22,27$\diamond$ 16	...	...	3
13,27$\diamond$ 16	..,13	...	2
27$\diamond$ 16	...	...	2
31$\diamond$ 16	..,31	...	3
28,29,30$\diamond$ 16,36	..,28,29,30	..,36	4
29,30$\diamond$ 16,36	...	...	1
30$\diamond$ 16,36	...	...	2
$\diamond$ 16,36	...	...	1
$\diamond$ 23,36	...	..,23	3
$\diamond$ 36	...	...	3
$\diamond$ 32,33,34,37	...	..,32,33,34,37	4
$\diamond$ 33,34,37	...	...	0
$\diamond$ 34,37	...	...	1
$\diamond$ 37	...	...	1
$\diamond$ 35	...	..,35	3
	...	...	1
		Total edges traversed	50

Table 2 Table showing how tags are propagated using our backward slicing algorithm. All paths traversed may be decomposed into a combination of *in-pdg*, *ascending*, and *descending* paths. The paths are traversed in the reverse direction. The value of $v_1.tag$ will be the $\sqcap$ of all the tags reaching it from various paths. The third column therefore shows the minmium value of $v_1.tag$. A value $\sqcap\bot$ indicates that such a path is not traversed.

	Edge type of path $v_1 \rightarrow ... \rightarrow v_n$	*Tag of* v_n	*Tag propagated to* v_1
1.	in-pdg	$\top$	$\sqcap\top$
2.	"	β	$\sqcap\beta$
3.	ascending	$\top$	$\sqcap\top$
4.	"	β	$\sqcap\bot$
5.	descending	$\top$	$\sqcap\beta$
6.	"	β	$\sqcap\beta$

Table 2 summarizes how AL algorithm (Figure 3) propagates tags along a path $v_1 \rightarrow ... \rightarrow v_n$ based on the type of the path and the value of $v_n.tag$. The following two Lemmas can be derived from this table, definition of $\sqcap$, and that the tags of vertices in S are initialized to $\top$.

Lemma 1: If there is an in-pdg or an ascending path from a vertex v to some vertex $s_i \in S$ **then** $v.tag = \top$, and *vice-versa*.

Proof: Left to the reader. ♡

Lemma 2: If $\exists a_j$ with $a_j.tag = \top$ such that there is a descending path from v to a_j **and** $\nexists s_i \in S$ such that there is an in-pdg or an ascending path from v to s_i **then** $v.tag = \beta$, and *vice-versa*.

Proof: Left to the reader. ♡

Theorem 1: $L_\top = H_1$ and $L_\beta = H_2$.

Proof: $L_\top = H_1$.

$v \in H_1$

The first pass of HRB algorithm only traverses in-pdg and ascending paths terminating at vertices in S.

$\equiv \exists$ an in-pdg or an ascending path from v to some $s_i \in S$.

Lemma 1

$\equiv v \in L_\top$.

Proof: $L_\beta = H_2$.

$v \in H_2$

The second pass of HRB algorithm traverses in-pdg and descending paths terminating at vertices in H_1.

Vertices already in H_1 are not included in H_2. The new vertices are reached by traversing descending paths (or else they would be in H_1).

$\equiv \exists a_j \in H_1$ such that there is a descending path from v to a_j and there is no in-pdg or an ascending path from v to any vertex in S.

$L_\top = H_1$

$\equiv \exists a_j \in L_\top$ such that there is a descending path from v to a_j and there is no ascending path from v to any vertex in S.

Lemma 2

$\equiv v \in L_\beta$. ♡

6 Analysis of complexity and implementation experience

The complexity of computing an interprocedural slice using SDG may be separated into the complexity of constructing the SDG and that of traversing this graph to identify statements in a slice. Horwitz et. al. [1] have analyzed that the cost of constructing SDG is polynomial in various parameters of the system and that the cost of traversing the SDG is bounded by the size of the SDG.

A more precise complexity of the traversal algorithm may be derived as a function of the number of edges in the final slice. Let E be the number of edges in the SDG of the slice with respect to vertex set S. The HRB and AL algorithms are both linear on the size of E. There is however a difference in number of times the two algorithms traverse each edge. The HRB algorithm requires at most two traversal of each edge in E for any choice of implementation of $Worklist$. In contrast, the AL algorithm can be implemented to traverse each edge at most one time. The details follow.

In a naive implementation of AL algorithm, the tag of each vertex can change twice (from $\perp$ to β and β to $\top$). This implies that the edges terminating at a vertex may be traversed twice. Consider however the following strategy to select a vertex from the $Worklist$:

select vertices with tag $\top$ before selecting any vertex with tag β

Since a β tag always propagates β, this strategy ensures that all vertices that may be marked $\top$ are visited before any vertex with a tag β is visited. Thus the tag of a vertex is changed at most once and each edge terminating at it is traversed only once.

Table 1 enumerates the application of AL algorithm using this strategy. The slice is computed by traversing 50 of the 53 edges in the slice. The 3 From-To edges (from a call-site in *Inc* to the entry of *Add*) marked o in Figure 1 are not traversed. These edges are also not traversed by the HRB algorithm. Figure 1 also contains the edge count statistics for the same example using the HRB algorithm; 28 edges are traversed twice thus getting a total edge traversal count to 78.

The HRB algorithm has been implemented in the Wisconsin Program Integration System (WPIS) [8]. We modified WPIS and implemented the AL algorithm as well. The user interface was modified so that that one or both of the slicing algorithm could be invoked. The two algorithms were implemented to use the depth-first strategy in selecting vertices for traversal. This precluded the need to maintain an explicit $Worklist$. When both the slicing algorithms were selected our implementation compared the results of the two algorithms to experimentally verify that the two algorithms create identical slices. The algorithms were also modified to collect statistics on edge traversal.

Figure 4 gives a comparative analysis of edge traversals for the two algorithm as a ratio of the total edges in the slice. Notice, that the depth-first traversal strategy for the AL algorithm does not ensure that edges be traversed only once. As a result there are some values in the region $x > 1.0$. However, there is only one value in the region $y < 1.0$ indicating that most of the time HRB algorithm required more traversals than the number of edges in the slice. Besides, even with the suboptimal implementation, the AL algorithm always traversed less number of edges than the HRB algorithm.

7 Conclusions

This paper presents an algorithm that improves upon the interprocedural slicing algorithm presented by Horwitz, Reps, and Binkley [1]. Our algorithm has the same order of complexity but with an improved constant. Instead of the (up to) two traversal of edges performed by HRB algorithm, our

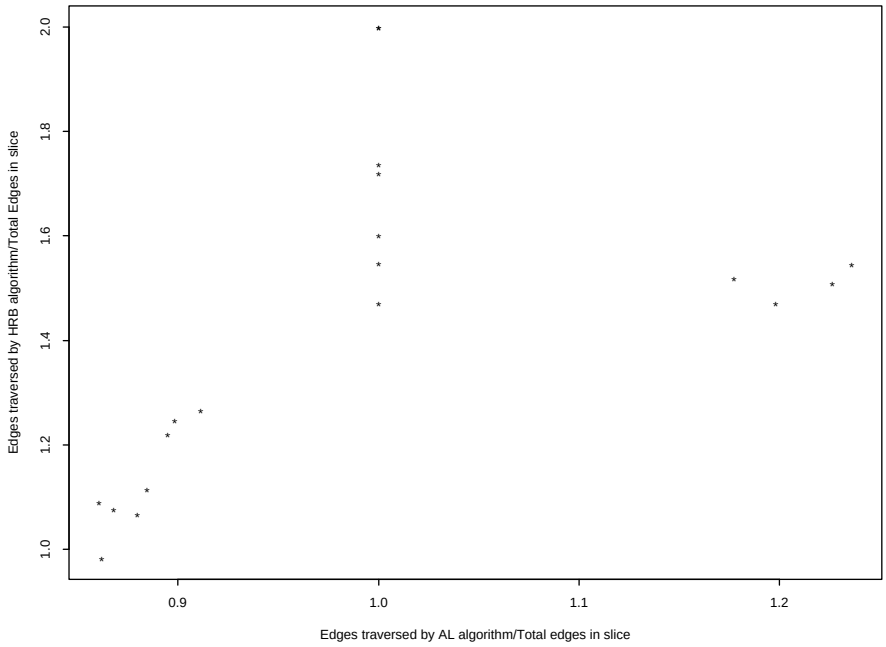

Figure 4 Comparative analysis of edges traversed by HRB algorithm and AL algorithm. The axes represent the edges traversed by two algorithm as a ratio of the total edges in the slice. Both the algorithms were implemented using depth-first graph traversal strategy. The results are from three programs besides the one in Figure 1. The dotted segment plots the values x =y. All the points lie above this line indicating that the number of edges traversed by HRB algorithm were always greater than those traversed by AL algorithm. Due to the choice of depth-first strategy, in 4 cases the AL algorithm traverses some vertices more than once ($x > 1.0$). In contrast, the HRB algorithm has all but one point in the region $y < 1.0$.

algorithm may be implemented to perform a maximum of one traversal per edge. In scenarios requiring *on-line* union of interprocedural slices our algorithm may be used to construct the slices incrementally. The HRB algorithm, as presented in [1], can not be used to generate union of slices incrementally. It may be modified to do so by maintaining a three valued tag, as done by our algorithm. The necessary relationships are presented in the theorem stating the equivalence between the slices computed by the two algorithms.

Acknowledgments: The study made use of the Wisconsin Program Integration System (WPIS) under license from the University of Wisconsin-Madison. We acknowledge Thomas Reps and Susan Horwitz for their role in its development. Nageswara Rao implemented the algorithm on WPIS and helped in drawing Figure 1. Pruek Poolkasem installed WPIS and GrammaTech's Synthesizer Generator [6, 7] needed by WPIS. The work was supported by the grant LEQSF (1991-92) ENH-98 from the Louisiana Board of regents and 1992 USL Summer Research Award.

Bibliography

[1] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Trans.*

Program. Lang. Syst. 12, 1 (1990), 26–60.

[2] Kuck, D. J., Muraoka, Y., and Chen, S. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers C-12*, 12 (Dec. 1972).

[3] Lakhotia, A. Constructing call multigraphs using dependence graphs. In *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'93)* (Jan. 1993 (to appear)).

[4] Myers, E. A precise interprocedural data flow algorithm. In *Proceedings of the 8th Annual Symposium on Principles of Programming Languages* (Jan. 1981), pp. 219–230.

[5] Ottenstein, K. J., and Ottenstein, L. M. The program dependence graph in a software development environment. *ACM SIGPLAN Notices 19*, 5 (May 1984).

[6] Reps, T., and Teitelbaum, T. *The Synthesizer Generator: A System for Contructing Language-Based Editor*. Springer-Verlag, New York, NY, 1988.

[7] Reps, T., and Teitelbaum, T. *The Synthesizer Generator Reference Manual*, third ed. Springer-Verlag, New York, NY, 1988.

[8] Reps, T. W. *The Wisconsin Program-Integration System Reference Manual*. University of Wisconsin-Madison, 1992.

[9] Wegman, M., and Zadeck, F. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 2 (Apr. 1990), 181–210.

[10] Weiser, M. Program slicing. *IEEE Trans. Softw. Eng. 10*, 4 (1984), 352–357.