Effects of Pointers on Data Dependences

Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA 30332
{orso,sinha,harrold}@cc.gatech.edu

Abstract

This paper presents a technique for computing and classifying data dependences that takes into account the complexities introduced by specific language constructs, such as pointers, arrays, and structures. The classification is finer-grained than previously proposed classifications. Moreover, unlike previous work, the paper presents empirical results that illustrate the distribution of data dependences for a set of C subjects. The paper also presents a potential application for the proposed classification—program slicing—and a technique that computes slices based on data-dependence types. This technique facilitates the use of slicing for program understanding because a user can either augment a slice incrementally, by incorporating data dependences based on their relevance, or focus on specific kinds of dependences. Finally, the paper presents a case study that shows how the incremental addition of data dependences allows for growing the size of the slices in steps.

1 Introduction

Understanding data dependences in programs is a prerequisite to several program-comprehension-related activities, such as reverse engineering, impact analysis, and debugging. In particular, slicing techniques, which are often used for program understanding, depend on the availability of reliable information about dependences among program variables. Such dependences can be identified by computing definition-use (def-use) associations, which relate statements that assign values to variables to statements that use those values. The problem of computing def-use associations in the absence of pointers is relatively straightforward. In such cases, definitions and uses of variables can be identified by using only syntactic information. Once definitions and uses are known, def-use associations can be computed using a traditional data-flow analysis algorithm [2].

Unfortunately, traditional approaches for computing def-use associations are inadequate in the presence of programming language constructs such as pointers, arrays, and structures. The possibility of directly accessing memory locations, in languages such as C, complicates the identification of definitions and uses in the code. For example, a variable may be accessed at a given statement without syntactically appearing in it, if the access occurs through a pointer dereference. Therefore, syntactic information is not sufficient in the presence of pointers, and the set of memory locations that can be accessed through a dereference must be determined prior to the computation of def-use associations. Moreover, because an assignment or use through the dereference of a pointer can potentially assign a value to, or use the value of, one of several variables, these indirect assignments and uses must be treated differently from direct (i.e., syntactic) assignments.

In the first part of this paper, we extend previously presented classification schemes to allow for a more fine-grained taxonomy of def-use associations. In our scheme, a def-use association is classified into one of 24 categories. This classification is based on the kinds of the definition and the use—either definite or possible—in the def-use association, and on the types of paths occurring between the definition and the use. In this way, each def-use association corresponds to a specific kind of data dependence. We extend the traditional reaching-definition algorithm to compute and classify def-use associations according to our classification scheme. We also present and discuss empirical results, for a set of C subjects, about the distribution of def-use associations into various categories.

In the second part of the paper, we present some possible applications of the proposed classification. In particular, we evaluate the effects of classifying data dependences on program slicing: we introduce a slicing paradigm in which slices are computed by following only specified types of data dependences. Based on this paradigm, we present an incremental slicing technique. The technique can start the analysis of a program by computing slices that consider only "strong" (i.e., definite) data dependences, and then augment the slices incrementally by incorporating additional, "weaker," data dependences. This slicing approach lets the user first focus on a smaller, and thus easier to understand, subset of the program, and then consider increasingly bigger parts of the code. The technique also provides a way to isolate the data dependences that are caused by the presence of pointers. In this way, it is possible to highlight subtle data dependences that can affect the behavior of the program in possibly unforeseen ways, and provide useful information about those dependences. Finally, the technique offers a way of controlling the size of a slice by eliminating certain data dependences from the slices. We also present a case study that we performed to investigate the practical usability of the presented technique.

The main contributions of the paper are:

- A fine-grained classification of data dependences for languages, such as C, that let the programmer directly manipulate memory.
- Empirical results that illustrate the distribution of data dependences into various categories.
- A new slicing technique that allows for building slices by considering only a subset of data dependences, based on their relevance.
- A case study that shows the results of the application of the slicing technique to a real example.

The rest of the paper in organized as follows. The next section provides background information about data dependences, alias analysis techniques, and program slicing. Section 3 presents a classification scheme for data dependences and a technique for computing data dependences according to the classification. Section 4 illustrates the application of the data-dependence classification scheme for program slicing. Section 5 discusses related work. Finally, Section 6 presents conclusions and identifies potential future work.

2 Background

In this section, we define data dependences. We also briefly describe alias analyses and program slicing.

Data-flow analysis techniques require the control-flow relation of the program being analyzed. This relation can be represented as a control-flow graph. A *control-flow graph* (CFG) contains nodes, which represent statements, and edges, which represent potential flow of control among the statements. In addition, the CFG contains a unique entry node and a unique exit node. For each call site, the CFG contains a call node and a return node. For example, Figure 1 presents a sample program and the CFG for the program. Each node in the CFG represents a statement in the program[1] and each edge represents a transfer of control from the statement represented by the source of the edge to the statement represented by the target of the edge; nodes 1 and 8 are the entry and exit nodes, respectively.

```
1  begin M
2    read x
3    read y
4    if x > y then
5      read x
6      print x
     else
7      print y
     endif
8  end M
```

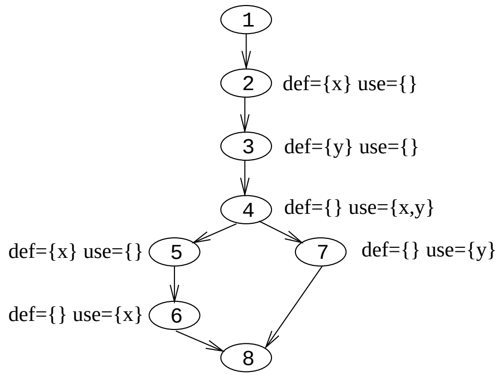

Figure 1. Sample program to illustrate definitions, uses, and data dependences (above); control-flow graph for the program annotated with def and use sets (below).

A statement *defines* a variable if the statement assigns a value to that variable. A statement *uses* a variables if the statement reads the value stored in that variable. For example, in the sample program, statement 2 defines variable `x`, statement 3 defines variable `y`, and statement 4 uses both `x` and `y`. To perform data-flow analyses, we associate two sets of variables with each node in the CFG: the *definition set*, $def(n)$, for a node n contains those variables that are defined at node n; the *use set*, $use(n)$ contains those variables that are used at node n. For example, in the sample program, def(3) = {`y`} and use(4) = {`x`, `y`}.

A *path* in a CFG is a sequence of nodes $(n_1, n_2, \ldots, n_k)$, $k \geq 0$, such that, if $k \geq 2$, for $i = 1, 2, \ldots, k-1$, (n_i, n_{i+1}) is an edge in the CFG. A *definition-clear path* (def-clear path) with respect to a variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that no node in $n_1, n_2, \ldots, n_k$ defines v. For example, in the sample program, (2, 3, 4) is a def-clear path with respect to variable `x`, whereas, because of the definition of `x` at node 5, path (2, 3, 4, 5, 6) is

[1]A CFG can also be built at the basic-block level; in such a CFG, each node represents a sequence of single-entry, single-exit statements.

```
      int i;
      main() {
          int *p;
          int j, sum1, sum2;
 1.       sum1 = 0;
 2.       sum2 = 0;
 3.       read i, j;
 4.       while ( i < 10 ) {
 5.           if ( j < 0 ) {
 6.               p = &sum1;
              }
              else {
 7.               p = &sum2;
              }
 8.           *p = add( j, *p );
 9.           read j;
          }
10.       sum1 = add( j, sum1 );
11.       print sum1, sum2;
      }
```

```
      int add( int val, int sum ) {
          int *q, k;
12.       read k;
13.       if ( sum > 100 ) {
14.           i = 9;
          }
15.       sum = sum + i;
16.       if ( i < k ) {
17.           q = &val;
          }
          else {
18.           q = &k;
          }
19.       sum = sum + *q;
20.       i = i + 1;
21.       return sum;
      }
```

Figure 2. Program Sum**.**

not. A definition d_2 *kills* a definition d_1 if both d_1 and d_2 refer to the same variable v, and there exists a def-clear path between d_1 and d_2. For example, the definition of x at node 5 kills the definition of x at node 2.

A *reaching-definition set*, $rd(j)$, defined with respect to a node j, is the set of variable–node pairs $< v, i >$ such that $v \in$ def(i) and there exists a def-clear path with respect to v from i to j. A *data dependence* is a triple (d, u, v) such that $v \in$ use(u) and $< v, d > \in$ rd(u). A data dependence is also referred to as a *definition-use association* (def-use association, or DUA). The computation of data dependences can be performed by first computing reaching definitions, and then examining, for each use, the reaching definitions for that use.

An *alias* is a name referring to the same memory location as another name, at a given program point. In such a case, that memory location can be accessed through any of these two names. An alias relation at program point n is a *may alias* relation if the relation holds on some, but not all, program paths leading up to n. An alias relation at point n is a *must alias* relation if the relation holds on all paths up to n. As an example, consider program Sum (Figure 2). On line 8, *p is a may alias for sum1 and sum2, because it can refer to either sum1 or sum2, depending on the path followed to get to statement 8 (i.e., depending on whether statement 6 or statement 7 is executed). Therefore, the alias set for *p at statement 8 contains two elements: sum1 and sum2. A variety of alias-analysis algorithms have been presented in the literature, which vary in the efficiency and the precision with which they compute the alias relations (e.g., [3, 15, 25, 16]). For the empirical studies reported in this paper, we used the may-alias information computed by the flow-sensitive, context-sensitive alias algorithm described by Landi and Ryder [15].

Program slicing is a technique for identifying transitive control and data dependences in a program. A *backward slice* for a program P, computed with respect to a *slicing criterion* $< s, V >$, where s is a program point and V is a set of program variables referenced at s, includes statements in P that may influence the values of the variables in V at s [28].[2] There are two alternative approaches to computing slices that either propagate solutions of data-flow equations using a control-flow representation [12, 28] or perform graph reachability on dependence graphs [14, 24]. For this work, we extended the dependence-graph-based approach to computing slices [14, 24].

3 Data Dependences in the Presence of Pointers

The presence of pointers causes complex data-dependence relationships. Because of pointers and aliasing, it may not be possible to identify unambiguously the variable that is actually defined (resp., used) at a statement containing a definition (resp., use). To account for such effects, we classify data dependences based on two factors: (1) types of definitions and uses, (2) types of the paths from the definitions to the uses. In the rest of this section, we present a new classification scheme that extends the classification scheme presented by Ostrand and Weyuker [20].

[2]A slice can also be computed in the forward direction: a forward slice includes those statements in P that may be influenced by the values of the variables in V at s.

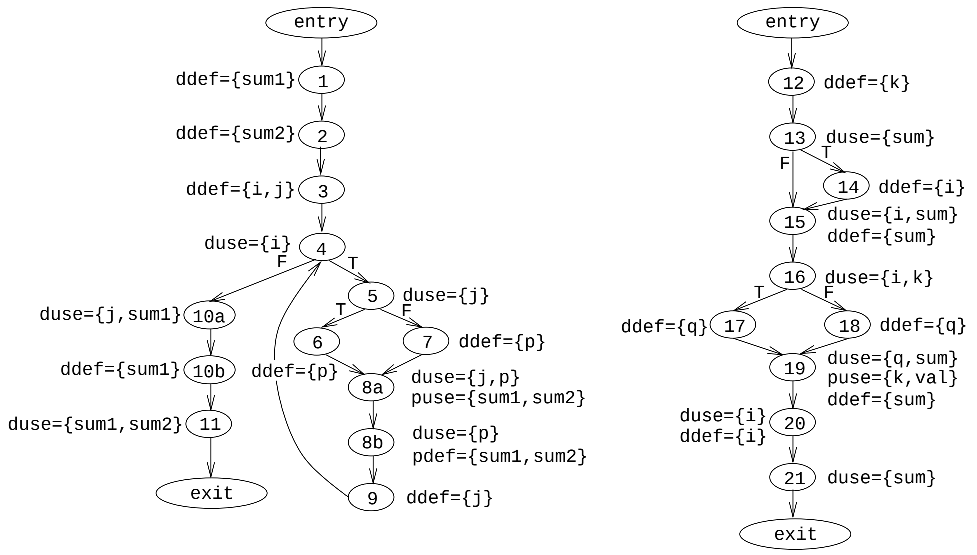

Figure 3. Control-flow graphs for program `Sum` (Figure 2) with definite and possible definition and use sets at each node.

3.1 Classification of definitions and uses

In the presence of pointers, the traditional data-flow definitions, presented in Section 2, prove to be inadequate. In programs with pointers, accesses to a variable that involve a dereference of a pointer can potentially access one of several variables. The traditional notion of definitions and uses fails to distinguish such accesses from those in which only a single variable can be referenced. For example, in program `Sum`, we can identify unambiguously the variables that are defined at statement 3—those variables are always `i` and `j`. However, for the definition in statement 8, the variable that is actually defined is the variable to which `p` points at that statement. Depending on the execution path to that statement, `p` can point to different variables: if the predicate in statement 5 is true, `p` points to `sum1` at statement 8, whereas if the predicate in statement 5 is false, `p` points to `sum2` at that statement. Thus, statement 8 can potentially define either `sum1` or `sum2`. Unlike the definitions of `i` and `j` at statement 3, however, neither of these two definitions is definite. To distinguish these types of definitions, we classify the definitions in statement 3 as definite definitions, and those in statement 9 as possible definitions.

A definition (resp., use) of variable v through an expression e at statement s is a *definite definition* (resp., *definite use*) of v if and only if static analysis determines that, at statement s, the only variable that can be accessed through e is v. A definition (resp., use) of variable v through an expression e at statement s is a *possible definition* (resp., *possible use*) if and only if static analysis determines that, at statement s, a set of variables $\mathcal{V}$ can be accessed through e, where $v \in \mathcal{V}$ and $\mathcal{V}$ contains at least two elements. Note that an access to a variable through a pointer dereference, such that the pointer can point to a single variable, is still considered a possible definition (use) of that variable. This occurs because of the limitations of static analysis in approximating the dereferenced memory locations in certain cases, such as when the dereferenced location is an element of an array or a heap element. For example, consider a statement n, `p = &a[i]`, where the index variable `i` is a loop-control variable or is passed in as a parameter to the procedure that contains the statement. In this case, static analysis cannot determine all memory locations to which `*p` can be aliased to at n; typically, static analysis would approximate the aliases of `*p` with a single element alias set—$\{$`a[]`$\}$. Although the alias set for `*p` contains a single element, a definition or use that occurs through `*p` is not a definite definition or definite use. A similar problem occurs if `*p` is aliased to a heap location. Therefore, to preserve the safety of the reaching-definition analysis in the presence of such alias relations, definition and uses that occur through

Table 1. Def-use types based on the types of definitions and uses.

	Definite definition	Possible definition
Definite use	def-use type 1	def-use type 3
Possible use	def-use type 2	def-use type 4

Table 2. Classification of Π—paths from definitions to uses—after incorporating the occurrences of definite killing paths. The last base type for Π is listed only to illustrate the completeness of the base classification of Π; for such base types of Π, the definitions and uses do not constitute def-use associations.

Base type of Π	Occurrence of definite killing path in Π	Extended type of Π
no possible def-clear	$\forall\pi \in \Pi$: π is not definite killing	DRD-K
	$\exists\pi \in \Pi$: π is definite killing	DRD+K
some possible def-clear	$\forall\pi \in \Pi$: π is not definite killing	DPRD-K
	$\exists\pi \in \Pi$: π is definite killing	DPRD+K
all possible def-clear	$\forall\pi \in \Pi$: π is not definite killing	PRD-K
	$\exists\pi \in \Pi$: π is definite killing	PRD+K
all definite kill	$\forall\pi \in \Pi$: π is definite killing	N/A

single-element alias sets are classified as possible.

Because each definition is either definite or possible, we associate with each node in the CFG two sets of definitions: the *definite-definition set*, $ddef(n)$, for a node n contains those variables that are definitely defined at node n; the *possible-definition set*, $pdef(n)$, for a node n contains those variables that are possibly defined at node n. Analogously, we associate with each node in the CFG two sets of uses: the *definite-use set*, $duse(n)$, for a node n contains those variables that are definitely used at node n; the *possible-use set*, $puse(n)$, for a node n contains those variables that are possibly used at node n. For example, Figure 3 shows the CFGs for the procedures in `Sum`, and lists the ddef, pdef, duse, and puse sets for each node; for clarity, we show only the non-empty sets at each node in the figure.

Based on the types of definitions and uses, it is thus possible to have four types of def-use associations, shown in Table 1.

3.2 Classification of def-clear paths

The second dimension for the classification of a def-use association considers the types of paths from the definition to the use. Let (d, u, v) be a def-use association. In the absence of the effects of pointer variables, it is sufficient to classify each path π from d to u into one of two types, based on whether the definition at d is killed along path π. However, the presence of possible definitions introduces an additional category in which π can be classified: a definition may be possibly killed along π. Thus, in the presence of pointers, we classify π into one of three types:

A *definite def-clear path* with respect to variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that no node in $n_1, n_2, \ldots, n_k$ contains either a definite or a possible definition of v. For example, in program `Sum`, path (1, 2, 3, 4, 10a) is a definite def-clear path with respect to variable `sum1`.

A *possible def-clear path* with respect to variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that there exists at least one n_i, $1 \leq i \leq k$, that contains a possible definition of v, but no node in $n_1, n_2, \ldots, n_k$ contains a definite definition of v. For example, in program `Sum`, the path (8a, 8b, 9, 4, 10a) is a possible def-clear path with respect to variable `sum1`, because node 8b contains a possible definition of `sum1` and no other node in the path contains a definite definition of `sum1`.

A *definite killing path* with respect to variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that there exists at least one n_i, $1 \leq i \leq k$, that contains a definite definition of v. For example, in program `Sum`, the path (10a, 10b, 11) is a definite killing path with respect to variable `sum1`, because node 10b contains a definite definition of `sum1`.

Based on the above definitions, we classify the set of paths from the definition to the use for a def-use association. Let (d, u, v) be a def-use association and let Π be the set of paths from d to u; we classify Π into one of three types:

1. Π is *no possible def-clear* if and only if there exists at least one path in Π that is a definite def-clear path with respect to v, and no path in Π is a possible def-clear path with respect to v.
2. Π is *some possible def-clear* if and only if there exists at least one path in Π that is a definite def-clear path with respect to v, and there exists at least one path in Π that is a possible def-clear path with respect to v.
3. Π is *all possible def-clear* if and only if there exists at least one path in Π that is a possible def-clear path with respect to v, and no path in Π is a definite def-clear path with respect to v.

For example, in program `Sum`, for def-use association (1, 8a, `sum1`), Π is no possible def-clear; for def-use association (8b, 11, `sum2`), Π is all possible-def clear; and, for def-use association (2, 11, `sum2`), Π is some possible def-clear.

To investigate further the occurrences of various types of def-use associations and the significance of those occurrences, we extend the above classification by considering

Table 3. Classification of def-use associations: 24 types that result from a cross product of def-use types (Table 1) and the second alternative for path classification (column 3 of Table 2).

	def-use type 1	def-use type 2	def-use type 3	def-use-type 4
DRD-K	DUA type 1	DUA type 7	DUA type 13	DUA type 19
DPRD-K	DUA type 2	DUA type 8	DUA type 14	DUA type 20
DRD+K	DUA type 3	DUA type 9	DUA type 15	DUA type 21
DPRD+K	DUA type 4	DUA type 10	DUA type 16	DUA type 22
PRD-K	DUA type 5	DUA type 11	DUA type 17	DUA type 23
PRD+K	DUA type 6	DUA type 12	DUA type 18	DUA type 24

Table 4. Def-use associations, with their types, that occur in program *Sum*.

Def-use association	Type	Def-use association	Type	Def-use association	Type
(1, 8a, `sum1`)	type 8	(1, 10a, `sum1`)	type 2	(2, 8a, `sum2`)	type 8
(2, 11, `sum2`)	type 2	(3, 4, `i`)	type 1	(3, 5, `j`)	type 3
(3, 8a, `j`)	type 3	(3, 10a, `j`)	type 3	(6, 8a, `p`)	type 3
(7, 8a, `p`)	type 3	(8a, 8a, `sum1`)	type 20	(8a, 10a, `sum1`)	type 14
(8a, 8a, `sum2`)	type 20	(8a, 11, `sum2`)	type 14	(9, 5, `j`)	type 3
(9, 8a, `j`)	type 3	(9, 10a, `j`)	type 3	(10a, 11, `sum1`)	type 1
(12, 16, `k`)	type 1	(12, 19, `k`)	type 7	(14, 15, `i`)	type 1
(14, 16, `i`)	type 1	(14, 20, `i`)	type 1	(15, 19, `sum`)	type 1
(17, 19, `q`)	type 1	(18, 19, `q`)	type 1	(19, 21, `sum`)	type 1

the occurrences of definite killing paths in Π. As a result, we obtain six types of paths from definitions to uses, which are summarized in Table 2. For completeness, in Table 2 we also mention a fourth type of Π, *all definite kill*. We do not consider this type in our classification because it refers to the case in which all the paths between the definition and the use are definitely not def-clear, and therefore the definition and the use are not part of a def-use association.

3.3 Classification of def-use associations

We classify def-use associations based on the types of the definition and the use (Table 1) and the types of the path between the definition and the use (Table 2). This cross product results in 24 types of def-use associations, shown in Table 3. Table 4 lists all the def-use associations that occur in program `Sum`, together with their types.

Our classification scheme extends the one proposed by Ostrand and Weyuker [20]. Ostrand and Weyuker's approach is coarser with respect to two different aspects. First, Ostrand and Weyuker do not distinguish sets of paths based on the presence of definite killing path (i.e., they classify paths according to the classification in column 1 of Table 2). Second, although such a classification allows for identifying 12 types of def-use associations, Ostrand and Weyuker consider separately only three of these types: *strong* def-use association (def-use type 1, no possible def-clear paths), *firm* def-use association (def-use type 1, some possible def-clear paths), and *weak* def-use association (def-use type 1, all possible def-clear paths). They group the remaining nine types—in which either the definition or the use is not definite—into one type, which they call *very weak* def-use association.

3.4 Computation of def-use associations

To compute the different types of def-use associations identified in the previous section, we modify both steps of the traditional algorithm for computing def-use associations: (1) the computation of reaching definitions using data-flow equations, and (2) the computation of the def-use associations using the reaching definitions.

The traditional algorithm for computing reaching definitions propagates iteratively a set of data-flow facts—the definite reaching definitions—until the value of the set at each statement reaches a fixed point. To compute the value of the set at each statement, the algorithm uses a pair of equations. The first equation describes the value of the set at the beginning of a statement, based on the value of the set at the end of each control-flow predecessor of the statement. The second equation describes the value of the set at the end of a statement, based on the transformation of the set by the statement. To facilitate the identification of each of the 24 types of def-use associations, we extend the

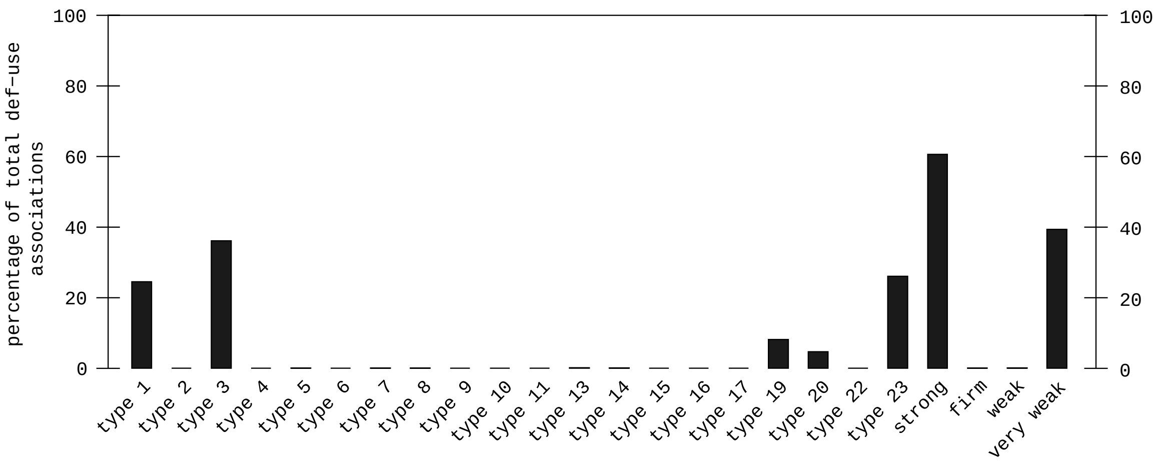

Figure 4. Distribution of data-dependence types using our classification and Ostrand and Weyuker's classification; types 12, 18, 21, and 24 did not occur in the subjects.

reaching-definition algorithm to propagate two additional sets of data-flow facts at each statement. The first set contains the possible definitions that reach a statement; the second set contains the killed definitions that reach a statement. The extended algorithm computes the three sets iteratively until the sets converge. To compute each of the additional sets, the algorithm uses a pair of data-flow equations that describe, respectively, the values of the sets at the beginning and the end of a statement; Reference [19] provides details of the algorithm and the data-flow equations.

In the second step of computing def-use associations, the algorithm performs set operations on the three sets computed during the first step to identify definitions that reach each statement along each of the six types of paths listed in column 3 of Table 2. Finally, using the types of the reaching definitions, the types of the uses, and the types of the paths along which the definitions reach the uses, the algorithm computes and classifies the def-use associations [19].

The time and space complexity of the extended algorithm is similar to that of the traditional algorithm. The extended algorithm computes two additional sets of data-flow facts; however, these sets can be represented and manipulated efficiently using bit vectors. The iterative propagation can be implemented efficiently using a depth-first ordering of the nodes in the CFG [2]. The extended algorithm computes the same number of def-use associations as the traditional algorithm.

3.5 Distribution of data-dependence types

To investigate the distribution of data dependences into various types, we implemented a prototype and performed empirical studies with a set of C subjects. We implemented the reaching-definition algorithm using the ARISTOTLE analysis system [13]. To account for the effects of aliases, we replaced the ARISTOTLE front-end with the PROLANGS Analysis Framework (PAF) [10]. 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 5 for the empirical studies.

Table 5. Programs used for the empirical studies reported in the paper.

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	Parser for antenna array description language	6201
tot_info	Statistical information combiner	477
unzip	Compress/extract utility	2906

Figure 4 illustrates the distribution of data-dependence types for the subjects. Each bar in the figure corresponds to a data-dependence type and represents the percentage of data-dependences for that type. The data in the figure illustrate that data dependences fall predominantly into only a few types. DUA type 1, DUA type 3, and DUA type 20 occur most frequently: together these three types account for over 86% of the total data dependences. These types along with DUA types 19 and 20 together constitute 99.4% of data dependences. Of the remaining 19 types of data dependences, 15 types occur in marginal numbers and account for the remaining 0.6% of the data dependences. The remaining 4 types of data dependences—types 12, 18, 21, and 24—do not occur in the subjects; these types are not listed along the horizontal axis in Figure 4.

The results of this study are preliminary in nature. Although the data in Figure 4 shows trends in the distribution of data-dependence types, the scarcity of the data points, prevents us from drawing any conclusions about the distribution. Further experimentation with more and diverse subjects will help determine if trends, such as the frequent occurrence of DUA type 23, persist. The data in the figure shows that for over 24% of the data dependences, no path from the definition to the use contains a redefinition of the relevant variable. This result is important for structural testing because it means that a test case that covers the definition and use statements also covers the corresponding def-use association.

The last four bars in Figure 4 show the distribution of data-dependence types according to Ostrand and Weyuker's classification [20]. According to their classification, over 60% of the data dependences are strong, and over 39% of the data dependences are very weak. Firm and weak data dependences constitute a little over 1% of data dependences.

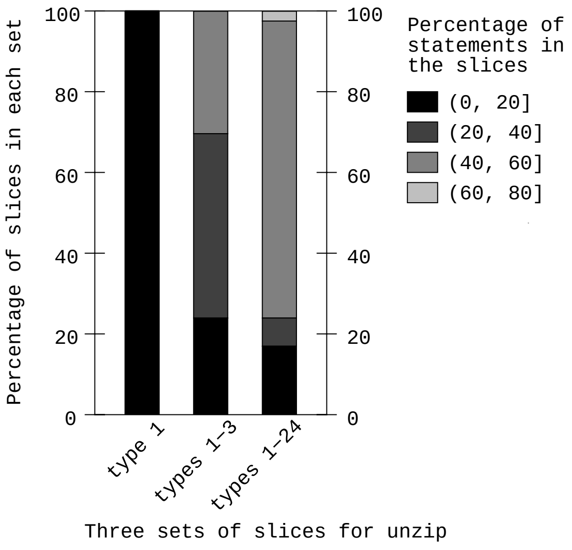

Figure 5. Percentage of slices for `unzip`—in each of slice sets for data-dependence types 1, 1–3, and 1–24—that included various percentages of program statements.

4 Applications of the Data-Dependence Classification

The ability to classify data dependences can be exploited for different applications. For example, data dependences that are ordered based on their "strength" can guide a data-flow testing strategy [9], can be used to perform impact analyses focused on different kinds of dependences, and can be analyzed to identify parts of the code where possibly unforeseen data dependences require careful software inspections. In short, all activities that depend on data-dependence information can utilize such a classification. In this paper, we focus on an application that is related to program understanding—program slicing. In the following section, we define a slicing technique that lets us compute slices based on data-dependence types; we also illustrate a case study in which we apply the technique.

4.1 Program Slicing

Traditional slicing techniques (e.g., [12, 14, 28]) include in the slice all statements that affect the slicing criterion through direct or transitive control and data dependences. Such techniques compute the slice by computing the transitive closure of all control 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 identifies one or more types of data dependences. The slice includes those statements that may affect the value of the variables in V at s through transitive control or specified types of data dependences.

Using the new slicing paradigm, we define a slicing technique that increases the scope of a slice incrementally by including data dependences of different types. The technique starts by considering the stronger types of data dependences and computes the slice based on those data dependences. Then, it increments the slice by considering additional types of (weaker) data dependences and adding to the slice statements that affect the slicing criterion through the weaker data dependences. This technique offers several advantages. First, it focuses the attention on specific types of data dependences and enables us to consider in stages the effects of various types of data dependences on slices. Second, it enables us to control the size of a slice by eliminating certain dependences initially and incorporating them later if required.

The new slicing paradigm and the incremental slicing technique apply to both the data-flow-based [12, 28] and the dependence-graph-based [14, 24] approaches for computing slices. To perform the case study, we extended the dependence-graph-based approach to represent data-dependence types.

4.2 Case Study for Slicing

To investigate the performance of the new slicing technique, we used the prototype for classifying data dependences and extended the dependence-graph-based slicer in the ARISTOTLE analysis system. We selected one subject, `unzip`, for the study. Based on the distribution of data-dependence types in `unzip`, we computed three sets of slices: the first set were based only on DUA type 1, the second set included DUA types 1 through 3, and the last set included all DUA types.

Figure 5 presents data to illustrate the growth in sizes of the slices in each set. It shows the distribution of the slices for unzip according to their sizes. The vertical axis represents the percentage of the slices for `unzip` (we computed 1861 slices for `unzip`); each segmented bar represents slices from one set; and the segments in the bar represent various ranges of the slice sizes. The figure illustrates how the distribution changed from one set to the next. In the first set of slices, which were computed by considering only DUA type 1, each slice contains fewer than 20% of the program statements. However, when we also consider DUA types 2 and 3, 45% of the slices contain between 20% and 40% of the program statements. In the third set of slices, which were computed by considering all types of data dependences, over 73% of the slices contain between 40% and 60% of the program statements.

Although the differences in the slice sizes between the first two sets are not related to the presence of pointers[3], the differences themselves are significant. Thus, the incremental slicing approach appears promising in reducing the sizes of slices. The differences in the sizes of the slices between the second and third sets are related to the effects of pointers. We examined manually the differences in some of the slices and found that the slices in the third set included statements that were related by subtle, hard-to-detect pointer-induced data dependences. The technique, thus, appears to be useful in isolating, and focusing attention on, such dependences. In future work, we intend to conduct more extensive empirical studies to evaluate the effectiveness of the slicing technique.

5 Related Work

Ostrand and Weyuker [20] extend the traditional data-flow testing techniques [9, 22] 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 def-use associations: strong, firm, weak, and very weak. The strong def-use association corresponds to DUA types 1 and 3 in our classification; the firm def-use association corresponds to DUA types 2 and and 4; the weak def-use association corresponds to DUA types 5 and 6; and finally, the very weak def-use association correspond to the remaining 18 types of def-use associations in our scheme. Our classification is finer grained. Ostrand and Weyuker's classification groups several types of dependences together, and thus, may miss the differences caused by such dependences.

Pande, Landi, and Ryder [21] describe an algorithm for computing interprocedural reaching definitions in the presence of pointers. They define *a conditional reaching definition* as a reaching definition that holds under the assumed conditions for aliasing.

Merlo and Antoniol [18] present techniques to identify implications among data dependences in the presence of pointers. They also distinguish definite and possible definitions and uses and, based on these, identify definite and possible data dependences. The definite data dependence corresponds to data-dependence types 1 and 3 in our classification, whereas the possible data dependence corresponds to types 2, 4–6, 8, 10–12, 14, 16–18, 20, and 22–24; the remaining types in our classification fall in neither the definite nor the possible data-dependence category in Merlo and Antoniol's classification.

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, 4, 6, 7, 17]). 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., [26]) and program slicing (e.g., [5, 16, 23]). However, none of that work considers definitions, uses, and def-use associations in terms of the certainty with which those entities occur. Tonella and colleagues [27] analyze the effects of the precision of the reaching-definition computation on def-use associations.

Other researchers (e.g., [8, 11]) 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.

6 Summary and Future Work

In this paper, we presented a technique for computing and classifying data dependences in programs that use pointers. Our classification is finer grained with respect to

[3]This occurs because `unzip` contained no DUA type 2. Therefore, the second set of slices were based on DUA types 1 and 3, neither of which involve pointers.

previously presented classifications, and allows for partitioning of data dependences into 24 types, based on their "strength." We also presented the first set of experimental results that illustrates the distribution of data dependences for a set of C subjects. Although we can draw no conclusive inference, the data gathered so far show trends that are worth further investigation.

We illustrated a potential application of the proposed classification for program slicing. Our slicing technique lets the user first focus on a smaller, thus easier to understand, subset of the program, and then consider increasingly bigger parts of the code. We have also presented a case study that shows how the addition of "weak" data dependences allows for incrementally growing the size of the slices.

In future work, we will conduct further empirical studies to evaluate both the distribution of the data dependences and the effectiveness of the incremental slicing technique. Our future work also includes the extensions of our prototype to use a different, more efficient, alias analysis algorithm. This improvement will allow us (1) to perform experiments on subjects of bigger size, and (2) to study the relation between the distribution of data dependences and the precision of the underlying alias analysis. We also plan to perform a study of the source code of the subjects trying 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 program-analysis algorithms and provide guidelines for programmers.

Acknowledgments

This work was supported in part by a grant from Boeing Commercial Airplanes to Georgia Tech, by National Science Foundation awards CCR-9707792, CCR-9988294, and CCR-0096321 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. The work was also supported by the ESPRIT Project TWO (EP n.28940), and by the Italian Ministero dell' Universita e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO Project.

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] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[3] L. O. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.

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

[5] 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.

[6] D. W. Binkley. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum*, pages 261–268, Nov. 1993.

[7] 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.

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

[9] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, Oct. 1988.

[10] P. L. R. Group. PROLANGS Analysis Framework. http://www.prolangs.rutgers.edu/, Rutgers University, 1998.

[11] M. Harman and S. Danicic. Amorphous program slicing. In *Proceedings of the Fifth International Workshop on Program Comprehension*. IEEE Computer Society Press, 1997.

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

[13] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Mar. 1997.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, Jan. 1990.

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

[16] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of ESEC/FSE '99 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, Sept. 1999.

[17] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the International Conference on Software Maintenance*, pages 421–432, August–September 1999.

[18] E. Merlo and G. Altoniol. Pointer sensitive def-use pre-dominance, post-dominance and synchronous dominance relations for unconstrained def-use intraprocedural computation. Technical Report EPM/RT-00/01, Ecole Polytechnique of Montreal, Mar. 2000.

[19] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences and program slicing. Technical Report GIT-CC-00-33, Georgia Institute of Technology, November 2000.

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

[21] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. Softw. Eng.*, 20(5):385–403, May 1994.

[22] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, SE-11(4):367–375, Apr. 1985.

[23] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 16–34, Sept. 1997.

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

[25] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

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

[27] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis. *Journal of Software Maintenance: Research and Practice*, 11(2):117–142, March–April 1999.

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