Precise Interprocedural Chopping

Thomas Reps and Genevieve Rosay
University of Wisconsin

Abstract

The notion of a *program slice*, originally introduced by Mark Weiser, is a fundamental operation for addressing many software-engineering problems, including program understanding, debugging, maintenance, testing, and merging. A slice determines either all program elements that might affect a given element ("backward slicing") or all elements that could be affected by a given element ("forward slicing").

Jackson and Rollins introduced a related operation, called *program chopping*, which is a kind of "filtered" slice: Chopping answers questions of the form "What are all the program elements *v* that serve to transmit effects from a given source element *s* to a given target element *t*?" However, Jackson and Rollins define only a limited form of chopping: Among other restrictions, they impose the restriction that *s* and *t* be in the same procedure.

This paper solves the unrestricted interprocedural chopping problem, as well as a variety of other useful variants of interprocedural chopping.

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: program dependence graph, program slicing, program chopping, debugging, interprocedural analysis, graph reachability, realizable path

1. Introduction

The *slice* of a program with respect to program-point *p* and variable *x* consists of all statements and predicates of the program that might affect the value of *x* at point *p*. This concept, originally discussed by Mark Weiser in [31], allows one to isolate individual computation threads within a program. Program slicing is a fundamental operation for addressing many software-engineering problems, including program understanding, debugging, maintenance, testing, and merging [19,12,14,9,5,4,22,20].

Jackson and Rollins introduced a related operation, called *program chopping* [15], which can be thought of as a kind of "filtered" slice: Chopping answers questions of the form "What are all the program elements *v* that serve to transmit effects from a given source element *s* to a given target element *t*?" Compared to slicing, chopping provides a more focused way of obtaining information about the transmission of effects through a program.

Example. Consider a situation in which a programmer adds a statement to a program at site *s*, only to find that the modified program now crashes at site *t*. The chop of the modified program with respect to *s* and *t* must contain all of the elements that transmitted effects from change-site *s* to failure-site *t*. □

The vertices in a chop with respect to *s* and *t* are a subset of both the vertices in the forward slice with respect to *s* and the vertices in the backward slice with respect to *t*. In most cases, the size of a chop is substantially smaller than the size of either the forward or backward slice (see Figure 8). (Furthermore, the chop with respect to *s* and *t* is, in general, a proper subset of the intersection of the forward slice from *s* with the backward slice from *t*.)

Jackson and Rollins solve only a limited class of chopping problems: Among other restrictions, they impose the restriction that *s* and *t* be in the same procedure. In contrast, this paper solves the *unrestricted interprocedural chopping problem*, as well as a variety of other useful variants of interprocedural chopping.

Interprocedural chopping involves generating a chop of an entire program, where the chop crosses the boundaries of procedure calls. In interprocedural-chopping problems, not all of the paths in the graph that represents the program correspond to possible execution paths. In general, the question of whether a given path is a possible execution path is undecidable, but certain paths can be identified as being impossible because they would correspond to execution paths with infeasible call/return linkages. For example, if procedure *Main* calls *P* twice—say at c_1 and c_2—one infeasible execution path would start at the entry point of *Main*, travel through *Main* to c_1, enter *P*, travel through *P* to the return point, and return to *Main* at c_2 (rather than c_1). Such paths fail to account correctly for the calling context (*e.g.*, c_1 in *Main*) of a called procedure (*e.g.*, *P*). Thus, one of the issues for obtaining *precise* solutions to chopping problems is to carry out the analysis of the program to ensure that only effects transmitted along *interprocedurally realizable paths* are considered (see Definition 2.1).

Example. An example program that sums the numbers from 1 to 10 is shown in column one of Figure 1. Columns two, three, and four show the results reported by three candidate algorithms for interprocedural chopping. Because the final value of *i* does not depend on the initial value of *sum*, the precise chop (column four) is empty. Column two shows what would be reported by an imprecise chopping method that (pessimistically, but safely) reports "effects"

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and 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).

Authors' address: Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.
Electronic mail: {reps,rosay}@cs.wisc.edu.

© 1995 ACM

In Proc. of the 3rd ACM Symp. on Foundations of Software Engineering ((Washington, DC, Oct. 10-13, 1995).

Example program	Imprecise same-level chop	"Naive" same-level chop	Precise same-level chop
proc *Main*	**proc** *Main*	**proc** *Main*	**proc** *Main*
s: *sum* := 0	*s*: *sum* := 0	*s*: *sum* := 0	*s*: *sum* := 0
i := 1	*i* := 1	*i* := 1	*i* := 1
while *i* < 11 **do**	**while** *i* < 11 **do**	**while** *i* < 11 **do**	**while** *i* < 11 **do**
call *Add* (*sum*, *i*)	**call** *Add* (*sum*, *i*)	**call** *Add* (*sum*, *i*)	**call** *Add* (*sum*, *i*)
call *Add* (*i*, 1)	**call** *Add* (*i*, 1)	**call** *Add* (*i*, 1)	**call** *Add* (*i*, 1)
od	**od**	**od**	**od**
output(*sum*)	**output**(*sum*)	**output**(*sum*)	**output**(*sum*)
t: **output**(*i*)	*t*: **output**(*i*)	*t*: **output**(*i*)	*t*: **output**(*i*)
end	**end**	**end**	**end**
proc *Add*(*x*: **inout**, *y*: **in**)	**proc** *Add*(*x*: **inout**, *y*: **in**)	**proc** *Add*(*x*: **inout**, *y*: **in**)	**proc** *Add*(*x*: **inout**, *y*: **in**)
$x := x + y$	$x := x + y$	$x := x + y$	$x := x + y$
end	**end**	**end**	**end**

Figure 1. An example program and the results reported by three different algorithms for non-truncated same-level interprocedural chopping. In all cases, the source element *s* and target element *t* used for chopping are *sum* := 0 and **output**(*i*), respectively.

transmitted along *all* paths (including non-realizable paths). □

Jackson and Rollins address only a limited form of the interprocedural-chopping problem. Their version of chopping has two restrictions:

(i) Source-element *s* and target-element *t* must be in the same procedure. (We call this the *same-level chopping* restriction.)

(ii) Information is reported for one procedure only—the procedure containing *s* and *t*. Although (same-level) realizable paths through other procedures that serve to transmit effects from *s* to *t are* taken into account by the Jackson-Rollins method, the elements on these paths are not reported. (We call this the *truncation* restriction.)

In other words, Jackson and Rollins solve only the ***truncated same-level chopping problem***.

The contributions of this paper can be summarized as follows:

- We identify three new variants of the interprocedural-chopping problem: (i) the ***unrestricted*** chopping problem, (ii) the ***truncated unrestricted*** chopping problem, and (iii) the ***(non-truncated) same-level*** chopping problem. These correspond to three different ways of generalizing the interprocedural-chopping problem studied by Jackson and Rollins.
- We present ***algorithms*** for solving these versions of the chopping problem precisely up to realizable paths.
- We show that our algorithms run in ***polynomial time***.
- We have implemented these algorithms to create ***a tool for chopping C programs***. Our limited experience with the tool is in agreement with what the asymptotic bounds on the running times of the algorithms predict:
  - Truncated unrestricted chopping is not substantially more expensive to perform than interprocedural slicing.
  - Both of the non-truncated chopping problems are substantially more expensive to perform than their truncated counterparts, although in practice it may still be feasible for a system to support the non-truncated chopping operations.

As we will see in Section 3, the Jackson-Rollins chopping method involves finding the intersection of two slices—a forward slice from source element *s* and a backward slice from target element *t*. For the truncated same-level chopping problem, this method *does* correctly account for the realizable-path issue. It is tempting to try to solve the three more general versions of the chopping problem by also taking the intersection of two slices; however, this method produces imprecise answers.

Example. Column three of Figure 1 illustrates what would be reported by a method that produces non-truncated same-level chops by taking the intersection of a forward slice and a backward slice. This *imprecisely* reports several elements in procedure *Add*. A precise chopping method would determine that there are *no* elements that transmit effects from source element *s* : *sum* := 0 to target element *t* : **output**(*i*) and report that the chop is empty. This is illustrated in column four of Figure 1. □

The remainder of the paper is organized as follows: Section 2 presents terminology and notation, and defines the interprocedural-chopping problems that will be considered in the paper. Section 3 describes how to solve these chopping problems precisely (*i.e.*, up to realizable paths). Section 4 presents some conclusions.

2. Terminology, Notation, and Definition of the Problem

Slicing and chopping are related operations, and both can be performed by solving certain kinds of reachability problems in a dependence-graph representation of the program [23,13,28,15]. In the case of *intra*procedural slicing, for example, Ottenstein and Ottenstein showed that slices can be obtained by solving a reachability problem on the program dependence graph (PDG): To perform a slice with respect to PDG vertex *v*, find all PDG vertices from which there is a path to *v* along control-dependence and/or flow-dependence edges [23]. In other words, (backward) slicing is a *single-target* reachability problem on the PDG. The related problem of forward slicing is a *single-source* reachability problem on the PDG. Chopping corresponds to a *single-source/single-target* reachability problem.

*Inter*procedural slices and chops could also be obtained by solving single-source, single-target, and single-source/single-target reachability problems on dependence graphs; however, the results obtained using this approach include "extra" components because not all paths in a dependence graph for a multi-procedure program correspond to possible execution paths. In order to obtain more precise interprocedural chops, instead of considering *all* paths in the program, we want to consider only *realizable* paths—paths that reflect the fact that when a procedure call finishes, execution returns to the site of the most recently executed call.[1]

2.1. System Dependence Graphs

The notion of a "realizable path" is most easily understood in terms of the concept of a "system dependence graph", a graph used to represent multi-procedure programs ("systems"). The system dependence graph is similar to other dependence-graph representations of programs (*e.g.*, [16,8]), but represents collections of procedures rather than just monolithic programs.

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.[2]

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. The example program and its system dependence graph are shown in Figure 2. □

Remark. SDGs are really a *class* of program representations. To represent programs in different programming languages one would use different kinds of PDGs, depending on the features and constructs of the given language. Although our examples use a very simple programming language (with assignment statements, conditionals, while-loops, call statements, and value-result parameter passing), the reader should keep in mind that we use the term "SDG" in the generic sense. In particular, our results should *not* be thought of as being tied to the restricted language used in our examples; they hold no matter what variety of SDG they are applied to.

The issue of how to create appropriate PDGs/SDGs is orthogonal to the issue of how to slice and chop them. A variety of other studies have investigated how to build dependence graphs for various features and constructs found in real-world programming languages. For example, previous work has addressed arrays [3,32,21,10,24,25], reference parameters [13], pointers [18,11,6], and non-structured control flow [2,7,1]. (Our C chopping tool incorporates several of these techniques.) □

2.2. Realizable Paths

The concept of a *realizable path* is formalized via the following definition:

Definition 2.1 (Realizable Paths). Let each call vertex in SDG G be given a unique index from 1 to *CSites*, where *CSites* is the total number of call sites in the program. 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 "$(_i$". Label all other edges in G with the symbol e.

A path in G is a ***same-level realizable path*** iff the sequence of symbols labeling edges in the path is a string in the language generated from nonterminal *matched* by the following context-free grammar:

$$
\begin{aligned}
\textit{matched} \;\rightarrow\;& \textit{matched}\ \ \textit{matched} \\
\mid\;& (_i\ \ \textit{matched}\ \ )_i \qquad \text{for } 1 \le i \le \textit{CSites} \\
\mid\;& e \\
\mid\;& \varepsilon
\end{aligned}
$$

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

$$
\begin{aligned}
\textit{unbalanced-right} \;\rightarrow\;& \textit{unbalanced-right}\ \ )_i\ \ \textit{matched} \quad \text{for } 1 \le i \le \textit{CSites} \\
\mid\;& \textit{matched} \\
\textit{unbalanced-left} \;\rightarrow\;& \textit{unbalanced-left}\ \ (_i\ \ \textit{matched} \quad \text{for } 1 \le i \le \textit{CSites} \\
\mid\;& \textit{matched} \\
\textit{realizable} \;\rightarrow\;& \textit{unbalanced-right}\ \ \textit{unbalanced-left}
\end{aligned}
$$

□

A same-level realizable path from v to w represents the transmission of an effect from v to w, where v and w are in the same procedure, via a sequence of execution steps during which the call stack can temporarily grow deeper—because of calls—but never shallower than its original depth before eventually returning to its original depth. The graph in the left half of Figure 3 shows an example of a same-level realizable path.

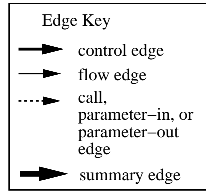
A realizable path from v to w represents the transmission of an effect from v to w, where v and w are *not* required to

---

[1] A similar goal of considering only paths that correspond to legal call/return sequences arises in the context of interprocedural dataflow analysis [30,17,29]. Several different terms have been used for these paths, including *valid paths*, *feasible paths*, and *realizable paths*.

[2] As defined in [13], procedure dependence graphs include four kinds of dependence edges: control, loop-independent flow, loop-carried flow, and def-order. However, for chopping 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-dependence edges and a single kind of flow-dependence edge.

```
proc Main
    sum := 0
    i := 1
    while i < 11 do
        call Add (sum, i)
        call Add (i, 1)
    od
    output(sum)
    output(i)
end
proc Add(x: inout, y: in)
    x := x + y
end
```



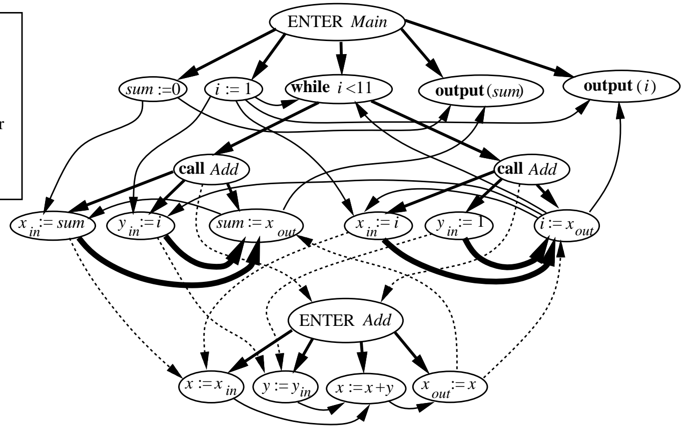


Figure 2. The example program and its corresponding system dependence graph. (Summary edges are discussed in Section 3.1.)

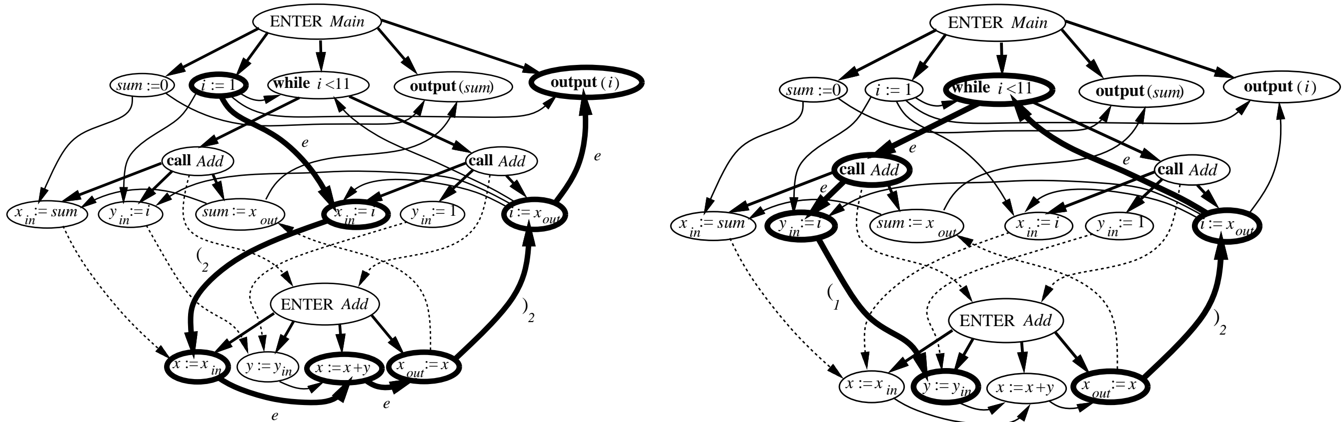


Figure 3. In the graph on the left, the path shown in bold is a same-level realizable path from $i := 1$ in procedure *Main* to **output**(i) in *Main*. The graph on the right shows an example of a realizable path that is not a same-level realizable path. It runs from $x_{out} := x$ in *Add* to $y := y_{in}$ in *Add*. Although both endpoints of the path are in procedure *Add*, the path is not a same-level realizable path: In particular, "$x_{out} := x$" $\rightarrow^*_{unbr}$ "**while** $i < 11$" is an "unbalanced-right" subpath, and "**while** $i < 11$" $\rightarrow^*_{unbl}$ "$y := y_{in}$" is an "unbalanced-left" subpath.

be in the same procedure. The "unbalanced-right" part of a realizable path represents an execution sequence that may leave the call stack shallower than it was originally; the "unbalanced-left" part represents an execution sequence that may leave the call stack deeper than it was originally. The graph in the right half of Figure 3 shows an example of a realizable path that is not a same-level realizable path.

We use the notation $p = u \rightarrow^* w$ to mean that p is a path from u to w (in the SDG of interest). The notation $v \in p$ means that v is one of the vertices of path p. When we are only concerned with the existence of a path between u and w, we indicate this by $u \rightarrow^* w$.

Ordinarily, we are also interested in characterizing paths as "matched", "unbalanced-right", "unbalanced-left", or "realizable" (corresponding to the nonterminals used in Definition 2.1). We use subscripts to indicate these characterizations: $u \rightarrow^*_m w$, $u \rightarrow^*_{unbr} w$, $u \rightarrow^*_{unbl} w$, and $u \rightarrow^*_r w$, respectively.

It is important to understand that the realizable-path relation $\rightarrow^*_r$ is *not a transitive relation*; that is, $u \rightarrow^*_r v$ and $v \rightarrow^*_r w$ does not necessarily imply that $u \rightarrow^*_r w$. The reason is that when the path $p1 = u \rightarrow^*_r v$ is an unbalanced-left path and the path $p2 = v \rightarrow^*_r w$ is an unbalanced-right path, the combined path $p1 \| p2$ is a realizable path only when each of the unmatched $)_i$ symbols in path $p2$ match up with an unmatched $(_j$ symbol in path $p1$; this need not be the case.

The other three relations, $\rightarrow^*_m$, $\rightarrow^*_{unbr}$, and $\rightarrow^*_{unbl}$, *are* transitive relations.

2.3. The Interprocedural-Chopping Problem

We now define four varieties of interprocedural-chopping problems:

Definition 2.2 (Precise Interprocedural Chopping). In each of the following clauses, an SDG G is chopped with respect to a given source vertex s and a given target vertex t.

(i) An ***unrestricted interprocedural chop*** consists of the set of vertices

$$\{ v \mid \exists p \text{ such that } p = s \rightarrow_r^* t \text{ and } v \in p \}.$$

(ii) A ***truncated unrestricted interprocedural chop*** consists of the set of vertices

$$\{ v \mid \exists w \text{ such that } s \rightarrow_{unbr}^* v \rightarrow_{unbr}^* w \rightarrow_{unbl}^* t \}$$
$$\cup \{ v \mid \exists w \text{ such that } s \rightarrow_{unbr}^* w \rightarrow_{unbl}^* v \rightarrow_{unbl}^* t \}.$$

(iii) A ***(non-truncated) same-level interprocedural chop*** consists of the set of vertices

$$\{ v \mid \exists p \text{ such that } p = s \rightarrow_m^* t \text{ and } v \in p \}.$$

(iv) A ***truncated same-level interprocedural chop*** consists of the set of vertices

$$\{ v \mid s \rightarrow_m^* v \rightarrow_m^* t \}.$$

□

Several remarks about Definition 2.2 are in order:

- Clause (iv) of Definition 2.2 is essentially equivalent to the interprocedural-chopping problem defined by Jackson and Rollins. Jackson and Rollins actually permit chopping with respect to a *set* of source elements and a *set* of target elements; however, extending clauses (i)−(iv) to permit multi-source/multi-target chops would require only minor changes to the chopping algorithms presented in Section 3. The details are left to the reader.
- Definition 2.2 defines the chop operation to return a set of vertices. An alternative approach is to define the chop operation to return a set of edges. Again, this would require only modest changes in Definition 2.2 and the algorithms of Section 3.
- The two truncated chopping operations are more focused than their unrestricted counterparts. They do not report the vertices that occur along "side excursions" (*i.e.*, same-level realizable paths) through called procedures. (It is important to understand that the *effects* transmitted along such "digressions" *are* taken into account; however, the vertices encountered on such paths are not reported.)
- Note that in clause (i) of Definition 2.2, because the realizable-path relation is not transitive, we could not define an unrestricted interprocedural chop to be the set of vertices $\{ v \mid s \rightarrow_r^* v \rightarrow_r^* t \}$. Similarly, in clause (iii), we could not define a non-truncated same-level interprocedural chop to be the set of vertices $\{ v \mid s \rightarrow_{unbl}^* v \rightarrow_{unbr}^* t \}$. Both of these definitions would report the vertices shown in the "naive" chop given Figure 1. Figure 4 illustrates how the "naive" chop is obtained.

Example. Examples of a truncated unrestricted chop (Definition 2.2(ii)) and a non-truncated unrestricted chop (Definition 2.2(i)) are shown in Figure 5. □

3. Precise Interprocedural Chopping

In this section, we describe how to solve interprocedural-chopping problems precisely up to realizable paths.

Remark. Although recursion is not used in our running example, the algorithms presented below work correctly for programs with recursive and mutually recursive procedures. □

3.1. Summary Edges

As with the SDG-based techniques for interprocedural slicing, the first step is to augment the SDG with *summary*

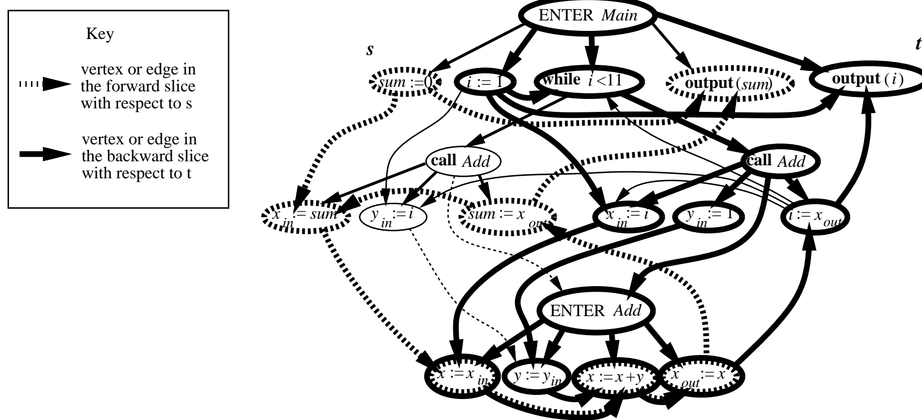


Figure 4. This illustrates how the "naive" chop shown in column three of Figure 1 is obtained. There is both an unbalanced-left path (and hence a realizable path) from s to the vertices $x := x_{in}$, $x := x + y$, and $x_{out} := x$, as well as an unbalanced-right path from them to t.

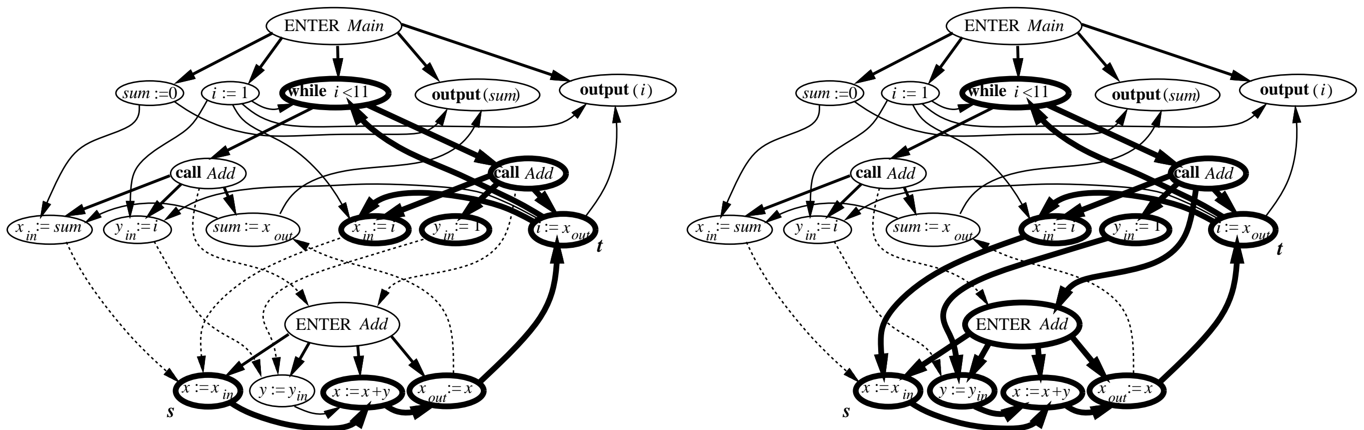


Figure 5. The graph on the left shows a truncated unrestricted chop. The graph on the right shows a non-truncated unrestricted chop.

edges, which record at call sites the existence of certain same-level realizable paths in the SDG:

$$
\begin{aligned}
&\mathit{SummaryEdges} = \qquad\qquad (\dagger)\\
&\quad \{\, (v, w) \mid v \in \mathit{ActualInVertices},\ w \in \mathit{ActualOutVertices},\\
&\qquad\quad v \text{ and } w \text{ are associated with the same call site},\\
&\qquad\quad (v, v') \in \mathit{ParameterInEdges},\\
&\qquad\quad (w', w) \in \mathit{ParameterOutEdges},\ \text{and } v' \to_m^* w' \,\}\\
&\quad \cup \{\, (v, w) \mid v \in \mathit{CallVertices},\ w \in \mathit{ActualOutVertices},\ \text{and}\\
&\qquad\quad \text{there is a control-dependence edge from } v \text{ to } w \,\}.
\end{aligned}
$$

(In previous work, "summary edge" has referred to actual-in/actual-out pairs meeting the condition of the first term of equation (†). For technical reasons, it is necessary for us to treat all control-dependence edges from call vertices to actual-out vertices in the same fashion as actual-in-to-actual-out summary edges, and it is for this reason that we refer to both classes of edges as "summary edges". However, when the SDG is augmented with summary edges, no additional call-to-actual-out edges actually need to be added to the SDG.)

A summary edge from an actual-in vertex v (corresponding to some actual parameter x before the call) to an actual-out vertex w (corresponding to some actual parameter y after the call) represents the fact that the value of y after the call might depend on the value of x before the call.

Note that we cannot determine whether there should be an actual-in-to-actual-out summary edge from v to w simply by determining whether there is a path in the SDG from v to w (for example, 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 and chopping, namely that not all paths in the SDG are realizable paths. An efficient method for determining the set of all actual-in-to-actual-out summary edges is given in [28].

Example. The actual-in-to-actual-out summary edges for the running example are shown in Figure 2. □

3.2. Following Matched, Unbalanced-Right, and Unbalanced-Left Paths

Once all summary edges have been found, it is an easy matter to "follow" matched, unbalanced-right, and unbalanced-left paths in the SDG. We define the following six operations on sets of vertices:

Definition 3.1.

$$
\begin{aligned}
f_m(S) &=_{df} \{\, v \mid \exists s \in S \text{ such that } s \to_m^* v \,\}\\
f_{unbr}(S) &=_{df} \{\, v \mid \exists s \in S \text{ such that } s \to_{unbr}^* v \,\}\\
f_{unbl}(S) &=_{df} \{\, v \mid \exists s \in S \text{ such that } s \to_{unbl}^* v \,\}
\end{aligned}
$$

$$
\begin{aligned}
b_m(T) &=_{df} \{\, v \mid \exists t \in T \text{ such that } v \to_m^* t \,\}\\
b_{unbr}(T) &=_{df} \{\, v \mid \exists t \in T \text{ such that } v \to_{unbr}^* t \,\}\\
b_{unbl}(T) &=_{df} \{\, v \mid \exists t \in T \text{ such that } v \to_{unbl}^* t \,\}
\end{aligned}
$$

□

Operations f_{unbr}, f_{unbl}, b_{unbr}, and b_{unbr} correspond to the individual "slicing passes" defined in [13].[3] All six operations can be implemented by simple reachability computations (for example, by depth-first search) on an SDG augmented with summary edges:

(i) The operation $f_m(S)$ can be implemented as a reachability algorithm as follows: Starting at members of S, the depth-first search traverses control-dependence edges, flow-dependence edges, and summary edges, but *not* call edges, parameter-in edges, or parameter-out edges, and returns all vertices encountered.

[3] The full forward and backward slicing operations themselves are defined as follows:

$$
\begin{aligned}
f_r(S) &=_{df} \{\, v \mid \exists s \in S \text{ s.t. } s \to_r^* v \,\} & b_r(T) &=_{df} \{\, v \mid \exists t \in T \text{ s.t. } v \to_r^* t \,\}\\
&= (f_{unbl} \circ f_{unbr})(S) & &= (b_{unbr} \circ b_{unbl})(T)\\
&= f_{unbl}(f_{unbr}(S)) & &= b_{unbr}(b_{unbl}(T)).
\end{aligned}
$$

(ii) Operation $f_{unbr}(S)$ can be implemented as a reachability algorithm that starts at members of S and traverses control-dependence edges, flow-dependence edges, summary edges, and parameter-out edges, but *not* call edges or parameter-in edges.

(iii) The operation $b_{unbl}(T)$ can be implemented as a backwards reachability algorithm (*e.g.*, by a depth-first search that traverses edges backwards, from target to source): Starting at members of T, it traverses control-dependence edges, flow-dependence edges, call edges, summary edges, and parameter-in edges, but *not* parameter-out edges.

The other three operations are implemented similarly.

The running time for all six operations is linear in the size of the SDG augmented with summary edges.

3.3. Same-Level Interprocedural Chopping

Our interprocedural-chopping algorithms are given in Figure 6. Because same-level chopping is used as a subroutine in the algorithm for unrestricted interprocedural chopping, we start by explaining the algorithms for the same-level interprocedural-chopping operations.

A truncated same-level interprocedural chop with respect to source s and target t consists of the set of vertices $\{ v \mid s \rightarrow_m^* v \rightarrow_m^* t \}$, or equivalently, the vertices v such that there are same-level realizable paths from both s to v and v to t. Therefore, TruncatedSameLevelChop(s, t) returns the set $f_m(\{ s \}) \cap b_m(\{ t \})$ (*i.e.*, the intersection of a forward "matched slice" from s and a backward "matched slice" from t, as proposed by Jackson and Rollins.)

A (non-truncated) same-level interprocedural chop consists of the set of vertices

$$\{ v \mid \exists p \text{ such that } p = s \rightarrow_m^* t \text{ and } v \in p \}.$$

Function SameLevelChop finds this set by first performing an initial TruncatedSameLevelChop and then extending the set of vertices returned, repeatedly performing additional TruncatedSameLevelChop operations on certain pairs of {formal-in, enter}/formal-out vertices—those pairs (v', w') that correspond to a summary edge (v, w) for which both v and w were found during a previous call on TruncatedSameLevelChop. This iterative process is carried out by auxiliary function SameLevelChopAux.

Because a summary edge (x, y) indicates that a same-level realizable path from $x \rightarrow_m^* y$ exists in the SDG, after each call on TruncatedSameLevelChop in SameLevelChopAux, all pairs of vertices $(x, y) \in$ *SummaryEdges* such that x and y are in the truncated same-level chop are added to the work-list to schedule further calls on TruncatedSameLevelChop (see lines [12]–[14] of SameLevelChopAux). In other words, during the loop in lines [5]–[17], whenever we have determined that the SDG contains a same-level realizable path $v \rightarrow v' \rightarrow_m^* x \rightarrow_m^* y \rightarrow_m^* w' \rightarrow w$, where (x, y) is a summary edge, (x, y) is placed on the work-list to schedule a further call on TruncatedSameLevelChop.

To prevent SameLevelChopAux from ever invoking TruncatedSameLevelChop more than once on a given pair of {formal-in, enter}/formal-out vertices, each {formal-in, enter}/formal-out pair taken out of the work-list is marked, and only unmarked pairs cause new calls on TruncatedSameLevelChop to be performed (see lines [4], [10], [11], and [12]).

Bounds on the running times of the different chopping algorithms can be expressed in terms of the following parameters:

P	The number of procedures in the program.
$CSites_p$	The number of call sites in procedure p.
MaxSites	The maximum number of call sites in any procedure.
CSites	The total number of call sites in the program. (This is bounded by $P \times MaxSites$.)
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.

Remark. In analyzing the costs of the chopping algorithms, we will assume that the SDG's summary edges have already been determined. This can be performed in time $O((P \times E \times Params) + (CSites \times Params^3))$ [28]. Note that this step only has to be performed once, when the first slice or chop is performed. Alternatively, the summary edges can be determined on demand, which allows some of the cost to be spread out over a sequence of slices and chops [27]. □

Function TruncatedSameLevelChop traverses edges and summary edges of only a single procedure, and the amount of work performed is linear in the number of edges traversed. Because there are at most $Params^2$ summary edges at each call site, the running time of TruncatedSameLevelChop is bounded by

$$O(E + MaxSites \times Params^2).$$

Most of the work for function SameLevelChop is performed by SameLevelChopAux. The version of SameLevelChopAux given in Figure 6 can perform a TruncatedSameLevelChop operation at most once for each {formal-in, enter}/formal-out pair in a PDG of the SDG. Because there are at most $Params^2$ such pairs in each procedure, the running time of SameLevelChop is bounded by

$$O\Big(\sum_{p \in Procedures} Params^2 \times (E + CSites_p \times Params^2)\Big)$$
$$= O((P \times E \times Params^2) + (CSites \times Params^4)).$$

A more efficient version of SameLevelChopAux that yields an asymptotically more efficient version of SameLevelChop (as well as Chop) is presented in Section 3.5. We have presented the less efficient version first because it is the easier of the two to understand.

3.4. Unrestricted Interprocedural Chopping

Because truncated unrestricted chopping (function TruncatedChop) is used as an initialization step in the algorithm for fully unrestricted chopping (function Chop), we first discuss TruncatedChop. A truncated unrestricted chop consists of the set of vertices

$$\{ v \mid \exists w \text{ such that } s \rightarrow_{unbr}^* v \rightarrow_{unbr}^* w \rightarrow_{unbl}^* t \} \cup \{ v \mid \exists w \text{ such that } s \rightarrow_{unbr}^* w \rightarrow_{unbl}^* v \rightarrow_{unbl}^* t \} \quad (‡)$$

Because the relations $\rightarrow_{unbr}^*$ and $\rightarrow_{unbl}^*$ are both transitive, in both terms of (‡) the condition on w is that $s \rightarrow_{unbr}^* w \rightarrow_{unbl}^* t$ hold. Thus, the first step is to find all such vertices w:

```
function TruncatedSameLevelChop(s, t)
  returns vertex set
begin
[1]  return f_m({ s }) ∩ b_m({ t })
end
```

```
function SameLevelChop(s, t) returns vertex set
declare
  S: vertex set
  WorkList: set of summary edges
begin
[1]  S := TruncatedSameLevelChop(s, t)
[2]  WorkList := { (x, y) ∈ SummaryEdges | x, y ∈ S }
[3]  return SameLevelChopAux(S, WorkList)
end
```

```
function SameLevelChopAux(Answer, WorkList) returns vertex set
parameters
  Answer: vertex set
  WorkList: set of summary edges
declare
  S: vertex set
begin
[4]   Remove all marks on {formal-in, enter}/formal-out pairs
[5]   while WorkList ≠ ∅ do
[6]     Select and remove a summary edge (v, w) from WorkList
[7]     let v′ = the formal-in or enter vertex that corresponds to v and
[8]         w′ = the formal-out vertex that corresponds to actual-out w
[9]     in
[10]      if (v′, w′) is unmarked then
[11]        Mark (v′, w′)
[12]        S := TruncatedSameLevelChop(v′, w′)
[13]        Answer := Answer ∪ S
[14]        WorkList := WorkList ∪ { (x, y) ∈ SummaryEdges | x, y ∈ S }
[15]      fi
[16]    ni
[17]  od
[18]  return Answer
end
```

```
function TruncatedChop(s, t)
  returns vertex set
declare
  W, VR, VL: vertex sets
begin
[1]  W := f_unbr({ s }) ∩ b_unbl({ t })
[2]  VR := f_unbr({ s }) ∩ b_unbr(W)
[3]  VL := f_unbl(W) ∩ b_unbl({ t })
[4]  return VR ∪ VL
end
```

```
function Chop(s, t) returns vertex set
declare
  W, VR, VL: vertex sets
  WorkList: set of summary edges
begin
[1]   /* Perform a TruncatedChop */
[2]     W := f_unbr({ s }) ∩ b_unbl({ t })
[3]     VR := f_unbr({ s }) ∩ b_unbr(W)
[4]     VL := f_unbl(W) ∩ b_unbl({ t })
[5]   /* Invoke SameLevelChopAux with all summary edges on
[6]    * unbalanced-right paths from s to W or unbalanced-left
[7]    * paths from W to t */
[8]     WorkList :=   { (x, y) ∈ SummaryEdges | x, y ∈ VR }
[9]                 ∪ { (x, y) ∈ SummaryEdges | x, y ∈ VL }
[10]  return SameLevelChopAux(VR ∪ VL, WorkList)
end
```

Figure 6. Algorithms for the four chopping operations defined in Definition 2.2.

[1] $W := f_{unbr}(\{ s \}) \cap b_{unbl}(\{ t \})$

In other words, $W = \{ w \mid s \rightarrow^*_{unbr} w \rightarrow^*_{unbl} t \}$. The next step is to use W find the v's that appear in (‡):

[2] $VR := f_{unbr}(\{ s \}) \cap b_{unbr}(W)$
[3] $VL := f_{unbl}(W) \cap b_{unbl}(\{ t \})$

That is, $VR = \{ v \mid \exists\, w \in W \text{ such that } s \rightarrow^*_{unbr} v \rightarrow^*_{unbr} w \}$ and $VL = \{ v \mid \exists\, w \in W \text{ such that } w \rightarrow^*_{unbl} v \rightarrow^*_{unbl} t \}$. TruncatedChop returns the set $VR \cup VL$.

Because summary edges have been added to the SDG, the running times of f_{unbr}, f_{unbl}, b_{unbr}, and b_{unbl} are all linear in the size of the SDG augmented with summary edges. The number of edges in the SDG itself is bounded by $P \times E$, and the number of summary edges added to it is no more than $CSites \times Params^2$. Thus, the running time of TruncatedChop is bounded by

$$O((P \times E) + (CSites \times Params^2)).$$

We now turn to function Chop. A (non-truncated) unrestricted chop consists of the set of vertices $\{ v \mid \exists\, p \text{ such that } p = s \rightarrow^*_r t \text{ and } v \in p \}$. As with function SameLevelChop, this set can be obtained by repeatedly performing TruncatedSameLevelChop operations on certain pairs of {formal-in, enter}/formal-out vertices. In fact, auxiliary

function SameLevelChopAux performs exactly what is needed; for Chop, the initial vertex set passed to SameLevelChopAux is the same set $VR \cup VL$ computed by TruncatedChop(s, t), and the initial work-list consists of all summary edges along unbalanced-right paths from s to W or unbalanced-left paths from W to t. Because the version of SameLevelChopAux given in Figure 6 may perform a TruncatedSameLevelChop operation for each {formal-in, enter}/formal-out pair in a PDG of the SDG, the running time of Chop is bounded by

$$O((P \times E \times Params^2) + (CSites \times Params^4)).$$

3.5. A More Efficient Version of SameLevelChopAux

We now describe a version of SameLevelChopAux that is asymptotically more efficient than the one given in Figure 6. The improved version of SameLevelChopAux is presented in Figure 7. With this version of SameLevelChopAux, the asymptotic running times of both SameLevelChop and Chop are bounded by

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

The improvement to SameLevelChopAux is obtained by reducing the number of times each PDG is sliced with respect to a given formal-in, enter, or formal-out vertex. There are two key ideas behind the improved method:

(i) For a given vertex x of the SDG to be in the answer set, we need only find *one* {formal-in, enter}/formal-out pair (v', w') such that $x \in b_m(w')$, $x \in f_m(v')$, and both v' and w' are in the answer set. An invariant of the loop on lines [3]–[26] of Figure 7 is that for each summary edge (v, w) selected from the work-list, v and w are both known to be in the answer set. Consequently, at line [9], vertices v' and w' are both known to be in the answer set. The purpose of lines [10]–[16] is to generate sets of "candidate vertices" ($b_m(w')$ and $f_m(v')$) that may eventually end up in the final answer set.

If a given $x \in b_m(w')$ is not also in $f_m(v')$, it may still eventually end up in the answer set if there is some formal-in or enter vertex v'' for which $x \in f_m(v'')$. This will be discovered on a subsequent iteration of the main loop (lines [3]–[26]).

(ii) A slice with respect to a given formal-in, enter, or formal-out vertex is performed exactly *once*, rather than once for each {formal-in, enter}/formal-out pair. These results are saved and reused on subsequent iterations of the loop. (Lines [10] and [14] test to see whether the candidate sets already exist.)

As with the version of SameLevelChopAux given in Figure 6, because a summary edge (x, y) indicates that a same-level realizable path from x to y exists in the SDG, the main loop looks for summary edges (x, y) such that

```
       function SameLevelChopAux(Answer, WorkList) returns vertex set
       parameters
         Answer: vertex set
         WorkList: set of summary edges
       begin
[1]      Remove all marks on {formal-in, enter}/formal-out pairs
[2]      Set all b-candidates, b-actual-out, and f-candidates sets to null
[3]      while WorkList ≠ ∅ do
[4]        Select and remove a summary edge (v, w) from WorkList
[5]        let v′ = the formal-in or enter vertex that corresponds to v and
[6]            w′ = the formal-out vertex that corresponds to actual-out w
[7]        in
[8]          if (v′, w′) is unmarked then
[9]            Mark (v′, w′)
[10]           if b-candidates(w′) = null then
[11]             b-candidates(w′) = b_m(w′)
[12]             b-actual-out(w′) = b_m(w′) ∩ ActualOutVertices
[13]           fi
[14]           if f-candidates(v′) = null then
[15]             f-candidates(v′) = f_m(v′)
[16]           fi
[17]           for each x ∈ b-candidates(w′) do
[18]             if x ∈ f-candidates(v′) then
[19]               Answer := Answer ∪ { x }
[20]               Remove x from b-candidates(w′)
[21]               if x ∈ (ActualInVertices ∪ CallVertices) then
[22]                 for each y such that (x, y) ∈ SummaryEdges and y ∈ b-actual-out(w′) do
[23]                   WorkList := WorkList ∪ { (x, y) }
[24]                 od fi fi
[25]           od fi ni
[26]     od
[27]     return Answer
       end
```

Figure 7. A version of SameLevelChopAux that yields asymptotically more efficient versions of functions SameLevelChop and Chop.

$x \in f_m(v')$ and $y \in b_m(w')$. These are placed on the work-list for subsequent processing. (See lines [21]–[24] of Figure 7.)

The reason why this method yields an asymptotic improvement in the running time of SameLevelChopAux is as follows: A vertex x is removed from *b-candidates*(w') whenever it is placed in *Answer* (see lines [19]–[20]). For a given x, line [19] may be executed at most *Params* times—that is, at most once for each formal-out vertex of the PDG that contains x. This saves a factor of *Params* in the bound on the function's asymptotic running time: If line [20] were not performed, then line [19] could be executed up to $Params^2$ times—once for each {formal-in, enter}/formal-out pair.

The cost of the improved SameLevelChopAux is dominated by the cost of creating the "candidate sets" in lines [10]–[16] and the cost of adding summary edges to the work-list in lines [21]–[24]. To create the candidate sets, in each PDG an f_m operation may be performed for each formal-in or enter vertex, and a b_m operation may be performed for each formal-out vertex, so the total cost is bounded by

$$O(\sum_{p \in Procedures} Params \times (E + CSites_p \times Params^2))$$
$$= O((P \times E \times Params) + (CSites \times Params^3)).$$

To determine which summary edges should be added to the work-list, for each of the $CSites_p \times Params$ actual-in or call vertices x in some PDG p, line [21] may be executed at most *Params* times (once for each actual-out vertex of p). Because x may have at most *Params* outgoing summary edges, the total cost of lines [21]–[24] is bounded by

$$O(\sum_{p \in Procedures} Params \times CSites_p \times Params^2)$$
$$= O(CSites \times Params^3).$$

To summarize, with the improved version of SameLevelChopAux, the asymptotic bounds on the running times of our algorithms for the four varieties of interprocedural chopping problems are as follows:

	Truncated	Non-Truncated
Same-level chopping	$O(E + MaxSites \times Params^2)$	$O((P \times E \times Params) + (CSites \times Params^3))$
Unrestricted chopping	$O((P \times E) + (CSites \times Params^2))$	$O((P \times E \times Params) + (CSites \times Params^3))$

(In this table, we are assuming that the SDG's summary edges have already been determined.[4])

4. Conclusions

In this paper, we have used the SDG representation introduced by Horwitz et al. [13] to formulate and solve four varieties of interprocedural chopping. (We also rely on the improved method for determining an SDG's summary edges that was described by Reps et al. [28].) These chopping problems generalize the restricted version of interprocedural chopping studied by Jackson and Rollins (*i.e.*, the truncated same-level interprocedural chopping problem) [15].

Truncated same-level chopping of C programs has been implemented by Jackson and Rollins in their ChopShop system. Recently, we implemented all the algorithms from Section 3 (including the efficient version of SameLevelChopAux given in Figure 7) to create a tool that allows one to chop C programs via any of the four interprocedural-chopping operations defined in Section 2.3. (The implementation is based on an offshoot of the Wisconsin Program Integration System [26] that has been retargeted to operate on C programs.)

Our limited experience with chopping is in agreement with what is predicted by the asymptotic bounds on running times listed in the table given above.

- Truncated unrestricted chopping is not substantially more expensive to perform than interprocedural slicing.
  - From the standpoint of asymptotic complexity, this is expected because, once the set of all actual-in-to-actual-out summary edges has been determined, both slicing and chopping are linear in the size of the SDG augmented with summary edges.
  - The table given in Figure 8 reports the times required for three representative invocations of TruncatedChop(s, t) on the source code of the Unix utility *compress*.[5]
- Both of the non-truncated chopping problems are substantially more expensive to perform than their truncated counterparts, although in practice it may still be feasible for a system to support the non-truncated chopping operations.
  - The asymptotic running times of the algorithms given in Section 3 are of the same order as that of determining the set of summary edges of an SDG. Unfortunately, a computation with this expense is required with *each* non-truncated chop.
  - The table in Figure 8 also reports the times for invocations of Chop(s, t) on *compress*. For comparison purposes, the time required to determine the summary edges for *compress*'s SDG is 22.06u + 0.59s seconds.

With the less efficient version of SameLevelChopAux from Figure 6, the time for chopping can rise dramatically:

---

[4]The cost of finding the set of summary edges is $O((P \times E \times Params) + (CSites \times Params^3))$. This dominates the cost of truncated unrestricted chopping by a factor of *Params* and is the same as the cost of both non-truncated problems. However, as stated earlier, the step of determining the summary edges only has to be performed once, when the first chop is performed.

[5]*Compress* has 1,503 lines of code and its SDG contains 3,745 vertices. *P* is 17, *MaxSites* is 11, *CSites* is 28, *E* is 12,237, and *Params* is 66.

	Operation	# of Vertices in Answer	Time (user-time + system-time in seconds)
Test 1	TruncatedChop(s, t)	288	1.50u + 0.11s
	$f_r(s)$	3227	1.21u + 0.00s
	$b_r(t)$	2177	1.08u + 0.08s
	Chop(s, t)	1994	68.64u + 0.76s
Test 2	TruncatedChop(s, t)	779	2.96u + 0.00s
	$f_r(s)$	3310	2.28u + 0.02s
	$b_r(t)$	2225	1.77u + 0.00s
	Chop(s, t)	2043	70.12u + 0.16s
Test 3	TruncatedChop(s, t)	1111	2.87u + 0.02s
	$f_r(s)$	3042	1.49u + 0.01s
	$b_r(t)$	2234	1.45u + 0.34s
	Chop(s, t)	1998	70.30u + 0.48s

Figure 8. Times for three representative truncated and non-truncated unrestricted chops of the Unix utility *compress*. For comparison purposes, times for the forward slice $f_r(s)$ and the backward slice $b_r(t)$ are also reported for each test. These measurements were carried out on a Sun SPARCstation 20 Model 61 with 64 MB of RAM.

	Operation	# of Vertices in Answer	Time (user-time + system-time in seconds)
Test 4 (SameLevel-ChopAux from Figure 7)	Chop	1997	70.52u + 0.00s
Test 4′ (SameLevel-ChopAux from Figure 6)	Chop	1997	2315.21u + 0.10s

Jackson and Rollins refer to chopping as ". . . a generalization of slicing [in which] most slicing notions, such as backward and forward slicing, are easily expressed as special cases" [15]. Although strictly speaking, this is true—for example, a forward interprocedural slice with respect to vertex s is a (multi-target) chop with respect to source s and a target-set consisting of all vertices of the SDG—this gives a misleading impression that chopping somehow subsumes slicing. In fact, as is apparent from Section 3, the relationship is essentially the other way around: The primitive operations needed for interprocedural chopping are exactly the ones developed earlier for interprocedural slicing: an algorithm for obtaining an SDG's summary edges, and the operations f_m, f_{unbr}, f_{unbl}, b_m, b_{unbr}, and b_{unbl}. It would be silly to implement interprocedural slicing via an interprocedural-chopping algorithm, for slicing is the simpler operation!

We close by proposing four other variants of interprocedural chopping that may also be useful:

(i) A ***return-path interprocedural chop*** consists of the set of vertices

$$\{ v \mid \exists p \text{ such that } p = s \to^*_{unbr} t \text{ and } v \in p \}.$$

(ii) A ***truncated return-path interprocedural chop*** consists of the set of vertices

$$\{ v \mid s \to^*_{unbr} v \to^*_{unbr} t \}.$$

(iii) A ***call-path interprocedural chop*** consists of the set of vertices

$$\{ v \mid \exists p \text{ such that } p = s \to^*_{unbl} t \text{ and } v \in p \}.$$

(iv) A ***truncated call-path interprocedural chop*** consists of the set of vertices

$$\{ v \mid s \to^*_{unbl} v \to^*_{unbl} t \}.$$

Like the same-level chopping operations, the return-path and call-path chopping operations are more "focused" than their unrestricted counterparts. Return-path and call-path chops report only a subset of the vertices reported by the unrestricted chopping operations. They eliminate from consideration certain kinds of "effect-transmission" paths due to calling context (*e.g.*, when there is a procedure that calls both the procedure containing s and the procedure containing t). Unlike the same-level chopping operations, in return-path and call-path chops s and t are not required to occur in the same procedure.

The implementations of return-path and call-path chopping are similar to the implementations given in Section 3. Again, the non-truncated versions of the operations are likely to be significantly more expensive to perform than the truncated ones.

References

1. Agrawal, H., "On slicing programs with jump statements," *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation,* (Orlando, FL, June 22-24, 1994), *ACM SIGPLAN Notices* **29**(6) pp. 302-312 (June 1994).
2. Ball, T. and Horwitz, S., "Slicing programs with arbitrary control flow," pp. 206-222 in *Proceedings of the First International Workshop on Automated and Algorithmic Debugging,* (Linköping, Sweden, May 1993), *Lecture Notes in Computer Science,* Vol. 749, Springer-Verlag, New York, NY (1993).
3. Bannerjee, U., "Speedup of ordinary programs," Ph.D. dissertation and Tech. Rep. R-79-989, Dept. of Computer

Science, University of Illinois, Urbana, IL (October 1979).

4. 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).

5. Binkley, D., "Using semantic differencing to reduce the cost of regression testing," *Proceedings of the 1992 Conference on Software Maintenance* (Orlando, Florida, November 9-12, 1992), pp. 41-50 (1992).

6. Chase, D.R., Wegman, M., and Zadeck, F.K., "Analysis of pointers and structures," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 296-310 (June 1990).

7. Choi, J.-D. and Ferrante, J., "Static slicing in the presence of GOTO statements," *ACM Trans. Program. Lang. Syst.* **16**(4) pp. 1097-1113 (July 1994).

8. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* **9**(3) pp. 319-349 (July 1987).

9. 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).

10. Goff, G., Kennedy, K., and Tseng, C.-W., "Practical dependence testing," *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, Ontario, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 15-29 (June 1991).

11. Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 28-40 (July 1989).

12. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

13. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).

14. Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 234-245 (June 1990).

15. Jackson, D. and Rollins, E.J., "A new model of program dependences for reverse engineering," *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (New Orleans, LA, December 7-9, 1994), *ACM SIGSOFT Software Engineering Notes* **19** pp. 2-10 (December 1994).

16. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

17. Landi, W. and Ryder, B.G., "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 1991), ACM, New York, NY (1991).

18. Larus, J.R. and Hilfinger, P.N., "Detecting conflicts between structure accesses," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 21-34 (July 1988).

19. Lyle, J. and Weiser, M., "Experiments on slicing-based debugging tools," in *Proceedings of the First Conference on Empirical Studies of Programming,* (June 1986), Ablex Publishing Co. (1986).

20. Markosian, L., Newcomb, P., Brand, R., Burson, S., and Kitzmiller, T., "Using an enabling technology," *Commun. of the ACM* **37**(5) pp. 58-70 (May 1994).

21. Maydan, D.E., Hennessy, J.L., and Lam, M.S., "Efficient and exact data dependence analysis," *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, Ontario, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 1-14 (June 1991).

22. Ning, J.Q., Engberts, A., and Kozaczynski, W., "Automated support for legacy code understanding," *Commun. of the ACM* **37**(5) pp. 50-57 (May 1994).

23. 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, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

24. Pugh, W., "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Supercomputing 1991*, (November 1991).

25. Pugh, W. and Wonnacott, D., "Eliminating false data dependences using the omega test," *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation,* (San Francisco, CA, June 17-19, 1992), *ACM SIGPLAN Notices* **27**(7) pp. 140-151 (July 1992).

26. Reps, T., "Demonstration of a prototype tool for program integration," TR-819, Computer Sciences Department, University of Wisconsin, Madison, WI (January 1989).

27. Reps, T., Sagiv, M., and Horwitz, S., "Interprocedural dataflow analysis via graph reachability," TR 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (April 1994). (Available on the World Wide Web at ftp://ftp.diku.dk/diku/semantics/papers/D-215.ps.Z.)

28. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (New Orleans, LA, December 7-9, 1994), *ACM SIGSOFT Software Engineering Notes* **19**(5) pp. 11-20 (December 1994).

29. Reps, T., Horwitz, S., and Sagiv, M., "Precise interprocedural dataflow analysis via graph reachability," pp. 49-61 in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).

30. 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).

31. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

32. Wolfe, M.J., "Optimizing supercompilers for supercomputers," Ph.D. dissertation and Tech. Rep. R-82-1105, Dept. of Computer Science, University of Illinois, Urbana, IL (October 1982).