***Proceedings of the International Conference on Software Maintenance (ICSM '01), IEEE Copyright***

# Incremental Slicing Based on Data-Dependences Types

Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold
College of Computing, Georgia Institute of Technology
{orso,sinha,harrold}@cc.gatech.edu

## Abstract

*Program slicing is useful for assisting with many software-maintenance tasks. The presence and frequent usage of pointers in languages such as C causes complex data dependences. To function effectively on such programs, slicing techniques must account for pointer-induced data dependences. Existing slicing techniques do not distinguish data dependences based on their types. This paper presents a new slicing technique, in which slices are computed based on types of data dependences. This new slicing technique offers several benefits and can be exploited in different ways, such as identifying subtle data dependences for debugging, computing reduced-size slices quickly for complex programs, and performing incremental slicing. This paper describes an algorithm for incremental slicing that increases the scope of a slice in steps, by incorporating different types of data dependences at each step. The paper also presents empirical results to illustrate the performance of the technique in practice. The results illustrate that incremental slices can be significantly smaller than complete slices. Finally, the paper presents a case study that explores the usefulness of incremental slicing for debugging.*

## 1 Introduction

A *program slice* of a program $\mathcal{P}$, computed with respect to a *slicing criterion* $< s, V >$, where $s$ is a program point and $V$ is a set of program variables, includes statements in $\mathcal{P}$ that may influence, or be influenced by, the values of the variables in $V$ at $s$ [24]. A program slice identifies statements that are related to the slicing criterion through transitive data and control dependences. A slice provides information that is useful for several software-maintenance tasks, such as identifying the cause of a software failure, evaluating the effects of proposed modifications to the software, and determining parts of the software that should be retested after modifications. To function effectively on programs that use pointers, slicing techniques must accommodate the effects of these pointers on data-dependence relationships; pointers are used frequently in programs written in widely-used languages such as C. The presence of pointers causes subtle data dependences in these programs. Much research has addressed the problem of computing slices in the presence of pointers (e.g., [1, 3, 5, 15]). None of that research, however, has considered classifying data dependences into different types and investigating how these types affect the computation of slices. In recent work, we introduced a fine-grained classification of data dependences that arise in the presence of pointers [17]. The classification distinguishes data dependence based on their "strength" and on the certainty with which the data dependences occur along various execution paths. We also presented empirical results that illustrate the distribution of such dependences for a set of C subjects and introduced a new slicing paradigm that computes slices based on types of data dependences. The main benefit of this paradigm is that it lets us compute slices by considering only a subset of the types of data dependences. By ignoring certain data dependences, the paradigm provides a way of reducing the sizes of slices, thus making the slices more manageable and usable.

In this paper, we illustrate a technique for incremental slicing, based on the new slicing paradigm. We present an algorithm that computes a slice in several steps, by incorporating additional types of data dependences at each step. This technique can be useful for improving the effectiveness of various applications of slicing. For example, consider the use of slicing for program comprehension. When we are trying to understand just the overall structure of the program, we can ignore weak data dependences and focus on only stronger data-dependence types. To do this, we can use the incremental slicing technique to start the analysis by considering only strong data dependences, and then augment the slice incrementally by incorporating additional weaker data dependences. This approach lets us focus initially on a smaller, and thus easier to understand, subset of the program and then consider increasingly larger parts of the program.

Alternatively, for applications such as debugging, we may want to start focusing on weak, and therefore not obvious, data dependences. By doing this, we can identify subtle pointer-related dependences that may cause unforeseen be-

```
   int i, j;
   main() {
     int j, sum;
 1   read i;
 2   read j;
 3   sum = 0;
 4   while ( i < 10 ) {
 5     sum = add( sum );
     }
 6   print sum;
   }
```

```
   int add( int sum ) {
 7  if ( sum > 100 ) {
 8    i = 9;
    }
 9  sum = sum + j;
10  read j;
11  i = i + 1;
12  return sum;  }
```

**Figure 1.** Program `Sum1`.

havior in the program.

Interprocedural slicing techniques based on the system-dependence graph (SDG) [11, 22] and data-flow equations [10, 24] form two alternative, general classes of slicing techniques. To compute incremental slices, we extend the SDG-based approach for slicing: we extend the SDG to accommodate data-dependence types, and we modify the SDG-based slicing algorithm to compute summary information for procedures while slicing.

In this paper, we also present empirical results that illustrate the performance of the incremental slicing technique in practice. We also present a case study in which we investigate the usefulness of the incremental slicing technique for debugging.

The main contributions of the paper are:

- An incremental slicing technique that computes a slice in steps, by incorporating additional types of data dependences at each step.
- Empirical studies that illustrate the performance of the incremental slicing technique in practice.
- A case study that illustrates the use of the incremental slicing technique for debugging.

## 2 Background

### 2.1 Program slicing

For this work, we extend the SDG-based slicing approach [11, 22]; the approach based on data-flow equations [10, 24] could be extended similarly.

A *system-dependence graph* (SDG) [11] is a collection of *procedure-dependence graphs* (PDGs) [7]—one for each procedure—in which nodes are statements or predicate expressions. *Data-dependence edges* represent flow of data between statements or expressions; *control-dependence edges* represent control conditions on which the execution of a statement or expression depends. A *data dependence* is a triple $(d, u, v)$ where $d$ and $u$ are statements and $v$ is a variable, $d$ defines $v$, $u$ uses $v$, and there exists a path from $d$ to $u$ along which $v$ is not redefined. For example, in program `Sum1` (shown in Figure 1), (1, 4, `i`) is a data dependence.

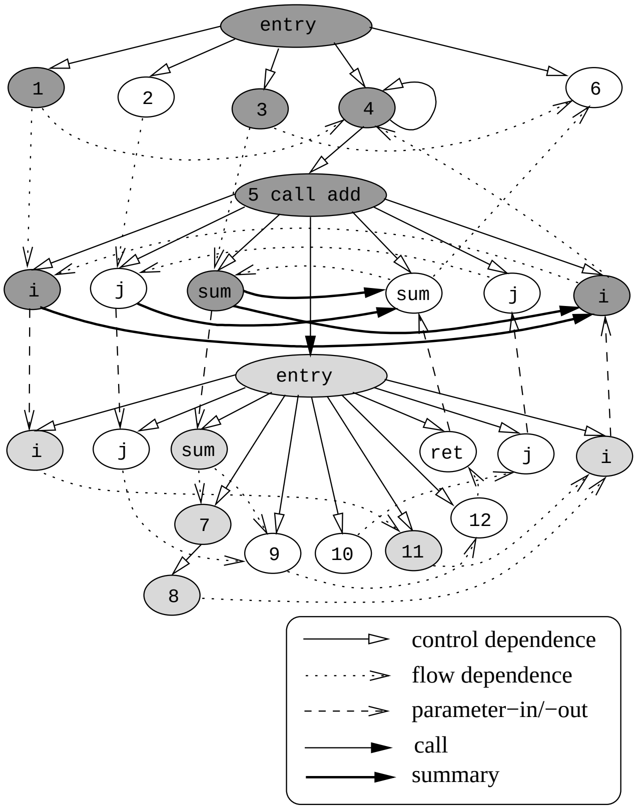

**Figure 2.** System-dependence graph for `Sum1`.

Each PDG contains an *entry node* that represents entry into the procedure. To model parameter passing,[1] an SDG associates formal parameter nodes with each procedure-entry node: a *formal-in node* is created for each formal parameter that may be referenced or modified by the procedure; a *formal-out node* is created for each formal parameter that may be modified [13] by the procedure and for the return value of a function. An SDG associates a *call node* and a set of actual parameter nodes with each call site in a procedure: an *actual-in node* for each actual parameter at the call site that may be referenced or modified by the called procedure; an *actual-out node* for each actual parameter that may be modified by the called procedure and for the return value if it is assigned to a variable at the call site.

An SDG connects PDGs at call sites. A *call edge* connects a call node to the entry node of the called procedure's PDG. Parameter-in and parameter-out edges represent parameter passing: *parameter-in edges* connect actual-in and formal-in nodes, and *parameter-out edges* connect formal-out and actual-out nodes.

Figure 2 shows the SDG for program `Sum1`. In the figure, ellipses represent program statements (labeled by statement numbers), parameter nodes, entry points, and call sites; various types of edges (shown by the key) represent

[1] Global variables are treated as parameters.

```
    int i;
    main() {                          int add( int val,
      int *p;                                  int sum ) {
      int j, sum1, sum2;                int *q, k;
  1   sum1 = 0;                     12  read k;
  2   sum2 = 0;                     13  if ( sum > 100 ) {
  3   read i, j;                    14    i = 9;
  4   while ( i < 10 ) {                }
  5     if ( j < 0 ) {              15  sum = sum + i;
  6       p = &sum1;                16  if ( i < k ) {
        }                           17    q = &val;
        else {                          }
  7       p = &sum2;                    else {
        }                           18    q = &k;
  8     *p = add( j, *p );              }
  9     read j;                     19  sum = sum + *q;
      }                             20  i = i + 1;
 10   sum1 = add( j, sum1 );        21  return sum;
 11   print sum1, sum2;               }
    }
```

**Figure 3.** Program Sum2.

dependences and bindings. The parameter nodes are labeled by the variable name to which they correspond; the formal-out node for the return value is labeled ‘ret’.

Horwitz, Reps, and Binkley [11] compute interprocedural slices by solving a graph-reachability problem on an SDG. To restrict the computation of interprocedural slices to paths that correspond to legal call/return sequences, an SDG uses summary edges to represent the transitive flow of dependence across call sites caused by data dependences or control dependences. A *summary edge* connects an actual-in node and an actual-out node if the value associated with the actual-in node may affect the value associated with the actual-out node.

The interprocedural backward slicing algorithm consists of two phases. The first phase traverses backwards from the node in the SDG that represents the slicing criterion along all edges except parameter-out edges, and marks those nodes that are reached. The second phase traverses backwards from all nodes marked during the first phase along all edges except call and parameter-in edges, and marks reached nodes. The slice is the union of the marked nodes.

For example, consider the computation of a slice for $< 5, \{\texttt{i}\} >$; this slicing criterion is represented in the SDG of Figure 2 by the actual-out node for i at the call site to add. In its first phase, the algorithm adds to the slice the nodes shaded darkly in Figure 2. In its second phase, the algorithm adds to the slice the nodes that are reachable (along a backward traversal) from those added in the first phase; these nodes are shaded lightly in the figure.

## 2.2 Data dependences in the presence of pointers

**Table 1.** Classification of data dependences [17].

| | Ddef–<br>Duse | Ddef–<br>Puse | Pdef–<br>Duse | Pdef–<br>Puse |
|---|---|---|---|---|
| DRD-K | type 1 | type 7 | type 13 | type 19 |
| DPRD-K | type 2 | type 8 | type 14 | type 20 |
| DRD+K | type 3 | type 9 | type 15 | type 21 |
| DPRD+K | type 4 | type 10 | type 16 | type 22 |
| PRD-K | type 5 | type 11 | type 17 | type 23 |
| PRD+K | type 6 | type 12 | type 18 | type 24 |

The presence of pointer dereferences gives rise to complex data dependences in a program. In such programs, when analyzing a definition or a use, it may not be possible to identify unambiguously the variable that is defined or used; such a definition or use could modify or use one of several variables. For example, consider program Sum2 (Figure 3).[2] The definition in statement 8 can modify either sum1 or sum2 depending on how the predicate in statement 5 evaluates. To distinguish such definitions from those in which the variable can be identified unambiguously—for example, the definition of sum1 in statement 1—we classify the definition of sum1 (and of Sum2) in statement 8 as a *possible definition* and the definition of Sum1 in statement 1 as a *definite definition*. Like definitions, we also classify uses as definite or possible. Finally, based on the occurrence of types of the definitions along a path, we classify paths from the definition to the use into one of six types.

Table 1 lists the types of data dependences that result from combining the types of definitions, uses, and paths [17]. The first column in the table lists the types of paths; these paths are distinguished based on whether they contain definite or possible redefinitions of the relevant variable. The naming convention for the paths reflects the types of redefinitions that occur along the paths. The letters preceding “RD” in the name indicate the type of reaching definition: a “D” indicates a definite reaching definition, whereas a “P” indicates a possible reaching definition. A “+K” or a “-K” indicates the presence or absence, respectively, of a definite kill of the relevant variable. For example, the set of paths $\Pi$ from a definition of variable $v$ to a use of $v$ is classified as DPRD+K if (1) at least one path in $\Pi$ contains no definite redefinition of $v$, (2) at least one path in $\Pi$ contains a possible redefinition of $v$, and (3) and at least one path in $\Pi$ contains a definite redefinition of $v$. For further example, $\Pi$ is is classified as DPRD-K if previously defined conditions (1) and (2) hold, but condition (3) does not. To illustrate some of the types of data dependences that occur in Sum2, (1, 8, sum1) is a type 8 data dependence: node 1 contains a definite definition of sum1; node 8 contains a possible use of sum1; the set of paths from node 1 to 8 is DPRD-K because the set includes a path that satisfies condition (1) (this path does not iterate the loop in statement 4) and a path that satisfies condition (2) (this path iterates the loop in state-

[2] Sum2 is an extension of Sum1 with the addition of pointers; it is overly complicated to illustrate our technique and the complex dependences that can be introduced by pointers.

ment 4 at least once). Similarly, (8, 11, sum2) is a type 14 data dependence. Reference [17] provides additional details of the classification scheme and describes the algorithm for computing the types of data dependences.

# 3 Incremental Slicing in the Presence of Pointers

The classification of data dependences into distinct types leads to a new slicing paradigm, in which only statements that are related to the slicing criterion by one or more specified types of data dependences are included in the slice. In the next subsection, we describe this paradigm. Based on the new paradigm, we then describe an incremental slicing technique for computing a slice.

## 3.1 New slicing paradigm

Traditional slicing techniques (e.g., [10, 11, 24]) include in the slice all statements that affect the slicing criterion through direct or transitive control and data dependences. Such techniques compute a slice by computing the transitive closure of all control dependences and all data dependences starting at the slicing criterion. The classification of data dependences into different types leads to a new paradigm for slicing, in which the transitive closure is performed over only the specified types of data dependences, rather than over all data dependences. In this slicing paradigm, a *slicing criterion* is a triple $< s, V, T >$, where $s$ is a program point, $V$ is a set of program variables referenced at $s$, and $T$ is a set of data-dependence types. A *program slice* contains those statements that may affect, or be affected by, the values of the variables in $V$ at $s$ through transitive control or specified types of data dependences.

To compute slices in the new paradigm using the SDG-based approach, we extend the SDG in two ways. The first extension consists of annotating each data-dependence edge with the type of the corresponding data dependence. The traditional SDG does not distinguish data dependences based on their types and, therefore, does not contain such annotations. To illustrate, Figure 4 presents the SDG for Sum2. Each data-dependence edge in the figure is labeled with the type of that data dependence. For example, the data-dependence edge from node 1 to the actual-in node for *p at call node 8 is labeled 't8'; similarly, the data-dependence edge from the actual-out node for *p at that call node to node 11 is labeled 't14'.

Because the SDG introduces placeholder definitions and uses at formal-in and formal-out nodes, the data-dependence edges that are incident from, or incident to, such nodes have placeholder definition and use types associated with them; such definitions or uses are always definite. In Figure 4, such data-dependence edges—whose source contains a placeholder definition type or whose target contains a placeholder use type—are distinguished. For example, the data-dependence edge from the formal-in node for sum at call node 8 to node 13 has a placeholder definition type associated with it.

The second extension consists of associating a type with each summary edge. The traditional SDG does not associate types with summary edges. Because data-dependence edges have types associated with them, the summary edges computed using those data dependences also have types associated with them—these types are the types of data dependences that are followed while computing the summary edges. For example, the SDG in Figure 4 contains the summary edges that are created by traversing only type 1 data dependences; thus, the summary edges have the same type associated with them and are labeled 't1' in the figure. Associating data-dependence types with summary edges lets us use the two-phase slicing algorithm [11] with minimal changes.

To compute a slice for criterion $< s, V, T >$, the SDG must contain summary edges for data-dependence types $T$. After the summary edges are computed, the slicing algorithm proceeds like the two-phase slicing algorithm [11]. During the first phase, the algorithm traverses backward along control, flow, call, parameter-in, and summary edges. During the second phase, the algorithm traverses backward along control, flow, parameter-out, and summary edges. However, the algorithm traverses backward along a flow-dependence or a summary edge only if the data-dependence types associated with that edge appear in the set $T$ of data-dependence types mentioned in the slicing criterion. If an edge has placeholder definition or use type associated with it, the algorithm extracts the other components of the data-dependence type—for example, in case of a placeholder definition, the algorithm extracts the use type and the path type—and matches them with the types specified in the slicing criterion.

The nodes included in the slice for criterion $< 10, \{\texttt{sum1}\}, \{t1\} >$ are shaded lightly in Figure 4.

## 3.2 Incremental slicing technique

Using the new slicing paradigm, we define an incremental slicing technique. The *incremental slicing technique* computes a slice in steps, by incorporating additional types of data dependences at each step; the technique thus increases the scope of a slice in an incremental manner. In a typical usage scenario, the technique starts by considering the stronger types of data dependences and computes a slice based on those data dependences. Then, it increments the slice by considering additional, weaker data dependences and adding to the slice statements that affect the criterion through the weaker data dependences.

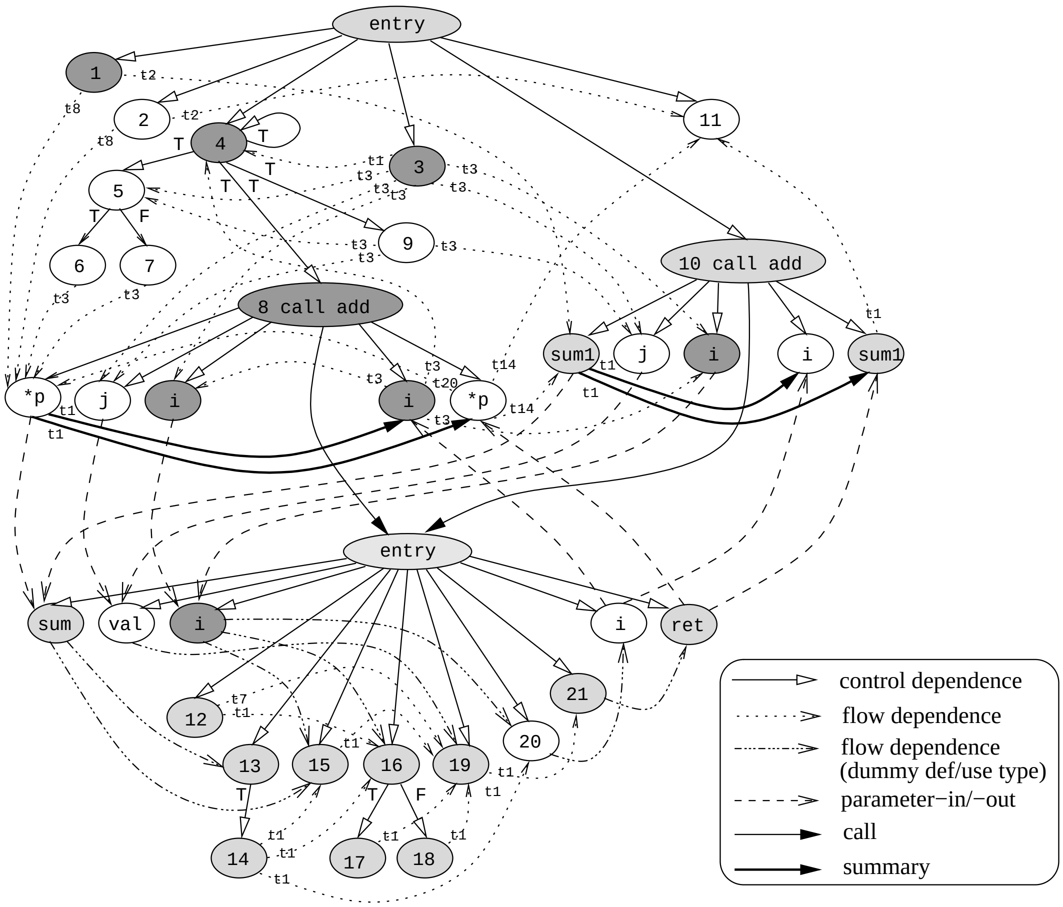

**Figure 4.** System-dependence graph for Sum2 to support slicing using types of data dependences. The lightly shaded nodes are included in the slice for $< 10, \{\texttt{sum1}\}, \{t1\} >$. The darkly shaded nodes are the additional nodes included in the slice for $< 10, \{\texttt{sum1}\}, \{t1, t2, t3\} >$.

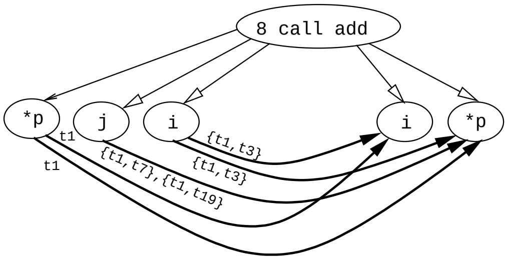

**Figure 5.** Summary edges, with the associated data-dependence types, required at call node 8 in the SDG for Sum2.

For example, the lightly shaded nodes in Figure 4 are included in the slice for $< 10, \{\texttt{sum1}\}, \{t1\} >$. Using the incremental technique, when type 2 and type 3 data dependences are also considered, the darkly shaded nodes are added to the slice. Figure 5 shows all summary edges that are required at call node 8.

To support incremental slicing using the SDG, we need to compute summary edges not only for the different types of data dependences (24 in our classification scheme), but also for the various combinations of data-dependence types ($2^{24} - 1$ combinations in our scheme). Clearly, it is not practical to precompute summary edges for all possible combinations of data-dependence types, even if all the data-dependence types do not occur in programs. An alternative is to compute the summary edges on demand, as and when they are required while computing a slice. Although, in the worst case, the demand approach can compute as many summary edges as the precompute approach, the results of our experiments indicate that this may not occur in practice; we discuss this aspect further in Section 4.

We modified the SDG-based slicing algorithm to compute summary edges on demand; Figure 6 presents the modified algorithm, `ComputeSlice`.

Like Horwitz, Reps, and Binkley's slicing algorithm, `ComputeSlice` proceeds in two phases. To identify

```
algorithm ComputeSlice
 input    $< s, V, T >$   slicing criterion
 output   $slice$         slice for $< s, V, T >$
 global   $G$             SDG for program $\mathcal{P}$
begin ComputeSlice
1.  $n =$ node in $G$ corresponding to $s$
2.  $slice$ = GetReachableNodes( $\{n\}$, $T$, {call, param-in} )
3.  $slice$ = GetReachableNodes( $slice, T$, {param-out} )
4.  return $slice$
end ComputeSlice

function GetReachableNodes
 input    $N$           set of SDG nodes
          $T$           data-dependence types
          $edgeTypes$   interprocedural edges to follow
 output   $slice$       nodes for a slicing phase
 declare  $worklist$    nodes traversed in the SDG
begin GetReachableNodes
 5.  $worklist$ = $N$; $slice = \phi$
 6.  while $worklist \neq \phi$ do
 7.      remove node $n$ from $worklist$
 8.      case $n$ is actual-out node:
 9.          foreach $m$ s.t. $(m \rightarrow n) \in$ control do
10.              add $m$ to $worklist$ and $slice$
11.          endfor
12.          if $\nexists m$ s.t $(m \rightarrow n) \in$ summary($T$) then
13.              $x$ = formal-out node corresponding to $n$
14.              ComputeSummaryEdges( $x, T$ )
15.          endif
16.          foreach $m$ s.t. $(m \rightarrow n) \in$ summary($T$) do
17.              add $m$ to $worklist$ and $slice$
18.          endfor
19.      default:
20.          foreach $m$ s.t. $(m \rightarrow n) \in$ fbw($T$), control,
             $edgeTypes$ do
21.              add $m$ to $worklist$ and $slice$
22.          endfor
23.      endcase
24.  endwhile
25.  return $slice$
end GetReachableNodes
```

**Figure 6.** Algorithm for computing a slice that computes summary edges on demand.

nodes that are included in the slice in each phase, ComputeSlice calls function GetReachableNodes() (lines 2, 3). During the first call, GetReachableNodes() computes reachability starting at the slicing criterion. During the second call, GetReachableNodes() computes reachability starting at the nodes traversed during the first call.

GetReachableNodes() uses a worklist to traverse backward along matching flow-dependence edges, control edges, and the specified types of interprocedural edges: call and parameter-in edges during the first phase, and parameter-out edges during the second phase (lines 19–22). On reaching an actual-out node $n$ (line 8), first, the function traverses backward along control edges incident on $n$ (lines 9–11). Next, the algorithm checks whether summary edges for the relevant data-dependence types $T$ have been computed for $n$ (line 12). If the summary edges for $T$ have not been computed, the function computes them on demand, starting at the formal-out node that is connected to $n$ (lines 13–14). After computing the summary edges, the function traverses backward along those edges (lines 16–18).

```
function ComputeSummaryEdges
 input    $x$          a formal-out node in $G$
          $T$          data-dependence types
 declare  $worklist$   nodes traversed in the SDG
begin ComputeSummaryEdges
 1.  $worklist$ = $x$
 2.  while $worklist \neq \phi$ do
 3.      remove node $n$ from $worklist$
 4.      case $n$ is actual-out node:
 5.          foreach $m$ s.t. $(m \rightarrow n) \in$ control do
 6.              add $m$ to $worklist$
 7.          endfor
 8.          if $\nexists m$ s.t $(m \rightarrow n) \in$ summary($T$) then
 9.              $x$ = formal-out node corresponding to $n$
10.              ComputeSummaryEdges( $x, T$ )
11.          endif
12.          foreach $m$ s.t. $(m \rightarrow n) \in$ summary($T$) do
13.              add $m$ to $worklist$
14.          endfor
15.      case $n$ is formal-in node:
16.          foreach $m$ s.t $(m \rightarrow n) \in$ param-in edge do
17.              $y$ = corresponding actual-out
18.              create summary($T$) $(m \rightarrow y)$
19.          endfor
20.      default:
21.          foreach $m$ s.t. $(m \rightarrow n) \in$ fbw($T$), control do
22.              add $m$ to $worklist$
23.          endfor
24.      endcase
25.  endwhile
end ComputeSummaryEdges
```

**Figure 7.** Algorithm for computing summary edges on demand.

ComputeSummaryEdges(), shown in Figure 7, is a recursive function that takes as inputs a formal-out node $x$ and a set of data-dependence types $T$. It identifies those formal-in nodes (in the PDG that contains $x$) that are reachable from $x$ backward along control edges and flow edges of type $T$ (lines 20–23). On reaching an actual-out node (line 4), the function performs actions similar to GetReachableNodes(): it traverses backward along control edges (lines 5–7) and, if required, invokes itself recursively to compute summary edges for type $T$ (lines 8–10), and then traverses along those edges (lines 12–14). On reaching a formal-in node $n$ (line 15), the function has identified a summary dependence from $n$ to $x$. Therefore, the function ascends along each parameter-in edge connected to $n$, and creates a summary edge for type $T$ at each such call site (lines 16–19).

ComputeSlice involves two traversals of the SDG, both of which are linear in the size of the SDG; therefore, the time complexity of ComputeSlice is linear in the size

**Table 2.** Programs used for the empirical studies.

| Subject | Description | LOC |
|---|---|---|
| armenu | Aristotle Analysis system interface | 11320 |
| dejavu | Regression test selector | 3166 |
| lharc | Compress/extract utility | 2550 |
| replace | Search-and-replace utility | 551 |
| space[3] | Parser for antenna array description language | 6201 |
| tot_info | Statistical information combiner | 477 |
| unzip | Compress/extract utility | 2906 |

**Table 3.** Sets of slices computed for each subject. Each set contains slices that were computed starting at each program statement and corresponding to the specified data-dependence types.

| Subject | Slices Computed |
|---|---|
| armenu | S1{t1} S2{t1–t3} S3{t1–t24} |
| dejavu | S1{t1} S2{t1–t3} S3{t1–t19} S4{t1–t20} **S5**{t1–t24} |
| lharc | S1{t1} S2{t1–t3} S3{t1–t19} S4{t1–t20} **S5**{t1–t24} |
| replace | S1{t1} S2{t1–t3} S3{t1–t20} **S4**{t1–t24} |
| space | S1{t1} S2{t1–t3} S3{t1–t19} S4{t1–t20} **S5**{t1–t24} |
| tot_info | S1{t1} S2{t1–t3} S3{t1–t14} S4{t1–t24} |
| unzip | S1{t1} S2{t1–t3} S3{t1–t20} **S4**{t1–t24} |

of the SDG. The more significant cost element is the space complexity because of the exponential worst-case theoretical bound on the number of summary edges. In practice, we do not expect this worst case to occur. However, if this cost is realized, it makes the storage of summary edges infeasible. A more practical approach would be to discard and recompute the summary edges for each slice. Such an approach retains the linear time complexity of `ComputeSlice`, while avoiding its exponential space complexity. This approach trades off the exponential space complexity of the algorithm for the number of slices that may need to be computed: in the worst case, a slice for each combination of the data-dependence types may be computed. Our empirical evidence, presented in the next section, suggests that considering all combinations of data-dependence types is not required in practice.

## 4 Empirical Results

To investigate the performance of the incremental slicing technique in practice, we implemented a prototype and performed empirical studies with a set of C subjects. We implemented the reaching-definitions algorithm, the SDG construction algorithm, and the SDG-based slicing algorithm using the ARISTOTLE analysis system [2]. To account for the effects of aliases, we replaced the ARISTOTLE front-end with the PROLANGS Analysis Framework (PAF) [19]. We used PAF to gather control-flow, local data-flow, alias, and symbol-table information; we then used this information to interface with the ARISTOTLE tools. We used the programs listed in Table 2 for the empirical studies. The size of the programs that we considered is constrained by the computational cost of the flow- and context-sensitive alias analysis [12] that we used.

The goal of the study was to evaluate how the sizes of slices increase as additional types of data dependences are considered during the computation of the slices. For each subject, we computed intraprocedural data dependences, and classified them into the types listed in Table 1. Then, based on the distribution of the data-dependence types, we selected the slices to compute; Table 3 lists the slices that were computed for each subject. The table lists the data-dependence types that we considered when computing slices for the subjects. For example, for `unzip`, we computed four sets of slices: S1, S2, S3, and S4. The slices in the first set were based only on data-dependence type 1, whereas those in the second, third, and fourth sets were based on data-dependence types 1 through 3, 1 through 20, and 1 through 24, respectively. The last set of slices—those based on all types of data dependences—are the same that would be computed by a slicer that ignores the distinctions among data dependences. The data-dependence types for each successive set of slices are inclusive of the types for the previous set of slices. Thus, each slice from a set is a superset of the corresponding slice from the previous set, which lets us study the growth in the sizes of the slices.

We envision that such usage of the incremental slicing technique would be typical and the most beneficial when the technique is incorporated into a software-maintenance tool. The tool would present the distribution of the data-dependence types, whose computation is far less expensive than the computation of slices, to the user. The user then, based on the distribution, would select one or more set of data-dependence types to consider for the computation of the slices.

The data distribution that we observed for our subjects suggests that the worst-case exponential complexity of the summary-edge computation may not occur in practice [17]. For all subjects, only four or five types of data dependences appeared predominantly. Although other types of data dependences occurred, they were insignificant in number. Therefore, for each subject, we identified only three to five meaningful sets of data-dependence types for which to compute slices—those for which we expected significant

[3] Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided `Space` and many of its test cases. Chengyun Chu assisted with further preparation of the Space program and development of its test cases.

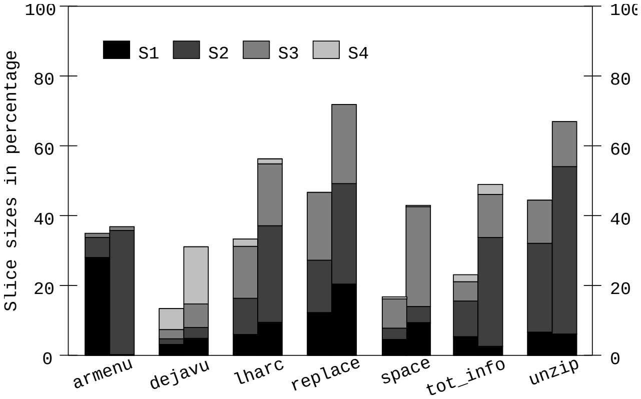

**Figure 8.** Increase in the sizes of the slices for the slice sets listed in Table 3. The first segmented bar illustrates the average increase in the slice sizes from one set to the next. The second segmented bar illustrates the increase in the size of the slice that exhibited the maximum growth from the first set to the final set.

differences in the slices. Moreover, we observed that, for our subjects, the occurrence of different types of data dependences is independent of the program size. Therefore, we expect such a trend to occur in larger programs, and consider unlikely for all data-dependence types to occur in equally significant numbers, even in large programs.

For each subject, we computed slices starting at each program statement (for variables that are used at that statement)—one such set of slices for each combination of data-dependence types listed in Table 3. Figure 8 presents data about the increase in the sizes of the slices. The vertical axis represents the sizes of the slices as percentages of the number of statements in the program. The figure contains two segmented bars for each subject. The first segmented bar in Figure 8 illustrates the average increase in the slice sizes; each segment shows the average increment in the slice size for a set over the previous set. For example, consider the first segmented bar for `lharc`. The average size of the slices in set S1, which were computed for data-dependence type 1, is 6% of the program size. The slices in set S2 were computed for data-dependence types 1 through 3. On average, the slices in S2 are larger than the slices in S1 by 10% of the program statements; therefore, the average size of the slices in S2 is 16% of the program size. Similarly, the slices in set S3, which were computed for data-dependence types 1 through 19, include on average an additional 15% of the program statements; the average size of the slices in S3 is thus 31% of the program size. The average size of the slices increases by another 2% (of the program size) when data-dependence type 20 is also considered—this is illustrated by the segment for set S4 in the figure. The slices in the final set for `lharc`—set S5—were computed using all data-dependence types. However, the slices in this set showed no increase in their sizes compared with the slices in S4;[4] therefore, the segmented bar for `lharc` has no segment for set S5.

The second segmented bar in Figure 8 illustrates the increase in the size of the slice that exhibited the maximum growth from the first set to the final set; each segment illustrates the increase in the size of this slice for a set over the previous set. For example, for `lharc`, the slice that showed that maximum growth from the first set to the final set included an additional 47% of the program statements in the final set than in the first set. This slice grows from 10% of the program size in set S1 to 38% in set S2, 56% in set S3, and finally to 57% in set S4.

The increases in the sizes of the slices vary across the subjects as additional data-dependence types are considered. For example, on average, the slice sizes for `dejavu` increase by only 2% of the program size when data-dependence types 2 and 3 and considered in addition to data-dependence type 1; however, for `unzip`, the slice sizes for the same types increase by over 25%. The addition of data-dependences caused by pointers, which occurs starting in slice set S3 for each subject, also causes increases in slice sizes that vary across subjects. On average, pointer-related data-dependences cause the slices to increase by only a little over 1% for `armenu`, 9% for `space`, but 17% for `lharc` and over 19% for `replace`. As stated above, the second segmented bar illustrates growth in the slice size for the slice that had the largest increase caused by the additional data-dependence types for each subject. Such slices grew by over 60% of the program size from S1 to S3 for `unzip`, and by over 51% from S1 to S3 for `replace`. The data in Figures 8 illustrate the usefulness of the new slicing paradigm in controlling the sizes of the slices.

## 5 Application of the Incremental Slicing Technique

We performed a case study to investigate the usefulness of incremental slices for debugging. The goal of the study was to determine how incremental approximations to dynamic slices succeed in isolating a fault. For the study, we chose a version of `space` that contains a known fault, and a test case that exposed the fault. To simulate a typical debugging scenario, we examined the output of the fault-revealing test case and selected a suitable slicing criterion at an output statement in the program. Next, we examined the distribution of data-dependence types for `space` and, based on the occurrences of various types, selected eight combinations of data-dependence types for computing the slices: {t1}, {t1–t3}, {t1–t7}, {t1–t13}, {t1–t14}, {t1–t19}, {t1–t20},

[4] This occurs because the statements related through the additional types of data-dependences were already included in the slices in the previous set through control dependences or other types of data dependences.

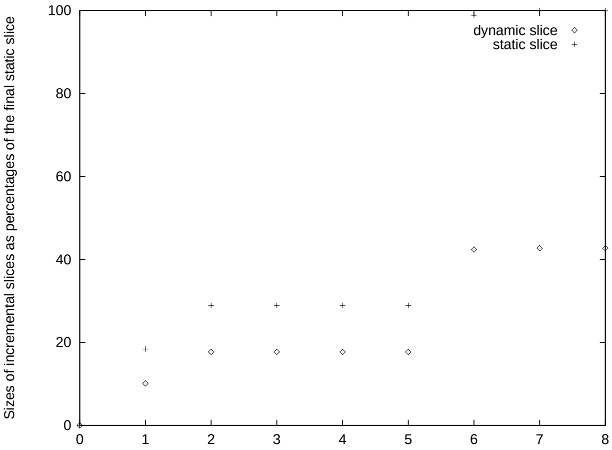

**Figure 9.** Eight incremental static and dynamic slices computed for `space` to locate the fault.

and {t1–t24}. Using these types, we computed incremental slices and intersected them with the statement trace of the fault-revealing test case to obtain approximations to the corresponding dynamic slices.

Figure 9 presents a plot of the sizes of the eight static and dynamic slices computed for `space`; the figure shows the sizes of the slices as percentages of the size of the last static slice. The sixth incremental slice, which was computed for data-dependence types 1 through 19, contains the fault. The static-slice increment that contains the fault is 70% of the final static slice, whereas the dynamic-slice increment that contains the fault is 25% of the final static slice. The distribution of data-dependence types for `space` were such that inclusion of the additional data dependences in the third, fourth, and fifth slices caused no increases in the slice sizes, whereas the additional data dependences in the sixth slice caused a significant increase in the slice size. It is difficult to speculate whether such behavior would persist across larger and more varied subjects. Nonetheless, the case study does suggest the benefits of incremental slices in narrowing the search space for faults during debugging; thus, incremental slicing could be usefully incorporated into a software-maintenance tool.

## 6 Related Work

Several researchers have considered the effects of pointers on program slicing and have presented results to perform slicing more effectively in the presence of pointers (e.g., [1, 3, 5, 15]). Some researchers have also evaluated the effects of the precision of the pointer analysis on subsequent analyses, such as the computation of def-use associations (e.g. [23]) and program slicing (e.g. [4, 14, 21]). However, none of that research distinguishes data dependences based on types of the definition, the use, and the paths between the definition and the use—it views uniformly each data dependence that arises in the presence of pointers.

Other researchers (e.g. [6, 9]) have investigated various ways to reduce the sizes of slices. However, they have not considered classifying data dependences and computing slices based on different types of data dependences as a means of reducing the sizes of slices.

Ostrand and Weyuker [18] extend the traditional data-flow testing techniques [8, 20] to programs that contain pointers and aliasing. To define testing criteria that adequately test the data-flow relationships in programs with pointers, they consider the effects of pointers and aliasing on definitions and uses. They classify definitions, uses, and def-clear paths depending on the occurrences of pointer dereferences in those entities. Based on these classifications, they identify four types of data dependences: strong, firm, weak, and very weak. The classification proposed by Ostrand and Weyuker, however, is coarser grained with respect to the one that we are using. The strong data dependence corresponds to data-dependence types 1 and 3 in our classification; the firm data dependence corresponds to types 2 and and 4; the weak data dependence corresponds to types 5 and 6; and finally, the very weak data dependence corresponds to the remaining 18 types of data dependences. Ostrand and Weyuker's classification groups together several types of dependences, and thus, may miss the differences caused by such dependences. Furthermore, Ostrand and Weyuker do not investigate how such classification affects the computation of program slices.

Merlo and Antoniol [16] present techniques to identify implications between nodes and data dependences. They distinguish definite and possible definitions and uses. They define relations to estimate the data dependences whose coverage is implied by the coverage of a node.

## 7 Summary and Future Work

In this paper, we presented a new incremental slicing technique, in which slices are computed by considering subsets of data dependences based on their types. Using this technique, we can increase the scope of a slice in steps, by incorporating additional types of data dependences at each step. To compute incremental slices, we extended the SDG representation and modified the SDG-based slicing algorithm to accommodate data-dependence types. We presented empirical results to illustrate the performance of the technique in practice. The results show that computing slices in increments can be useful for software-maintenance tools because each increment to the slice can be significantly smaller than the complete slice.

We also presented the results of a case study that shows how the new technique can be used for debugging purposes. We computed slices for a subject containing a known, subtle pointer-related fault and showed how incremental slicing can be used to narrow the search space for that fault. More generally, by decreasing the amount of information that is presented to software maintainers and by focusing on specific types of dependences, incremental slices can reduce the complexity of software-maintenance tasks, such as debugging.

Our future work includes extensions to our tool to use different, and more efficient, alias-analysis algorithms. This improvement will enable us to (1) perform experiments on subjects of bigger sizes, and (2) study the relation between the distribution of data dependences and the precision of the underlying alias analysis. We will also study the source code of the subjects to try to identify patterns in that code that can cause specific types of data dependences. We believe that such patterns could be of great help to tune analysis algorithms and provide guidelines for the programmers. Finally, we plan to further investigate the practicality and usefulness of our slicing paradigm for various applications, and to extend and generalize the incremental-slicing approach by considering additional parameters that can be used to compute slices in increments.

## References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, Analysis, and Verification*, pages 60–73, Oct. 1991.

[2] Aristotle Research Group. ARISTOTLE: Software engineering tools. http://www.cc.gatech.edu/aristotle/, 2000.

[3] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of ACM SIGSOFT 6th Intl. Symposium on the Found. of Software Engineering*, pages 46–55, Nov. 1998.

[4] L. Bent, D. C. Atkinson, and W. G. Griswold. A comparative study of two whole-program slicers for C. Technical Report UCSD TR CS2000-0643, University of California at San Diego, May 2000.

[5] D. W. Binkley and J. R. Lyle. Application of the pointer state subgraph to static program slicing. *The Journal of Systems and Software*, 40(1):17–27, Jan. 1998.

[6] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, Nov. 1998.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, Jul. 1987.

[8] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[9] M. Harman and S. Danicic. Amorphous program slicing. In *Proceedings of the 5th Intl. Workshop on Program Comprehension*, pages 70–79, Mar. 1997.

[10] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th Intl. Conference on Software Engineering*, pages 74–83, Apr. 1998.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[12] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 235-248, Jun. 1992.

[13] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN '93 Conference on Program Language Design and Implementation*, pages 56–67, Jun. 1993.

[14] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of ESEC/FSE '99*, vol. 1687 of *LNCS*, pages 199–215. Springer-Verlag, Sep. 1999.

[15] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the Intl. Conference on Software Maint.*, pages 421–432, Aug. 1999.

[16] E. Merlo and G. Antoniol. A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In *Proceedings of CASCON '99*, pages 173–186, Nov. 1999.

[17] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences. In *Proceedings of the 9th Intl. Workshop on Program Comprehension*, pages 39–49, May 2001.

[18] T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for lang. with pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 74–86, Oct. 1991.

[19] Programming Language Research Group. PROLANGS Analysis Framework. http://www.prolangs.rutgers.edu/, Rutgers University, 1998.

[20] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Apr. 1985.

[21] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *4th Intl. Static Analysis Symposium*, vol. 1302 of *LNCS*, pages 16–34, Sept. 1997.

[22] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21 st Intl. Conference on Software Engineering*, pages 432–441, May 1999.

[23] P. Tonella. Effects of different flow insensitive points-to analyses on DEF/USE sets. In *Proceedings of the 3rd European Conference on Software Maint. and Reengineering*, pages 62–69, Mar. 1999.

[24] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.