Masaryk University
Faculty of Informatics

Slicing of LLVM bitcode

Master's Thesis

Marek Chalupa

Brno, Spring 2016

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marek Chalupa

Advisor: doc. RNDr. Jan Strejček, Ph.D.

Acknowledgement

I would like to thank to my supervisor for his time and always clear and critical judgement while discussing the thesis. Many thanks go also to my familly and friends that supported me during my whole studies.

Abstract

Symbiotic is an open-source verification tool that use program slicing to speed-up the verification. This thesis describes an implementation of a new slicer based on dependence graphs that replaces the old one which use a data-flow approach. First, we introduce the reader to the basic theory of program slicing and describe slicing algorithms, and then we describe the algorithms that are needed for building a dependence graph and their implementation. These are namely: an algorithm for the computation of control dependencies, a pointer analysis and a reaching defintion analysis. At the end of the thesis, we compare the old and the new slicers on non-trivial set of benchmarks.

Keywords

program slicing, slicing, backward static slicing, program slice, executable slice, dependence graph, program dependence graph, system dependence graph, static analysis, program analysis, data-flow analysis, pointer analysis, reaching definitions analysis, data dependence, control dependence, LLVM

Contents

1 Introduction

Bugs in computer programs have a great impact not only on the software usability, but also on the reliability and, in some corner cases, even on human safety. Finding them manually is hard and lengthy process. There is a number of techniques that can find (possible) faults in a software automatically. We can coarsely divide these techniques into two categories: a dynamic analysis and a static analysis. The dynamic analysis execute a program and look for errorneous behaviour during the execution. In contrary, the static analysis do not execute the program at all and tries to draw conclusions about a program only from the source code or some other program representation. A problem with a dynamic analysis (where belongs, for example, a testing) is that we are not able to execute a program in all possible settings that can occur during a runtime. Only to execute a program with all possible input values for a single 64-bit variable is usually infeasible. As a result, we can miss faults in a software because the program did not go along the error path during the execution.

The static analysis, on the other hand, can be capable of the exploration of the whole state space of a program in most cases, but it is computationaly very demanding. Since the static analysis does not execute the code, it does not have whole information about the program's configuration and its inputs (it has only what is in the source code). Therefore it must make some conservative assumptions (it *over-approximates* the analyzed information) about the program. As a result, static analysis tools may report a bug when there is none. If a static analysis does not make conservative assumptions, but rather *under-approximates* the analyzed information, it may incorrectly report that a program has no bug even when a bug is present.

Symbiotic [12, 43] is an open-source static analysis tool for verification of sequential programs that uses three well-known techniques: instrumentation, slicing and symbolic execution. An instrumentation is a routine that inserts auxiliary parts of code into the analyzed code (for example, checks for an out-of-bound access to arrays or proper locking of mutxes). The goal of the instrumentation in Symbiotic is to explicitely mark possible error sites in a program. The instrumented code is then sliced with respect to these error sites. Program slicing [48, 49] is a techinque that removes statements of a program that may not have any effect on the reachability of the highlighted error locations, thus effectively reducing the portion of the code that must be analyzed in the last step which is a symbolic execution. The symbolic execution executes the program, but instead of concrete values of variables use symbolic values [25]. Using symbolic values allows the symbolic execution to explore every path in the program (if there is finitely many of them) and verify whether some error site is reachable.

Symbiotic is very precise: it may produce an incorrect answer in only one situation: when slicing removes a non-terminating loop or call to `exit`, which makes a previously unreachable error reachable. That can happen because slicing assumes that every loop eventually terminates. A remedy of this behavior is possible, but currently not implemented (we describe it in Chapter 2).

The aim of this thesis is to replace the old slicer in Symbiotic that use a data-flow approach to program slicing by a new one that is based on dependence graphs. The implementation is supposed to use LLVM compiler infrastructure, as the rest of Symbiotic.

The outline of this thesis is as follows: in Chapter 2, we give the reader an insight into the theory of program slicing. We describe algorithms that have been invented to slice programs, namely a Weiser's algorithm based on a data-flow approach, and a class of algorithms that utilize dependence graphs. At the end of the chapter, we describe extensions to the dependence-graph based slicing algorithm to correctly handle pointers and other features of modern programming languages.

In Chapter 3, we describe the algorithms that we use in the implementation of the new slicer as well as details of their implementation (e.g. used data structures).

In Chaper 4, we evaluate the impact of the new slicer on the performance of Symbiotic and we compare the old and the new slicer on a nontrivial set of benchmarks.

Chapter 5 is a conclusion of work.

In Chapter 6, we propose a new notion of a control dependence and an algorithm for its computation. The new control dependence can seamlessly handle infinite loops as well as all other problematic parts of programs while slicing.

2 Program Slicing

The term *program slicing* is used for several techniques that can decompose a program based on data-flow information rather than on its structure. The exact definition may vary depending on the purpose of slicing. Most generally, we would say that program slicing is a technique that extracts statements of a program that are relevant to the program's behavior with respect to some (given) criterion. The result of such procedure is called a *slice*. This technique was informally introduced by Mark Weiser in 1981 in the paper of the same name [48] and again in a journal in 1984 [49] with more rigorous mathematical basement.

Weiser's motivation was to algorithmically express what a programmer does when debugging computer programs [50]. He described that notion formally and proved that computing minimal-size slices is in general undecidable [49], although a good approximation can be obtained using a data-flow analysis. Since then, slicing has shown to be useful in many other areas than debugging, for example testing, refactorization, integration, program verification, reverse engineering and more [11, 46, 51].

There have been developed many types of slicing since 1981 [46, 51]. In this thesis, we focus on the *backward static slicing*. Backward because the slice is created by searching the program backwards from the points of interest and static because the slice is created independently of a particular run of a program – it preserves a program's behavior (with respect to a given criterion) on any path that the program can take.

In this chapter, we give a few preliminary definitions and a definition of an executable slice (Section 2) and describe algorithms that have been invented to compute (executable) backward static slices (Section 2.1 and 2.2). Then we extend the algorithms to address more programming features, notably pointers and unstructured and interprocedural control flow (Section 2.3).

Executable program slice

To be able to define a slice, we need a few preliminary definitions. Understanding the flow of control in a program is necessary to draw conclusions about the program. It may not be easy, though (for example in the presence of *goto* statements). Representing a program as a graph helps fighting this problem:

Definition 1. *A* control flow graph (CFG) *of a program P is a quintuple (N, E, n_s, n_e, l) where (N, E) is a finite directed graph, N is a set of nodes and $E \subseteq N \times N$ is a set of edges. Each statement of P is represented by a node in the CFG*[1] *and edges between nodes represent the flow of control in P: there is an edge between nodes n_1 and n_2 iff n_2 can be executed immediately after n_1. There are distinguished entry and exit nodes in N, n_s and n_e, such that every node $n \in N$ is reachable from n_s, and n_e is reachable from n. Moreover, n_e has no outcoming edges. l is a partial labeling function $l : E \rightarrow \{T, F\}$ that assigns labels to edges in agreement with the flow of control in P.*

Let us establish a convention: we do not differentiate between statements of a program P and nodes of its CFG, since the CFG represents the program P (there is one-to-one correspondence). If not stated otherwise, we assume that programs use only *if-then-else* constructs, no *switch*[2] or alike. As a result, every node from a CFG has the output degree at most two. An example of a CFG can be found in Figure 2.1.

Definition 2. *Let (N, E, n_s, n_e, l) be a CFG of a program P. A run of the program P is a sequence of nodes from the CFG*

$$n_1, n_2, n_3, \ldots, n_k$$

where $n_1 = n_s$, $n_k = n_e$ and for all i, $1 \leq i < k$, $(n_i, n_{i+1}) \in E$.

If a run corresponds to some real execution of a program, we say that it is a *feasible* run, otherwise it is an *unfeasible* run. As can be seen

1. Some versions of control flow graphs do not include unconditional jump statements (*goto*, *break*, *continue*) as a node, but rather represent such statements as an edge. We include all statements of a program P as nodes to get bijective mapping between P and nodes of its CFG.
2. Every *switch* statement can be transformed into a sequence of *if-then-else* statements.

```
int n = 0;
int odds = 0;
int sum = 0;
int x = read();
while(x > 0) {
  sum += x;
  ++n;
  if (x % 2)
    ++odds;
  x = read();
}
print(sum);
print(n);
print(odds);
```

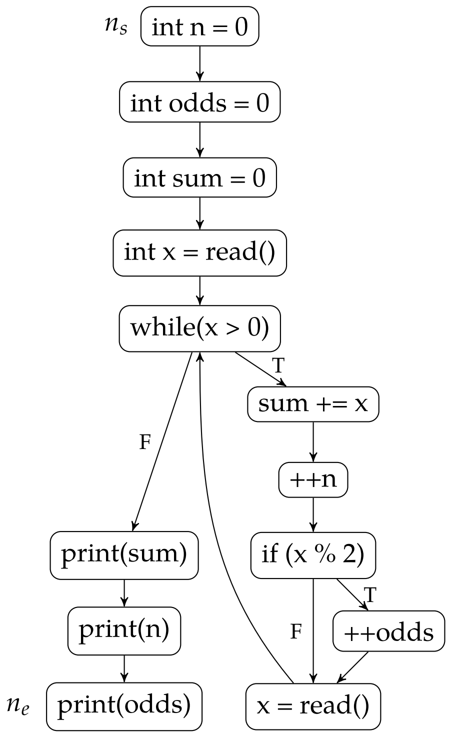


Figure 2.1: A simple program and its CFG.

from the definition, we consider only finite runs of programs. We give the reason why later in this section. With every feasible finite run of a program is associated a state trajectory, which is, informally, a sequence of snapshots of all variables during the execution of the program:

Definition 3. *A* state trajectory *of length k is a finite sequence of ordered pairs*

$$(n_1, s_1), (n_2, s_2), \ldots, (n_k, s_k)$$

where for each i, $1 \leq i \leq k$, n_i is a statement from P and $n_1, n_2, \ldots, n_k$ is a feasible run of P. Further, for all i, $1 \leq i \leq k$, s_i is a partial function mapping variables of the program P to their values. Each (n, s) from a state trajectory gives the values of variables from P before the execution of n.

We say that a program halts with a state trajectory T or that it produces a state trajectory T. To express what is interesting for us during a program's execution (and thereupon during slicing), we introduce the following definition.

Definition 4. *A* slicing criterion *of a program P is a tuple (n, V) where n is a statement in P and V is a subset of the variables in P.*

In our case, we are interested in values of some variables at some point in a program. Different slicing techniques can use different definitions of a slicing criterion. Every slicing criterion C determines a function $Proj_C$ that projects state trajectories to the states of variables given in C:

Definition 5. *Let $T = t_1, t_2, \ldots, t_k$ be a state trajectory. We define*

$$Proj_{(i,V)}(T) = Proj'_{(i,V)}(t_1), Proj'_{(i,V)}(t_2), \ldots, Proj'_{(i,V)}(t_k)$$

where $Proj'_C$ is defined as:

$$Proj'_{(i,V)}((n,s)) = \begin{cases} \varepsilon & \text{if } n \neq i \\ (s|_V) & \text{if } n = i \end{cases}$$

where ε is the empty string (deletion of the tuple) and $s|_V$ is the restriction of s to the domain V.

A projection of a state trajectory is therefore a sequence of states of memory before every execution of *i*. Now we are ready to define an executable program slice. Executable because we require the slice to be an executable program. Since we use slicing along with verification, this requirement is essential for us. However, many applications (e.g. debugging) do not need executable slices (they do not need a program, but only a set of statements from the slice).

Definition 6. *An* executable (backward static) slice *S of a program P with respect to a slicing criterion $C = (i, V)$ is any executable program with the following two properties:*

(1) S can be obtained from P by deleting zero or more statements from P

(2) Whenever P halts on an input I with a state trajectory T, then S also halts on the input I with a state trajectory T', and $Proj_C(T) =$

$Proj_{C'}(T')$, where $C' = (i', V)$, i' is i if i is in the slice, or i' is the instruction that is in the slice and that would be executed immediately after i if i would be in the slice. If there is no such i', we put i' to be the exit point of the program.

Note that a program is a (trivial) slice of itself. This slice is not very interesting, though. We will be concerned with searching slices as small (in a number of statements) as possible. Examples of different slices of the program from Figure 2.1 taken with respect to different slicing criterions can be found in Figure 2.2.

```
1.
2.
3. int sum = 0;
4. int x = read();
5. while(x > 0) {
6.   sum += x;
7.
8.
9.
10.  x = read();
11. }
12.
13.
14.
```

```
1. int n = 0;
2.
3.
4. int x = read();
5. while(x > 0) {
6.
7.   ++n;
8.
9.
10.  x = read();
11. }
12.
13.
14.
```

```
1.
2. int odds = 0;
3.
4. int x = read();
5. while(x > 0) {
6.
7.
8.   if (x % 2)
9.     ++odds;
10.  x = read();
11. }
12.
13.
14.
```

Figure 2.2: Three different slices of the program from Figure 2.1. Slices are taken with respect to slicing criterions (from left to right): $(14, \{sum\})$, $(14, \{n\})$, *and* $(14, \{odds\})$.

The definition of an executable slice intentionally ignores the cases when P fails to terminate normally[3]. Extending the definition to such cases brings many new problems [49]. For example, there is no guarantee that a slice of a program fails to halt whenever the original program fails to halt (see Figure 2.3 for an example). Therefore we consider only the normally terminating runs of programs. Nevertheless, it can be proven that slices computed by the algorithms that we introduce

3. A program fails to terminate normally if it diverges (loops infinitely) or it crashes, for example, as a result of a division by zero.

in this chapter (not slices in general) preserve the original program's behavior (with respect to a given slicing criterion) even when both, the program and its slice, diverge [7]. Reps and Yang also proved that when the original program halts on an input I, then also the slice computed by the algorithms halts on the input I. Moreover, the original program and the slice produce the same sequence of values for variables from the slicing criterion on *every* point in the program that is also in the slice [38].

```
1. x = read()                1.
2. while (x > 0)             2.
3.     x = 1                 3.
4. z = 3                     4. z = 3
5. print(z + x)              5.
```

Figure 2.3: A simple program and its slice with respect to a slicing criterion $(5, \{z\})$. For the input $x > 0$ the original program diverges, but the slice always terminates in a finite time.

2.1 A Data-Flow Approach to Program Slicing

In the previous section, we defined the notion of a slice. In this and the next section, we give algorithms for computing backward static slices. This section describes the original Weiser's algorithm. This algorithm was implemented in Symbiotic before we replaced it with an algorithm that uses a dependence graph. Since we are more interested in the description of the later one, we do not go into details of this algorithm. We also consider only sequential programs without pointers. Slicing programs with pointers is investigated in Section 2.3.

The Weiser's method uses a CFG of a program to perform slicing. Given a CFG and a slicing criterion $C = (i, V)$, the algorithm computes a set $R_C(n)$ of relevant variables for every node n (we refer to it as to the *relevant set for the node n*). The set of relevant variables for a node contains the variables whose value (transitively) affect the computation of variables from V at i. Along with the relevant sets, the set S of the statements that are in the slice is built.

Finding intraprocedural slices

We begin with describing slicing of single procedures (intraprocedural slicing). For a CFG of a program, let $REF(n)$ be a set of variables used at a program point n and $DEF(n)$ a set of variables defined at the program point n. Further, denote $succ(n)$ a set of immediate successors of the node n in the CFG. Given a slicing criterion $C = (s, V)$, we can compute a relevant set $R_C(n)$ for each statement n from the program as depicted in Algorithm 1 [49].

Algorithm 1 The computation of a relevant set $R_C(n)$ for a node n from a CFG and a slicing criterion $C = (i, V)$.

```
if $i = n$ then
    $R_C(n) \leftarrow V$
else
    $R \leftarrow (\cup_{m \in succ(n)} R_C(m))$
    $R_C(n) \leftarrow R \setminus DEF(n)$

    if $DEF(n) \cap R \neq \emptyset$ then
        $R_C(n) \leftarrow R_C(n) \cup REF(n)$
```

The algorithm starts at the slicing criterion and go backwards the CFG (from a node to its predecessors), computing the $R_C(n)$ for every node n of the CFG. In every step, it propagates what variables are still relevant to predecessors. If a node defines a variable v that is relevant, it removes it from relevant variables, because every other definition that occurs earlier in the program is overwritten by this one. However, the algorithm make new variables relevant if their value is used at a statement that defines a variable that is relevant. Alternatively, we can view this computation as looking for definitions of variables. A relevant set for a node n contains variables for which we still need to find a definition when we are on n. Once we find a definition of a variable v, we remove v from relevant variables and add all variables that are used in the definition of v instead. Due to the presence of loops in programs, we must iterate this computation until relevant sets stabilize (reach fixpoint). The set S of statements that are in the slice consists of all statements that define a variable that is in the relevant set of a successor, that is only if $DEF(n) \cap R(m) \neq \emptyset$ for m being a successor of n.

It this phase, the set S contains all nodes from the CFG that affect the value of the variables from the slicing criterion when they are executed. But whether they are executed depends on the predicates in the program that control their execution. For example, in the program from the Figure 2.1, the statement on the line 9 can influence the value of the variable *odds* only if the predicate on the line 8 evaluates to true. For this reason, we must add all the predicates that control the execution of statements from the set S to the slice along with statements they depend on: we take every predicate b that controls the execution of some statement from the set S and add all statements from a new slice taken with respect to a slicing criterion *(b, REF(b))*. This is equivalent to adding *REF(b)* to the $R_C(b)$ and repeating the computation starting in b. We must also explicitly add b to S. After this step, we get a new set S of the statements that are in the slice. This set can again contain statements for which we need to add a predicate, therefore we repeat the computation until a fixpoint is reached.

As an example, consider Figure 2.4 that shows the relevant sets computed for the program from Figure 2.1 with respect to the slicing criterion C = *(14,* $\{n\}$*)*. $R_C(14) = \{n\}$ from the definition (Figure 1). This relevant set is propagated undisturbed until `++n;` or `int n = 0;` statement is reached. The `++n` statement define n, but it also uses it, so $R_C(7) = \{n\} \setminus \{n\} \cup \{n\} = \{n\}$. The $R_C(1) = \{n\} \setminus \{n\} \cup \emptyset = \emptyset$ is computed for the statement `int n = 0;`. Relevant sets are stabilized and the set S after this step is $S = \{1, 7\}$ (drawn green in the picture). The execution of the statement `++n` on the line 7 is controlled by the predicate on the line 5, therefore we add 5 to S and *REF(5)* to the $R_C(5)$ and iterate the computation beginning on the node corresponding the line 5. The $R_C(5)$ is propagated to the immediate predecessors, where the computation is $R_C(10) = R_C(4) = \{n, x\} \setminus \{x\} \cup \emptyset = \{n\}$. A fixpoint is reached with the relevant sets stabilized in the depicted state. The newly added nodes are drawn yellow in the picture.

Computing interprocedural slices

Slicing programs with procedure calls (*interprocedural* slicing) is an important extension to the algorithm. To get a picture how the inter-

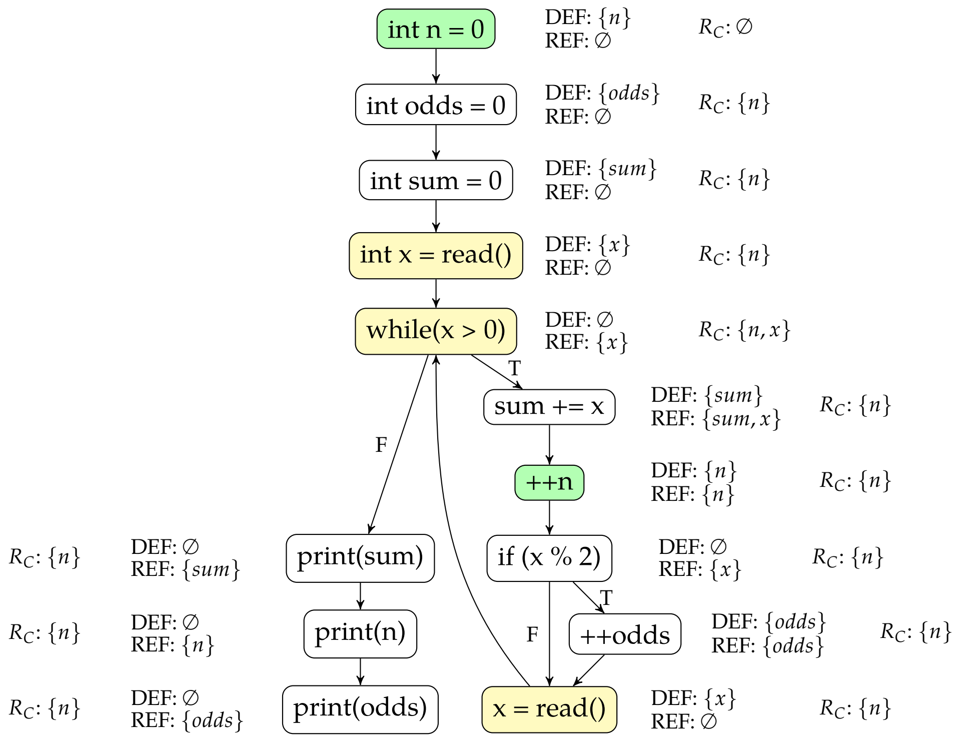


Figure 2.4: Relevant sets computed for the program from Figure 2.1 with respect to a slicing criterion $(14, \{n\})$. The nodes that are determined to have direct effect on the variable n on the line 14 are highlighted in green color and the nodes that are added to the slice because they control the execution of the green nodes are highlighted in yellow color.

procedural slicing works, imagine procedure inlining. When we inline a procedure, we replace its calls with its body and its parameters with the actual values that we pass to the call as arguments. The very same idea is behind the extension to the interprocedural slicing. When we run into a procedure call, we derive slicing criterion for the procedure from the relevant set of the statement following the call and from the variables passed to the procedure, and replace names of these variables with corresponding variables from the procedure. To show how it is done we examine this short piece of code:

```
1. a = 3
2. b = 2
3. call foo(a, b)
4. z = a + b
5. print(z)
```

```
6. foo(x, y):
7.     x = 5
8.     y = y + x
9.     return
```

We assume that we pass arguments by reference. When we slice with respect to the slicing criterion $(5, \{z\})$, we compute relevant set $R(5) = \{z\}$ and $R(4) = \{a, b\}$. The predecessor of 4 is the call of *foo*, therefore we use $R(4)$ to determine the values that we need to track in the procedure *foo*. The values are: $R(4) \cap ARGS(foo) = \{a, b\}$ where $ARGS(foo)$ are the arguments passed to the call of *foo*. But after passing these variables into the procedure, they are named differently. We must map the names of the actual parameters passed to the call to the names of formal parameters. In our case, we have $\{a, b\} \rightarrow \{x, y\}$. We create a new slicing criterion from these variables and the final location of the procedure, that is $(9, \{x, y\})$. This is the slicing criterion we use to slice the procedure *foo*. If there would be more call sites of this procedure, we create the slicing criterion with respect to all of them. Afterwards, we compute the relevant sets in the procedure *foo* and then determine the relevant set $R(3)$ for the call site itself. A simple idea is to include all the variables passed to the procedure call into this relevant set (plus the usual propagation of the relevant sets from the predecessors). That would work, but it could add unnecessarily too much into the slice. It is possible that we defined some passed value in the procedure (as it is the case in our example on the line 7) or that the value is not used at all. Therefore it is better to take the relevant set $R(7)$ of the initial statement of the procedure and add only these variables into the relevant set $R(3)$ (after names translation). Moreover, the variables that we passed into the procedure are already covered by the translated $R(7)$, so we can remove them from $R(3)$. The final computation is $R(3) = (R(4) \setminus ARGS(foo)) \cup \{b\} = \{b\}$, where the $\{b\}$ is the $R(7)$ translated.

Complexity of this algorithm is $O(n \cdot e \cdot log\ e)$ [49], that is $O(n^3 \cdot log\ n)$, where n is the number of nodes in the CFG and e is the number of edges in the CFG. Weiser also claims that practical results are much better.

2.2 Program Slicing Using Dependence Graphs

Another established slicing method uses a so-called *program dependence graph (PDG)*. Dependence graphs were introduced by Kuck [28] as a useful program representation for various program optimizations. Ottenstein and Ottenstein noticed that program dependence graphs are very suitable for program slicing due to an explicit representation of dependencies between statements in the program [35].

A program dependence graph is a directed graph[4] that has nodes from a CFG[5] and two kinds of edges – control dependence edges and data dependence edges. These two kinds of edges form two subgraphs in the PDG: a control dependence graph and a data dependence graph. Control dependence explicitly states what nodes are controlled by which predicate and data dependence bears an information about a definition-use relationship. A data dependence edge is between nodes n and m iff n defines a variable that m uses and there is no intervening definition of that variable on some path between n and m. In other words, the definitions from n reach uses in m (see Figure 3.13). An example of a PDG can be found in Figure 2.7.

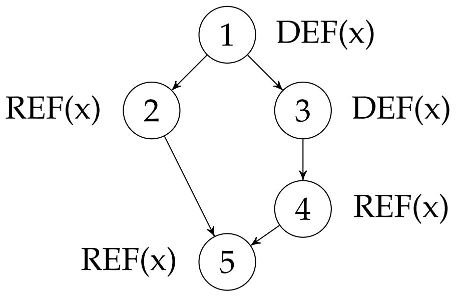


Figure 2.5: Reaching definitions. The definition of x at the node 1 reach uses at nodes 2 and 5, but not at the node 4. The definition of x at the node 3 reach uses at nodes 4 and 5, but not at the node 2.

4. Some publications consider a PDG to be a multi-graph [11, 24]. There can exist more dependence edges between two nodes arising from different aspects of the nodes semantics.
5. There exist more alternative variants of dependence graphs. For example, a dependence graph for an operation and its operands, or a dependence graph for a particular loop only [20, 28].

We deal with a construction of a dependence graph and with algorithms to compute control and data dependence edges in Chapter 3. Also, we do not consider programs with pointers and exceptions in this section. Extensions for such programs are in the next section.

We have explicitly said when two nodes are data dependent, but about control dependence we have talked rather vaguely. Let us recall control dependence more formally [20]. Before that, it is necessary to establish the definition of the post-dominance relation:

Definition 7. *Let (N, E, n_s, n_e, l) be a CFG, a node m* post-dominates *a node n in G iff m is on every path from n to n_e. If $A \neq B$, we say that m* strictly post-dominates *n. A node m is the* immediate post-dominator *of a node n iff m strictly post-dominates n and m is post-domintated by any other strict post-dominator of n.*

Immediate post-dominators form a post-dominator tree, which is a structure that has important role in the algorithms for computing the control dependence relation for a program. An example of a post-dominator the can be found in Figure 2.6. Now we can define the control dependence:

Definition 8. *Let G be a CFG, a node m is control dependent on a node n iff*

- *There exists a directed path from n to m such that every node on the path (excluding n) is post-dominated by m, and*
- *n is not strictly post-dominated by m*

If m is control dependent on n, then n must have two successors – following one always leads to the execution of m and following the other always avoids the execution of m [20]. Moreover, n is the "closest" of such nodes, therefore this control dependence handles correctly nested branching and loops.

Ferrante et al. use *augmented* CFG for computing control dependencies [20]. They add special *ENTRY* and *EXIT* nodes[6] and an edge between them. The edge has the effect that the *ENTRY* node is post-dominated only by the *EXIT* node and thus all the statements that

6. In fact, Ferrante et al. add only the *ENTRY* node as the *EXIT* node is already present from their definition of a CFG.

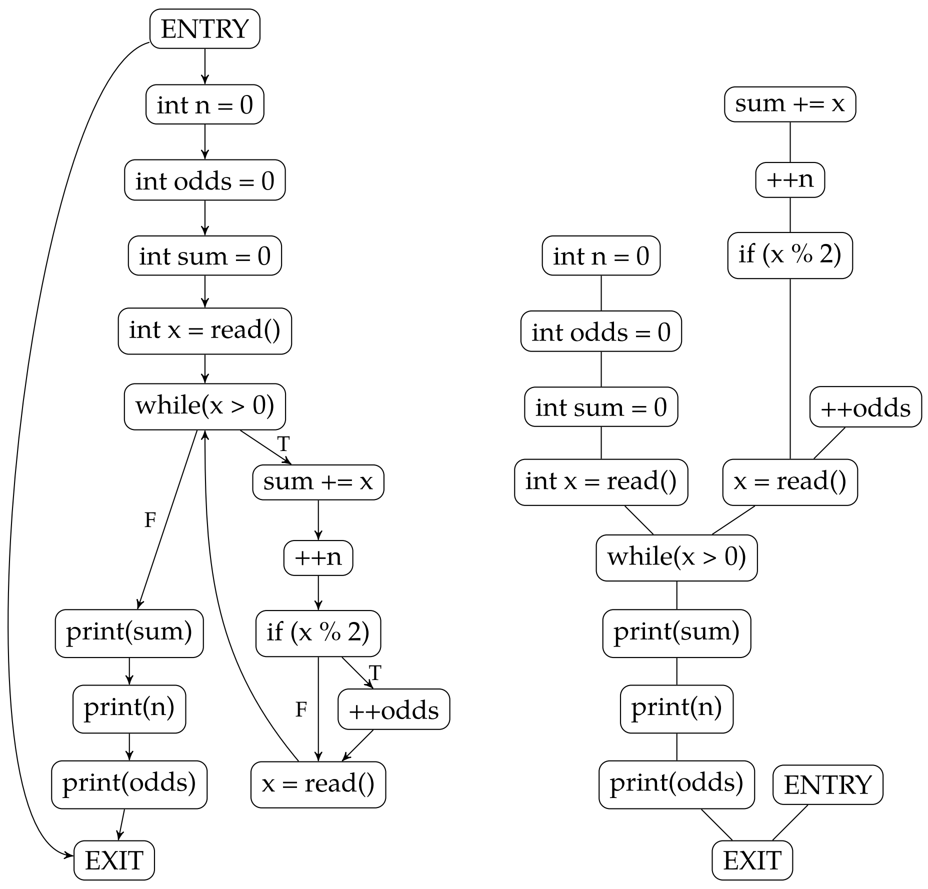


Figure 2.6: An augmented CFG of the program from the Figure 2.1 (left) and its post-dominator tree (right).

would not have control dependence in the orignial CFG are control dependent on the *ENTRY* node (the edge between *ENTRY* and *EXIT* makes the *ENTRY* behave as a predicate which is the "outermost" one, see Figure 2.6). In other words, in an augmented CFG, every node (excluding *ENTRY* and the *EXIT* nodes) is control dependent on some other node. Without augmenting the CFG, there are nodes that are not control dependent on any other node and the control dependence subgraph would be disconnected.

Ferrante et al. gave an efficient algorithm for computing control dependencies using a control flow graph and its post-dominator tree [20].

Two years later, Cytron and Ferrante et al. gave another algorithm that uses a notion of post-dominance frontiers [18]. We use the later one to compute control dependencies and it is described in Chapter 3.

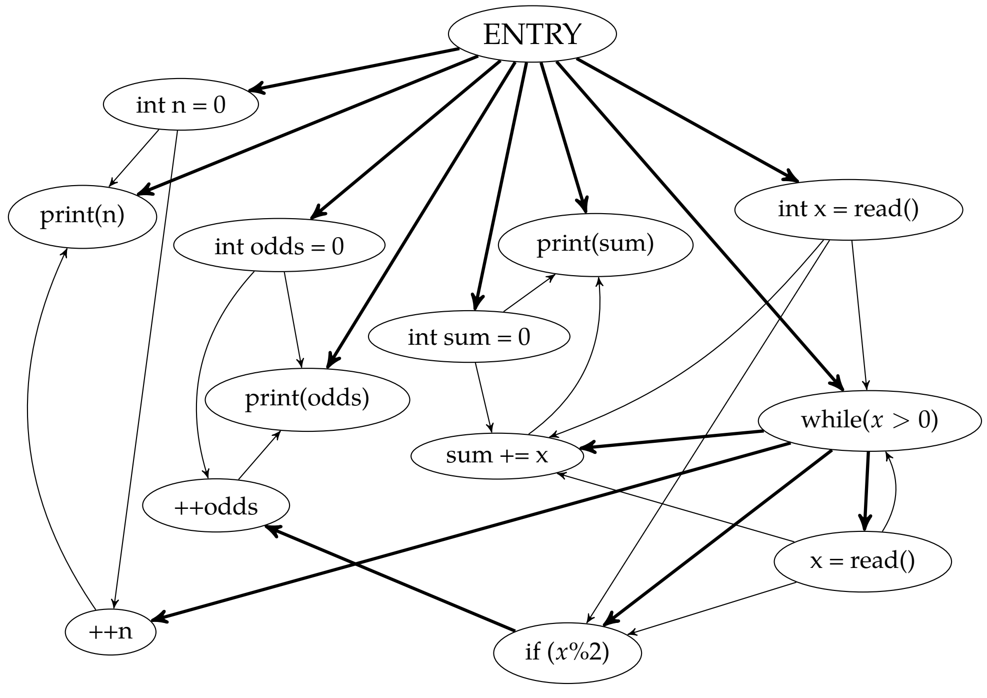


Figure 2.7: A program dependence graph for the program from Figure 2.1. Thick edges are the control dependence edges and thin edges are the data dependence edges.

Slicing programs using a PDG can be summarized into three steps [11]:

1. Construct a dependence graph of the program.
2. Slice the dependence graph.
3. Obtain sliced program from the sliced graph.

The step 1 include computation of control and data dependence edges, and it is the most demanding part of slicing using dependence graphs. We concentrate on this task in Chapter 3. Steps 2 and 3 are efficient: slicing a PDG is done by one or two (linear) searchings through the graph and mapping the sliced PDG can be done efficiently too. However, the later step may require some tricks, since we loose the control flow information when representing a program as a PDG. The usual method is to annotate nodes in a PDG to be able to map the nodes back to the original program (its source code), or to make a projection of the nodes back to the program's CFG. We can also reconstruct a CFG directly from a PDG, although it may look differently from the CFG of the original program [20].

The step 1 is of the same complexity as the analyses used to compute control and data dependence edges, which is at least $O(n^2)$ where n is the number of statements in the program [11]. This bound is tight for programs without pointers. With pointers, the pointer analysis and reaching definitions analysis are the main factors that set the complexity. Steps 2 and 3 have linear complexity in the number of statements of a program.

Slicing program dependence graph

Program dependence graph represents one procedure, therefore the slices we are about to talk are *intraprocedural* (as in Section 2.1, we first describe the intraprocedural slicing and than the interprocedural slicing). Slicing a PDG is very easy. We start with selecting a node (or a set of nodes) that serve as a slicing criterion. The reader can see that a slicing criterion for a PDG is different from the Weiser's slicing criterion – it is a node or a set of nodes in the PDG. It means that we are restricted to slicing with respect to only those variables that are used at the node or nodes in the slicing criterion (which is what we usually want anyway). We can get around this by creating an artificial node that uses all variables we are interested in, hence we get equivalent slicing criterion to the Weiser's tuple (s, V) [5]. Some authors add such an artificial node for every variable as a "final" use at the end of a program [24]. Consequently, a slicing criterion can be selected as a subset of these final uses (if we want to slice with respect to values of some variables at a program exit). After we selected a slicing criterion, we mark all the nodes that are backward-reachable from the slicing criterion along both data and control dependence edges (Figure 2.8). These nodes with induced edges form the resulting slice. The slice is not required to be executable by definition (indeed, the slice is a PDG, not a program), but making executable slices is possible with a little additional work as mentioned in the previous paragraph.

Interprocedural slicing

A straightforward extension to support interprocedural slicing would be to create a PDG for each procedure in a program and interconnect them with interprocedural dependence edges (i.e. an edge that goes to uses of a variable from reaching definitions of that variable even through boundaries of procedures) and with call edges that go from a call site to the procedure entry point as depicted in Figure 2.9 (returning a value from a procedure would be handled the same – as an edge from the definition of the returned value in the called procedure to uses of this value in caller procedure).

This naive approach has the advantage that the slicing algorithm is unchanged – we select a node or a set of nodes as a slicing criterion and then mark the nodes that are backward reachable along all edges from the slicing criterion. This method is imprecise – it ignores a calling context of the procedures. Since we have only one dependence graph per a procedure, all calls to a procedure are "summarized" in its only dependence graph and, for the same reason, the graphs behave as returning output values to every call site (for example, in Figure 2.9 the call of the procedure *inc* on the line 5 can not return to the line 4). Horwitz et al. introduced a method for creating more precise interprocedural slices [24]. They can rule out some unfeasible execution paths in a program even without fully context-sensitive analysis, with only one dependence graph per a procedure[7]. An extension of a program dependence graph – a *system dependence graph (SDG)*, is used to accomplish this task. An SDG consists of a program dependence graph [8] for each procedure along with these additions:

- Special *actual parameter* nodes are added to every call site and *formal parameter* nodes are added to the entry points of procedures. Actual parameters are made control dependent on the

7. An analysis is context-sensitive when it takes into the account a calling context of a procedure, which can be achieved (among others) by creating a distinct copy of a procedure for each call site of the procedure.
8. In this situation also called a *procedure dependence graph* [24].

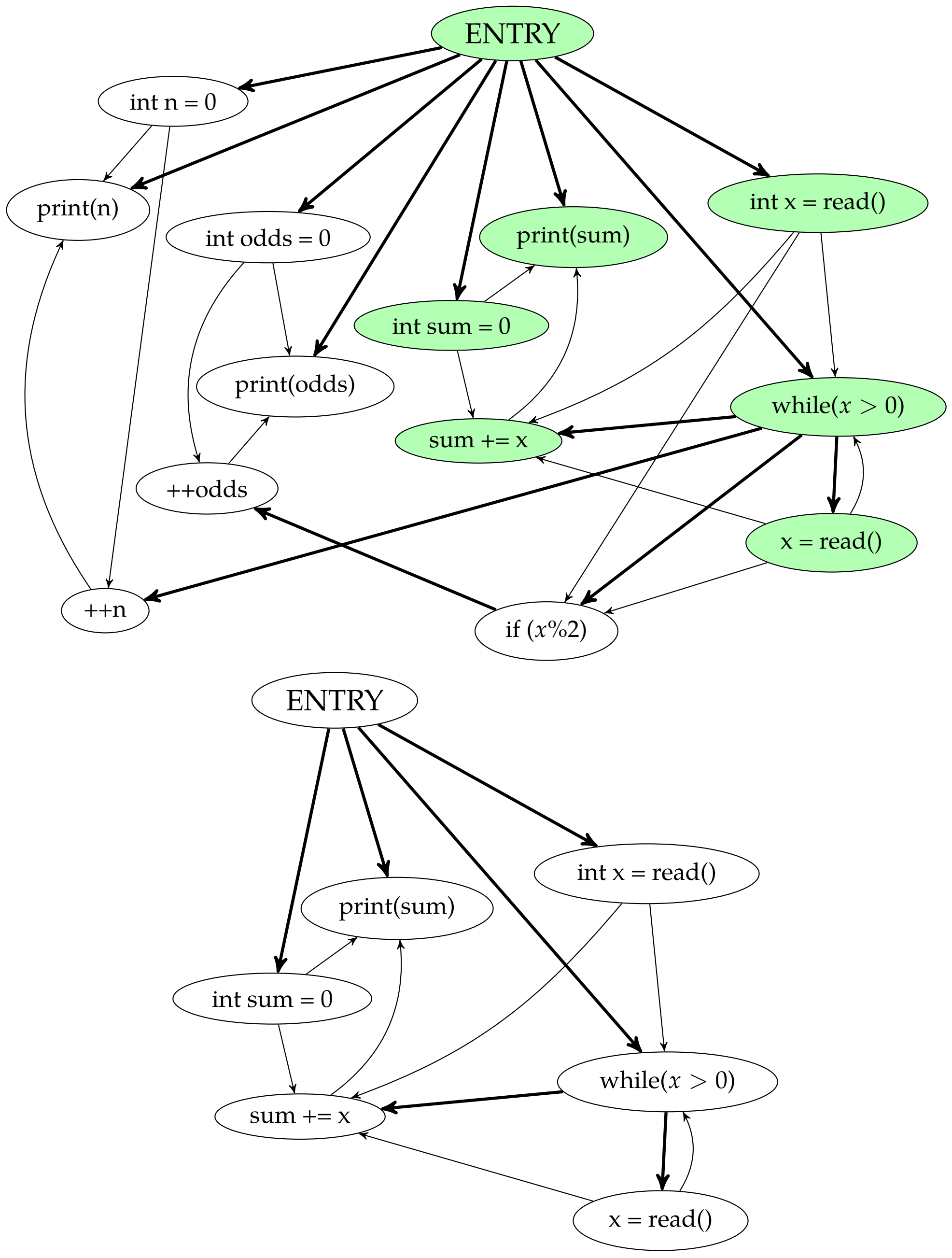


Figure 2.8: Nodes marked by the backward reachability from the node print(sum) *in the PDG from Figure 2.7 and the resulting sliced PDG.*

```
a = 0
b = 0
tmp = 0
inc(a,b)
inc(b,tmp)
print(b)
```

```
inc(x,y):
    x = x + 1
    y = y + 1
```

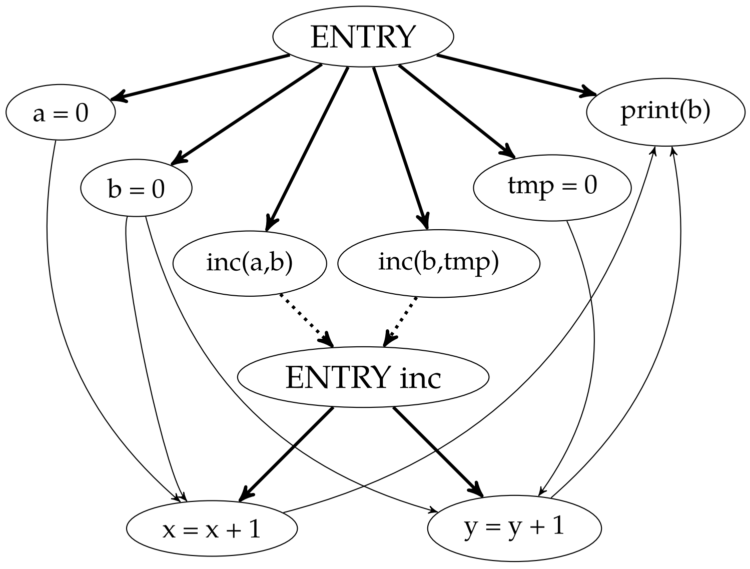


Figure 2.9: A naive version of an interprocedural program dependence graph for a program with procedure calls that connects uses with their (possible) reaching definitions with interprocedural edges. We assume that the parameters are input-output parameters (the modification of parameters is propagated back to a caller procedure)

call site and formal parameters are made control dependent on the entry point. Each parameter (both actual and formal) is doubled: once as an input parameter (represents values passed into a procedure from a caller via this parameter), and once as an output parameter (represents values passed from a procedure to a caller via this parameter). Horwitz et al. model the parameter passing by a value-result[9], which means that the actual parameters behave as assignments (see Figure 2.10), notably that the actual output parameters are definitions of the variables used as parameters.

There is a *parameter edge* from each actual input parameter to the corresponding formal input parameter (*parameter-in* edges) and from each formal output parameter to the corresponding actual output parameter (*parameter-out* edges). If a procedure returns a value, it can be modeled as an additional parameter. Moreover, global variables that are used or modified in the procedure are added as parameters too.

- *Summary edges* are added between actual input parameters and actual output parameters. These edges show (transitive) dependencies of the parameters. For example, if we use a formal parameter x to compute value returned from a procedure in a formal parameter y, there is an edge between actual parameters corresponding to x and y.
- A *Call edge* is added between a call site of a procedure and its entry point.

An example of an SDG can be found in Figure 2.10. Adding parameter nodes and parameter and call edges is straightforward. The computation of summary edges is slightly more challenging. Originally, these edges were computed using a special attribute grammar representing the parameters [24]. A faster algorithm was introduced by Reps et al. [39]. A summary edge is added from an actual input parameter A to an actual output parameter B whenever the formal output parameter corresponding to B is reachable from the formal input parameter corresponding to A along control dependence, data dependence and summary edges. Note that adding a summary edge can make new parameters reachable, which may trigger further addition of summary edges.

9. When a parameter is passed by value-result, it is copied from the actual parameter to the formal parameter at the procedure start and then the (possibly modified) formal parameter is copied back to the actual parameter at the procedure return.

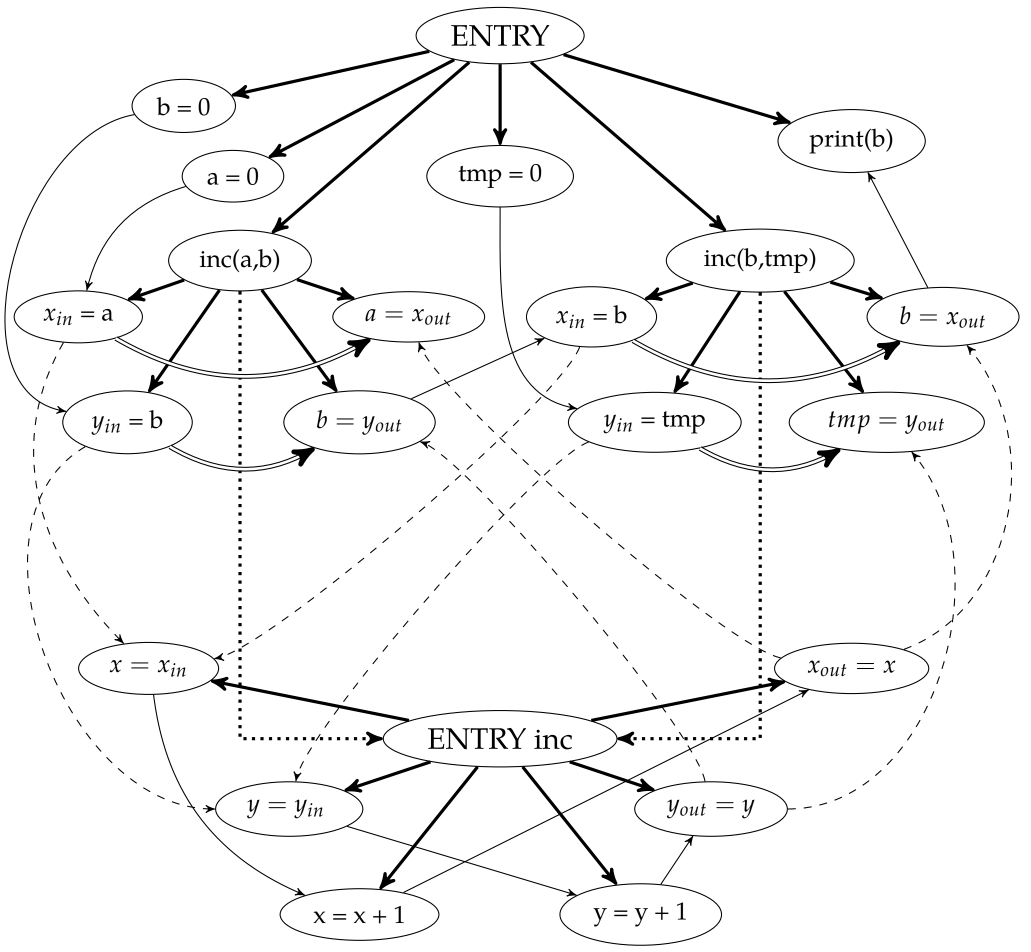


Figure 2.10: An SDG for the program from Figure 2.9. Thick edges are control dependence edges, thin solid edges are data dependence edges, dashed edges are parameter edges, dotted right-angle edges are call edges and finally the double edges are summary edges.

Same as the naive interprocedural version of a PDG, slicing system dependence graphs could copy the simple reachability algorithm known from PDG slicing and we would get correct slices [11] (see Figure 2.13). However, we can get more precise slices (the "context sensitive" ones) by slightly modifying the algorithm. Again, we mark nodes that are backward-reachable from a slicing criterion, but now we proceed in two passes (further in the text, we refer to this algorithm as the *two-pass algorihm*). In the first step, we go backwards along all edges but parameter-out edges. In the second step, we start in vertices reached in the first step and go backward along all edges but call and parameter-in edges (Figure 2.11). The nodes marked in these two passes with induced edges form the resulting slice (Figure 2.12). The step 1 identifies the statements that can affect the values at a slicing criterion without descending into procedures – that is possible due to the summary edges. Since we do not descend into procedures, we do not mark the nodes that are on infeasible return paths. Instead, we mark only the parameters that somehow contribute to the computation of values relevant to the slicing criterion using the summary edges. The step 2 finds the statements in procedures that we left out due to not descending into the procedures in the step 1 (the nodes that are supplied by the summary edges in the step 1).

Sliced SDG may not be an SDG anymore. For example, we may have removed an actual parameter of some call, but not the formal parameter (as we did in Figure 2.11 for the call `inc(a,b)`), which corrupts the definition of an SDG. If we want to continue working with an SDG, we may want to (transitively) add these missing pieces along with their dependencies, so that the result is an SDG again [11]. Similar steps must be taken to correctly map an SDG to the original program in order to create executable slice [9, 11].

This algorithm can be further extended. Recently, Aung et al. introduced a so-called *specialization slicing* [4]. Specialization slicing transforms an SDG into push-down automaton and unrolls it to find calling context patterns. The unrolled automaton is then sliced. The result is a program that has every procedure specialized for the calling context (the procedures are duplicated and sliced into different versions). This approach can handle even mutual recursion and correctly creates more versions of the recursive procedures that call each other. Sliced programs may be greater than original programs, but they contain

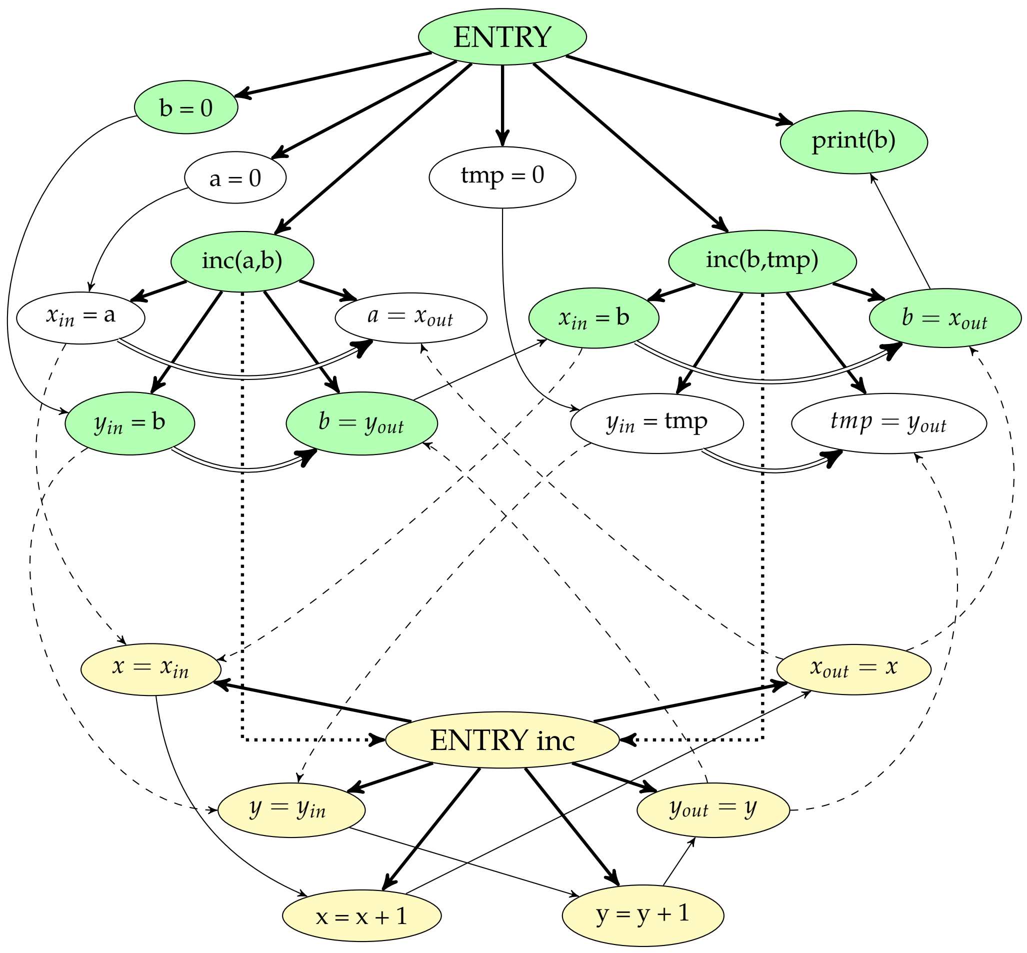


Figure 2.11: Nodes marked with the two-pass algorithm in the SDG of the program from Figure 2.9. The green nodes were marked by the first pass of the algorithm, and the yellow nodes by the second pass of the algorithm. The highlighted nodes with induced edges for the resulting slice.

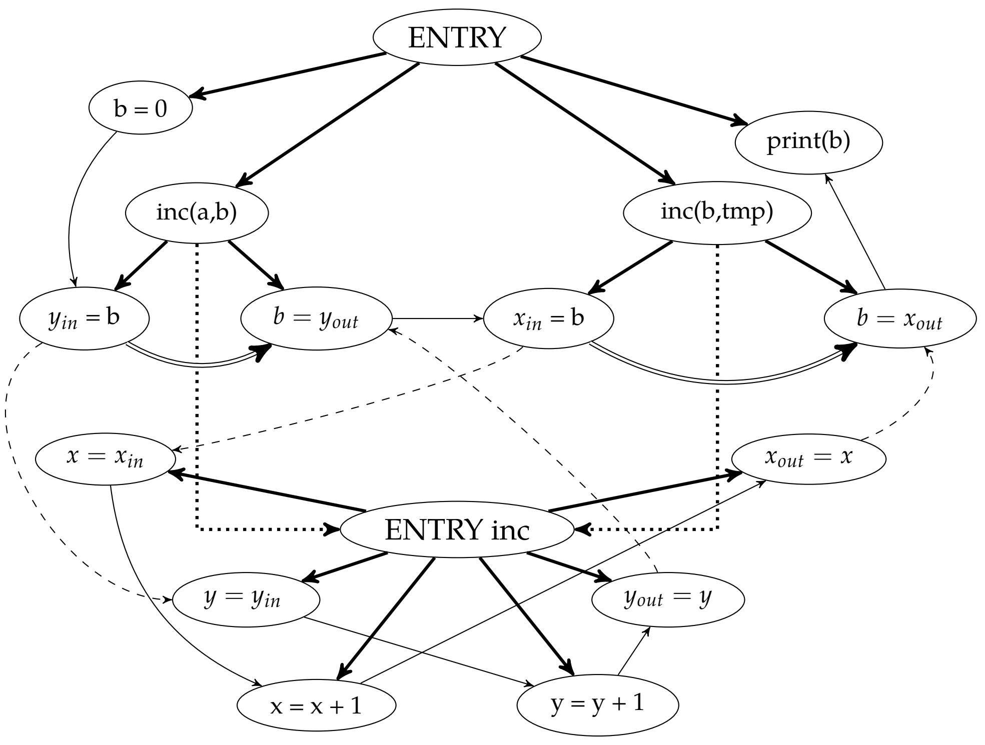


Figure 2.12: A sliced SDG from Figure 2.11.

only code that is necessary for the computation of the values in the slicing cirterion – they are specialized with respect to the slicing criterion (see Figure 2.13).

2.3 Slicing Programs with Pointers and Unstructured Control Flow

Until now, we talked about slicing sequential programs that do not contain pointers. Real programs use pointers on daily basis, which is an obstacle for slicing algorithms. Another problem is unstructured control flow – either intraprocedural that is introduced by using a *goto* statement and alike, or interprocedural, that arises when using exceptions, *longjmp* function in C, or *exit* system call and its alternatives inside procedures [42]. A bit special is the case of concurrent pro-

```
1.  a = 0                7.  inc(x,y):
2.  b = 0                8.    x = x + 1
3.  tmp = 0              9.    y = y + 1
4.  inc(a,b)
5.  inc(b,tmp)
6.  print(b)
```

```
1.                       7.  inc(x,y):
2.  b = 0                8.    x = x + 1
3.                       9.    y = y + 1
4.  inc(a,b)
5.  inc(b,tmp)
6.  print(b)
```

```
1.                       7.  inc1(y):
2.  b = 0                8.    y = y + 1
3.
4.  inc1(b)              9.  inc2(x):
5.  inc2(b)              10.   x = x + 1
6.  print(b)
```

Figure 2.13: Different slices for the program from Figure 2.9. A slice obtained using a naive version of an interprocedural PDG (top) is the same as the slice obtained using an SDG with simple backward reachability (the whole program). The slice obtained using the SDG with two-pass algorithm is depicted in the middle. At the bottom is a specialized slice obtained using specialization slicing.

grams. New dependencies emerge when threads can access shared memory. For all these cases, the algorithms hitherto mentioned could produce incorrect slices. In this section, we describe extensions to the dependence graph based algorithm that allow to slice programs with such features correctly[10] – since we are interested in slicing programs using dependence graphs, we omit the description of the extensions to the Weiser's algorithm (nevertheless, they usually follow the same idea). Dependence graphs are more suitable for such extensions, because unlike the data-flow algorithm of Weiser, we usually do not have to change the way of slicing. We only need to add new edges to a dependence graph, but the slicing algorithm is preserved.

Slicing programs with pointers

In programs without pointers, we could compute data dependencies syntactically. For programs with pointers it is not enough, because we could miss writes into variables via pointers. To account for such indirect definitions, we must have a sound information where every pointer may (or must)[11] point to during the program's execution. Therefore in order to compute correct slices, we must perform a pointer (or points-to) analysis first. Consider this C code fragment:

```
1. p = &a;
2. a = 2;
3. *p = 13;
4. print(a);
```

When slicing with respect to $(4, \{a\})$, the only lines that can be sliced away to get correct slice are 2 and 4. When a slicing algorithm does not know that `*p = 13` is a definition of the variable a, it may incorrectly slice away line 1 and 3.

Pointers change data dependencies: a data dependence edge is between nodes A and B when A possibly writes into a memory location and B may use the value possibly written by A to that memory location. We can obtain this information by one of many techniques for computing reaching definitions (or generally a data-flow analysis) in

10. Slicing concurrent programs is more challenging and it is out of scope of this thesis, therefore we omit the description of slicing programs with threads. To give an interested reader a reference, more information can be found, for instance, in [15], [26, 27], or [33].

11. A pointer *may* point to a memory location if it points to the memory location during some execution of a program. A pointer *must* point to a memory location if it points to the memory location during every execution of a program

the presence of pointers like [36] or [47]. We go into details regarding determining data dependence edges in the presence of pointers in Chapter 3. The precision of slicing depends mainly on the pointer analysis used. Consider a PDG for our example code (Figure 2.14). If the analysis knows that `*p = 13` is always definition of a (i.e. p must point to a), the PDG will not contain the edge from `a = 2` to `print(a)`, since the `a = 2` definition is determined to be overwritten by the `*p = 13`. On the other hand, if the analysis knows nothing more than that p may point to a, there must be an edge from `a = 2` as well as from `*p = 13` to `print(a)`, since both nodes may (with the information given) be a definition of a during some execution of the program.

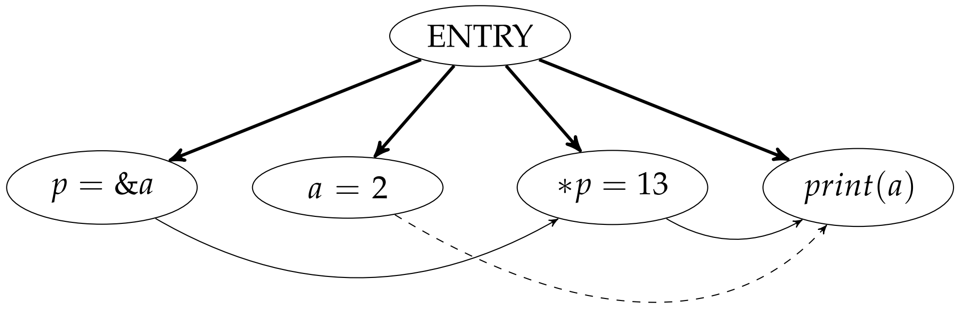


Figure 2.14: A PDG for the example program with pointers. Depending on the pointer analysis, there will or will not be the dashed data dependence edge. Both cases are correct, the later is more precise.

To correctly handle pointers with procedure calls, we must modify an SDG. Instead of adding parameter nodes syntactically according to a prototype of a called procedure, we compute two sets for each procedure p [24, 34]: *GREF(p)* and *GMOD(p)*. The set *GREF(p)* is a set of parameters of p and non-local variables that may be used inside p, and *GMOD(p)* is a set of parameters of p and non-local variables that may be modified inside p [6]. An input (formal and actual) parameter is then added for every variable in *GMOD(p)* ∪ *GREF(p)* and an output parameter is added for every variable in *GMOD(p)*. Summary edges are added in a similar fashion: an summary edge is added between actual parameters when the value of an output parameter may depend on a value of the input parameter.

Pointers do not change only data dependencies in the graph, but they can change even the flow of control. When a program contains calls of functions via function pointers, we must construct an SDG (or a CFG for Weiser's algorithm) only after a pointer analysis was performed to include such (possible) calls.

Slicing programs with unstructured and interprocedural control flow

Programs that contain an interprocedural control flow (i.e. exceptions or `longjmp` function calls in C) or call of the `exit` system call inside a procedure[12] may be sliced incorrectly with hitherto mentioned algorithms. Consider this samlpe code:

```
1. i = read()              4. check(x):
2. check(i)                5.   if (x < 0)
3. print(i)                6.     exit()
```

When slicing with respect to the line 3, the slicing algorithm slice away the whole procedure `check` and its call on the line 2. This is not a correct slice by the definition, since for any input i such that $i < 0$, both the original program and the slice halt, but the state trajectory of the slice is not equal to the state trajectory of the original program projected with respect to (3, $\{i\}$). Similar problem can arise when using exceptions or `longjmp` function in C, where a call of a procedure may not return to the call site.

To slice such programs correctly, we must make all statements that follow the potentionally non-returning call site control dependent on the statement that makes the execution stop or jump between procedures (i.e. the `exit` call in our example). The newly added control dependence edges are interprocedural – they go between procedures boundaries.

12. Here we must assume that we weakened the definition of a CFG: we do not have a single distiguished exit node that is reachable from every other node of the CFG anymore, but we may have "blind" ends scattered through the CFG.

Sinha et al. gave also a modification of the two-pass algorithm that can handle these cases correctly [42].

Similar problem is that slicing can remove a non-terminating loop and thus make unreachable code reachable (it is the same problem as with the call of the `exit` system call from a procedure, but now the program loops indefinitely instead of immediately terminating). This can be solved by the same trick: nodes that follow the loop are made control dependent on the loop's predicate (it is a so-called *weak control dependence*) [8, 14].

Programs that contain unconditional jumps like a `goto` statement and its restricted variants `break` and `continue` (and alike) may be projected incorrectly back to the source code. If our final representation is a CFG, we are free of this problem. The problem is that the unconditional jumps do not have any data nor control dependencies and thus are always sliced away. In a CFG, we do not need the unconditional jumps, since the flow of control is expressed by the edges, but if we omit a `goto` in a source code, the program change. This can be solved by special transformations of a CFG before slicing [5, 17] or by adding special "goto" nodes behaving like predicates into a PDG [17]. The later can yield smaller slices, but the slices are not subprograms of the original program due to the added "goto" nodes. Agrawal proposed a method [1] that yields results equivalent to the CFG transformations [17], but keep a CFG and a PDG untouched. Instead, it uses an auxiliary structure called *lexical successor tree* while slicing. Other solutions and improvements can be found in [30], [22] and [29].

3 Implementation

Before the actual description of the implementation of the slicing algorithm, we must give the reader a brief introduction of the infrastructure we used to implement it.

LLVM

LLVM [45] is a widely used compiler infrastructure. The name used to stand for "Low-level Virtual Machine", but nowadays LLVM is not an acronym but full name of the project. It began as a research project at the University of Illinois [31] in 2004.

LLVM consists of a code representation called LLVM IR (intermediate representation), which is an architecture independent language with types, and of a set of libraries and tools for generation, and manipulation and optimization of the IR. Simplified, we can think of a LLVM IR as an architecture independent assembler with types. LLVM IR can be held as a pure text (and thus be handled as a source code) or as an in-memory or an on-disk bitcode. All these forms are equivalent. Libraries for manipulation of the IR are written in C++, but a lot of standardly used languages have bindings to these libraries, therefore we can manipulate with a LLVM bitcode from other languages too.

LLVM bitcode is structured into modules that correspond to compilation units (source files). A Module consists of global variables and function definitions. A function is built up from basic blocks[1], which form a CFG: every basic block is terminated with a jump to another basic block or with a return instruction (returning from the function). In Figure 3.1 we give an example of a program in LLVM IR that prints "Hello, world!" in an infinite loop.

A strong point of LLVM IR is a type system. It is used not only to ensure that a (bit)code is well-formed, but even to enable some optimizations and easier analysis. LLVM has few single types that

1. A basic block is sequence of nodes of a CFG $n_1, n_2, \ldots, n_k$, such that for every i, $1 \leq i < k$, n_{i+1} is the immediate successor of n_i in the CFG, and every n_{i+1} has exactly one predecessor which is n_i.

```
declare i32 @puts(i8*)

@.str = private constant [14 x i8] c"Hello, world!\00"

define i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  br label %2

  ; <label>:2
  %3 = call i32 @puts(i8* getelementptr inbounds
                     ([14 x i8], [14 x i8]* @.str,
                     i32 0, i32 0))
  br label %2

  %5 = load i32, i32* %1
  ret i32 %5
}
```

Figure 3.1: Example of a LLVM IR that prints "Hello, world!" in an infinite loop. Basic blocks of the function `main` are highlighted in light green color

can be used to form derived types. Single types are `void`, `iN`, where N is the number of bits in the range from 1 to $2^{23} - 1$ (for example `i32` or `i123`), and floating point types `float` and `double` (and few more architecture dependent types that we do not need). From these can be further derived pointer types (`i32*` or `i8**`, etc.), function types (e.g. `i32 (i32, i8*)`) and finally aggregate types, such as structure types (e.g. `{i32, i32*}`) and array types (e.g. `[13 × i32]`). LLVM does not have a notion of `void *` type, it uses `i8*` instead.

We can divide variables in LLVM into two sorts: top-level variables (or values) and address-taken variables (memory locations)[2]. Top-level variables are similar to CPU registers in assembler. Address-taken variables are the same as variables in C programming language (these are nothing more than an allocated piece of memory). Every address-taken variable is created using the `alloca` instruction that returns the

2. LLVM documentation do not use any special term for variables in memory (and top-level variables are called simply values or registers). We borrowed the terminology from [21], as we benefit from it in the description of a pointer analysis.

address of the allocated memory. The address-taken variable can be access solely through this address. The address can be stored in different top-level variables, however, there is at least one top-level variable that points to the address-taken variable (the *alloca* instruction). In our example code (Figure 3.1), `%1` is a top-level variable pointing to the address-taken variable created by the `alloca` instruction.

Every top-level variable is unique and can not be reused: they are kept in a so-called *static single assignment* form (SSA) [2, 40]. In the SSA, every variable can be assigned only once in the source code. Assigning a variable on more sites in a program creates new versions of the variable in the code. When two or more different versions of a variable reach a program location, we create new version using a special *phi* function that represents a selection of the right version. See Figure 3.2 for an example.

```
int a;
if (...)
  a = 3;
else
  a = 4;
print(a);
```

```
int a0;
if (...)
  a1 = 3;
else
  a2 = 4;
a3 = phi(a1, a2);
print(a3);
```

```
%a = alloca i32
br ... , %l1 , %l2

; l1
store 3, %a;
%a1 = load %a
br %l3

; l2
store 4, %a;
%a2 = load %a
br %l3

; l3
%4 = phi [%a1 , %a2]
call print(%4)
```

Figure 3.2: A simple code and its SSA variant and equivalent code in LLVM.

The benefit of the SSA form is that every use of a top-level variable has exactly one reaching definition. This definition-use (def-use) associations are explicitly maintained in a LLVM bitcode. Top-level variables are *virtual* in the sense that they do not correspond to any physical part of a computer (although there is a similarity with registers). They can contain arbitrary large value and there can be any number of them. Top-level variables represent results of computations (instructions). Every operation in LLVM is decomposed in a sequence of instructions, where intermediate computations are stored in top-level variables. Mainly, a multiple pointer dereference is decomposed into a sequence of single pointer dereferences.

A global variable is always a pointer to memory. In text form, global variables (and functions) are prefixed with "@" and other variables (the top-level variables) are prefixed with "%" (the address-taken variables do not have any identificator).

To get a pointer into a structure or array, LLVM use `GetElementPtr` instruction that takes a pointer and shifts it by a given number of elements according to the type of the pointer (e.g. shift by one element in an array of the type `[10 x i32]` is a shift by 4 bytes). The only instructions that work with memory are `Store`, `Load`, `InsertElement` and `ExtractElement` (and some special vector instruction that we intentionally skip, as we do not plan to support them yet). The later are an atomic combination of the `GetElementPtr` and the `Store` (`Load`, respectively). Also, LLVM has *intrinsic* functions (undefined functions that have known semantics in LLVM) like `memcpy` or `memmove` that manipulate with memory too.

There are many other aspects of LLVM that would be worth to mention, but as these are not important for further text, we skip them.

A dg library

We created a library *dg* (stands for *dependence graph*) and implemented the slicing algorithm as a part of it. The *dg* is written in C++ and is designed as a set of generic templates and algorithms that can be instantiated to user's needs, and thus a dependence graph for a given backend and underlying analyses can be created with a minimal effort. So far, we support only LLVM as the backend. In this chapter, we describe the algorithms used to compute control and data dependence edges and their implementation, as well as the building process and slicing of a dependence graph itself.

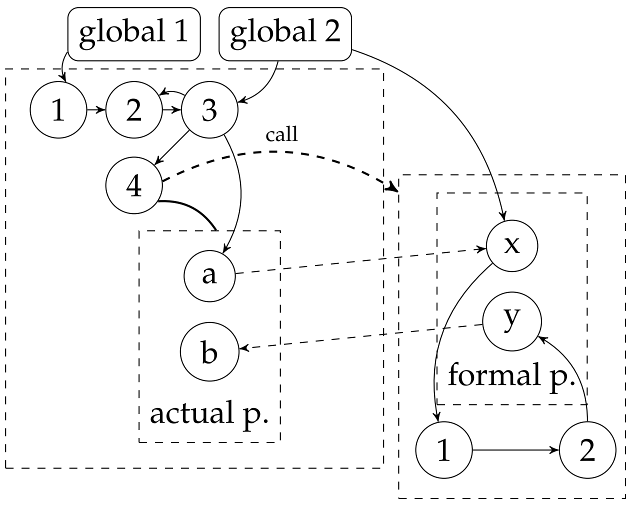


Figure 3.3: A generic dependence graph in the dg. *When a node has a pointer to another dependence graph, it is considered a call node and it can have actual parmeters (the node 4). Formal parameters may be associated to every dependence graph. Global nodes are shared between graphs. Dependence edges can go between arbitrary nodes.*

Our dependence graph is designed to be highly customizable. We do not differentiate between a main procedure and subprocedures as an SDG does: we have one class that represents a dependence graph of a procedure. Each dependence graph contains a map of (local) nodes for a quick lookup by a unique key. Moreover, it has a pointer to another map that contains "global" nodes that are visible to all dependence graphs. The distinction on local and global nodes is up to the user. Dependence graphs are not disallowed to use only the global nodes map, so that every graph has access to all nodes, or only local nodes maps to completely separate the graphs. A node can contain pointers to dependence (sub)graphs, in which case it is considered a call node. A call node may contain an (actual) parameters object. A parameters object is an object with a few auxiliary methods that only wraps the parameter nodes (which are usual nodes). A formal parameters object may be assigned to each dependence graph, representing the arguments of the procedure. See Figure 3.3 for a demonstation. The nodes in parameters can shadow the local nodes, which, in turn, can shadow the global nodes. In other words, the nodes lookup procedure proceeds in this order: try find a node in parameters, if not found then try in local nodes and only after that in global nodes.

Every node can bear data and control dependence edges to other nodes. These edges are double-linked – each node knows on which other nodes it is dependent, and which nodes depend on it.

Optionally, a dependence graph may contain also basic blocks (additionally to nodes). A basic block is implemented as an object that contains a list of nodes from a dependence graph and a set of labeled edges to successor and predecessor basic blocks. Basic blocks are useful when a dependence graph needs an access to the CFG for some reason. For example, we have implemented a data-flow framework that works on dependence graphs that use basic blocks. We used this framework for computation of data dependence edges earlier, but now we use another implementation of those analyses. Details are in Section 3.2. A basic block can also contain control dependencies, which can save a lot of edges that would normally go from each node in the graph (all nodes in a basic block have the same control dependencies).

Our implementation of a LLVM dependence graph has a dependence graph with local nodes for each function and global variables are shared as global nodes between graphs. The globals that are used in procedures are added also as parameter nodes. We use basic blocks that exactly correspond to blocks from LLVM. The control dependencies are kept in basic blocks. An example of a LLVM dependence graph can be found in Figure 3.14.

The slicer implements only the naive interprocedural slicing, as we have not implemented the two-pass algorithm of Horwitz et al. yet.

The *dg* library is available at `https://github.com/mchalupa/dg` under the MIT open-source licence. The installation and usage instructions can be found in project's README.

3.1 Computing Control Dependencies

Control dependencies are the first kind of edges we need to compute to build a dependence graph. We implemented the algorithm from [18] that uses so-called post-dominance frontiers. This is an alternative

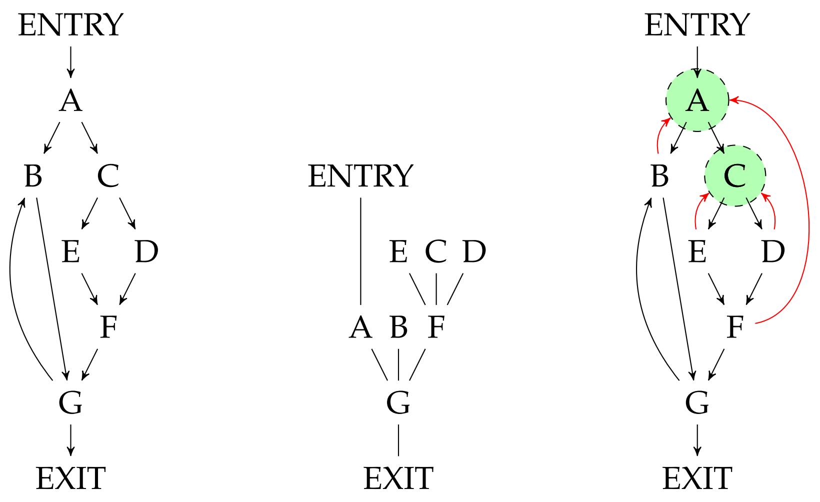


Figure 3.4: A Control flow graph, its post-dominator tree and post-dominance frontiers highlighted.

method for computing control dependencies to the original algorithm from [20].

Definition 9. *Let G be a CFG. A* post-dominance frontier *of a node $x \in G$ is a set of all nodes $y \in G$ such that x post-dominates a successor of y, but not strictly post-dominates y.*

Intuitively, the post-dominance frontier of a node x is a set of nodes where the post-dominance of x ends. An example can be found in Figure 3.4: the node A is in post-dominance frontier of the node F, because F post-dominates C [3], which is a successor of A, but does not (strictly) post-dominate A. C is in post-dominance frontier of the node D, because D post-dominates itself, but does not strictly post-dominate C.

Nodes from a post-dominance frontier of some node x are exactly the nodes on which is x control dependent [18], hence computing

3. A common practice is to draw a post-dominator tree as we did: root at the bottom and nodes going upwards. The parent-child relationship is also reversed – G is a child of $EXIT$, not the other way. The semantics of the tree is that a node immediately post-dominates its childs, and since the post-dominance is transitive and reflexive, it post-dominates all its descendants and itself.

control dependencies is the same as computing a post-dominance frontier for every node. Ferrante et al. use an augmented CFG to secure that every node has a control dependence – either on a real node of a CFG or on the artificial node *ENTRY*. The same trick could be used with post-dominance frontiers, but we do not do that. To the result of slicing, it does not matter whether a node has no control dependence or is control dependent on an artificial node that is not projected to the sliced program anyway.

For a given node x from a CFG, we could compute the post-dominance frontier of x from the definition in linear time by scanning the CFG and ins post-dominator tree. Consequently, for all nodes we have a naive algorithm that runs in quadratic time in a number of nodes. Cytron et al. gave a better algorithm that has the same worst-case complexity [18], but in practice behaves linearly in a number of nodes of a CFG. The algorithm makes use of two sets that are computed for every node before the actual inference of post-dominance frontiers. In the following text, we use *PDF(x)* to denote the post-dominance frontier of a node x, *pred(x)* to refer to a set of immediate predecessors of a node x in a CFG, *children(x)* to refer to a set of children of x in a post-dominator tree, $x \rightarrow_{pd} y$ is used to say that x post-dominates y, and finally, *ipdom(x)* stands for the immediate post-dominator of a node y. Let us define the two sets:

$$PDF_{\text{local}}(x) = \{y \in pred(x) \mid x \not\rightarrow_{pd} y\}$$

$$PDF_{\text{up}}(x) = \{y \in PDF(x) \mid ipdom(x) \not\rightarrow_{pd} y\}$$

The rationale for the $PDF_{local}(x)$ set is that whenever we have a predecessor of a node x that is not post-dominated by x, then it belongs to the *PDF(x)* (it follows from the reflexivity of the post-dominance [18]). Therefore $PDF_{\text{local}}(x) \subseteq PDF(x)$. Similarly, if we have a node z in a CFG, we can ask what nodes are common for the z's post-dominance frontier and for the post-dominance frontiers of its children in the post-dominator tree. It is exactly the PDF_{up} set (for $x \in children(z)$). Cytron et al. proved that the post-dominance frontier of a node x can be composed of *local* and *up* sets in the following way:

$$PDF(x) = PDF_{\text{local}}(x) \cup \bigcup_{z \in children(x)} PDF_{\text{up}}(z)$$

We can see that to compute the post-dominance frontier of a node x it is sufficient to know post-dominance frontiers of all its children in the post-dominator tree. This leads directly to the shape of the algorithm: we go from leaves towards the root of a post-dominator tree (in the reverse BFS order), computing sets PDF_{local}, PDF_{up}, and the *PDF(x)* eventually. The last thing we must take care of is the test for the post-dominance relation: when computing *local* and *up* sets, we perform tests whether some node x post-dominates some other node y. This queries can be solved quickly by an equality check, because given a node x, then for $y \in pred(x)$ or $y \in PDF(z)$, where z is a child of x, we get [18]:

$$x \rightarrow_{pd} y \iff idom(y) = x$$

To check whether $x \not\rightarrow_{pd} y$, we simply check that $idom(y) \neq x$. The whole algorithm is depicted in Algorithm 2.

Algorithm 2 The computation of post-dominance frontiers.

```
for X in a reverse BFS order of the post-dominator tree do

    PDF(x) ← $\emptyset$
    for all y $\in$ pred(x) do
        if idom(y) $\neq$ x then
            PDF(x) ← PDF(x) $\cup$ y

    for all z $\in$ children(x) do
        for all y $\in$ PDF(z) do
            if idom(y) $\neq$ x then
                PDF(x) ← PDF(x) $\cup$ y
```

The algorithm for the computation of post-dominance frontiers needs an access to the post-dominator tree of a program. LLVM can compute a post-dominator tree using the Lengauer-Tarjan algorithm [32], therefore we let LLVM to compute the post-dominator tree for us. LLVM is formed into basic blocks, therefore the post-dominator tree is computed for the blocks. We copy the immediate post-dominator edges of LLVM blocks into our basic blocks in a dependence graph, and run the algorithm 2 to compute post-dominance frontiers. The algorithm is implemented as a template, so it does not care what is a node and works also on the blocks of the dependence graph.

The algorithm assumes that we gave it a CFG as defined in Chapter 2. The cruical assumption in this case is that a CFG has a distinguished *exit* node from which is reachable any other node of the CFG. If there is no such node (e.g. the program may contain an infinite loop of the form `while(1) {...}`, which creates a loop without any exit point in the CFG), we can not construct a post-dominator tree and therefore we are not able to compute control dependencies. In this case, we fall-back to a conservative approximation: we make every basic block in the program control dependent on its predecessors. It is sound, but very imprecise solution. For example, in Figure 3.4, if we would want to keep the node (block) F in the slice, we would have to keep also E, D, C and A instead of only A with this fall-back solution. This problem with control dependencies has been addressed for example in [19].

3.2 Computing Data Dependencies

The other kind of edges we need to incorporate into a dependence graph are data dependence edges. Because LLVM keeps def-use chains for top-level variables, it is necessary to compute only the indirect data dependencies of address-taken variables. Address-taken variables can be accessed solely via pointers, therefore we must perform a pointer analysis in advance of the actual inference of indirect data dependencies.

3.2.1 Pointer Analysis Framework

The output of a pointer analysis is the information where into memory may a pointer point during (any) execution of the program[4] – a *points-to set* for every pointer. This information can be further classified as flow-sensitive or flow-insensitive. The difference is that a flow-sensitive analysis gives the points-to information for every location in

4. In contrary to an alias analysis, which gives an answer to queries: "Can pointers p and q points to the same memory?", or, respectively, gives a list of pairs (p, q) where p and q are aliases.

the program separately, whereas a flow-insensitive analysis computes one mapping that holds at every point in the program (it ignores the flow of control). See Figure 3.5 for an example. Another aspect of a pointer analysis is how it handles aggregate data types. If it takes an array or a structure as one variable, we call it *field-insensitive*. If it considers structures and arrays as to be composed from subelements and computes points-to information with respect to every subelement, we call it *field-sensitive*[5].

Program	Flow-Sensitive	Flow-Insensitive
1. p = &a	$p \to \{a\}$	$p \to \{a,b\}, q \to \{a,b\}$
2. q = &b	$p \to \{a\}, q \to \{b\}$	$p \to \{a,b\}, q \to \{a,b\}$
3. q = p	$p \to \{a\}, q \to \{a\}$	$p \to \{a,b\}, q \to \{a,b\}$
4. p = &b	$p \to \{b\}, q \to \{a\}$	$p \to \{a,b\}, q \to \{a,b\}$

Figure 3.5: The difference between flow-sensitive and flow-insensitive pointer analysis.

We implemented both flow-sensitive and flow-insensitive pointer analysis as a part of the *dg* library. Our pointer analysis framework is based on the observation of differences between different pointer analyses. We extracted the common behavior of a few pointer analysis, namely the Andersen's analysis (and it various mutations) and the Steensgaard's analysis, and made an unified framework where different pointer analyses share most of their code, and differ only in few functions that are called from the shared code. To put clear our motivation behind the framework, we must explain (although without unnecessary details) how these pointer analyses work.

Andersen's pointer analysis

Andersen's pointer analysis [3] is a flow-insensitive analysis based on computing data-flow equations. In the first step, it gathers pointer-relevant statements from the program and expresses them in terms

5. A common practice is to handle structures field-sensitively and arrays field-insensitively, as a single object.

of set inclusions (for example, to say that p may point to a, we write $\{a\} \subset p$, meaning that the memory location a is in the set of memory location that p can point to. For the assignment $p = q$ we have $q \subset p$, etc.). Then we solve this system of equations.

More convenient way to look at this analysis is to express it as a graph problem rather then a set of equations [41]. The points-to information can be captured as a *storage shape graph (SSG)* [13]. To every memory location that can be referenced, we assign a node and an edge between nodes A and B means that A may point to B. An example SSG for the program from Figure 3.5 can be found in Figure 3.6.

$$\begin{tikzcd}
\boxed{\mathrm{p}} \arrow[r] \arrow[dr] & \boxed{\mathrm{a}} \\
\boxed{\mathrm{q}} \arrow[r] \arrow[ur] & \boxed{\mathrm{b}}
\end{tikzcd}$$

Figure 3.6: A SSG for the program from Figure 3.5 computed by the Andersen's analysis.

To describe the Andersen's analysis using graphs, we can define operations on an SSG in the following way [37] (we use C syntax for reference and dereference of pointers):

p = &a	$p \rightarrow a$
q = p	$\forall x$ if $p \rightarrow x$ then $q \rightarrow x$
q = *p	$\forall x$ if $p \rightarrow x$ then ($\forall y$ if $x \rightarrow y$ then $q \rightarrow y$)

Read the rules as: when we find `p = &a` in a code and p is a pointer, then add an edge from the node p to the node a. Similarly, when we find `q = p` in a code and q, p are pointers, then add an edge from q to all nodes that p points to. Note that we have only single dereference in our rules. Multiple dereference would be created in the same manner (looking at successors of successors). However, each program with multiple dereferences can be transformed to an equivalent program with single dereferences by introducing temporary variables (which is exactly what LLVM does), therefore it is sufficient to give the rule for one dereference only. The same decomposition can be done for dereference of the left side (`*q = &a`), so we do not need that one too. We refer to these rules for adding edges simply as the rules.

The analysis proceeds by iterating over the relevant statements in the program and by adding edges to the SSG according to the rules until it reaches a fixpoint (until no new edge can be added).

This analysis is flow-insensitive by design (we gather the relevant statements, ignoring the flow of control), however, we can make an "Andersen's style" flow-sensitive analysis. Adhering to the graph view, we can assign a new SSG to every location in the program. This graph then captures the points-to information that is valid at the point in the program. We have to, of course, change the way the analysis proceeds, since now the order of instructions is important. We use a data-flow analysis [21] – we compute the pointer information for a statement (i.e. add edges to the graph belonging to the statement) from the statement itself and the information from predecessors and then propagate the information to all successors of the statement in the CFG. Successors will (possibly) modify the information (the graph) and pass the new state to their successors and so on until the stable state is reached. This approach is different from the flow-insensitive approach, but it still has something in common: it uses the same rules for adding edges, only now on local graphs connected to statements in the program.

Steensgaard's pointer analysis

Steensgaard's pointer analysis [44], when viewed as a graph problem[6], works the same as the flow-insensitive Andersen's analysis, but with an additional restriction on the underlying graph [41]. The restriction is that every node can have at most one outgoing edge. Every time two edges go from a node p to two different nodes a, b, we merge the nodes a and b, as is shown in Figure 3.7. Merging such nodes results to an SSG that has outgoing degree of every node at most one and its size is linear in the number of nodes. This analysis is very fast, but imprecise. The only difference from the Andersen's analysis is that

6. The Steensgaard's analysis is originally defined by the means of types and their unification – a type represents a set of memory locations a pointer may point to. We assign types to all pointers and unify them according to some rules until the program is well-typed. The result is a type assignment that corresponds to an over-approximation of points-to sets for each pointer [44].

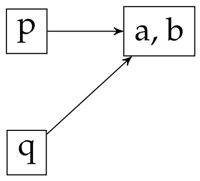


Figure 3.7: A SSG for the program from Figure 3.5 computed by the Steensaard's analysis. Nodes a *and* b *needed to be merged, because there were edges from* p *(and also from* q*) to both* a *and* b*.*

we perform merging of nodes in certain cases[7], but the rules used for adding edges are preserved.

Unified pointer analysis framework

We based our pointer analysis framework on the fact that different pointer analyses that can be defined as a graph algorithm share the rules for adding edges as we described them for the Andersen's analysis. If we always use the same rules, where is hidden the difference in above-mentioned analyses? Every operation can be broken down into two parts: choose relevant nodes from an SSG and add new edges according to the information in these nodes. If we choose different nodes, we get different behavior of the analysis. If we add different edges (or modify the graph as a part of adding an edge) we get different behavior. Imagine that an SSG have always enough nodes for a variable that we can use during the analysis, i.e. for every variable p, we can have nodes p_1, p_2, p_3, etc. If we always use a node p_1 in the rules for a variable p, the analysis will be flow-insensitive. If we properly add edges and choose the right nodes between more "versions" of the variable p, we can get flow-sensitive analysis. In Figure 3.8, we have the same SSG for the program from Figure 3.5 but with different analyses run on it. When we run flow-insensitive analysis,

7. Actually, another difference is that the Steensgaard's analysis uses auxiliary structures to be able to reach a fixpoint after only one scan of a program [44]. This is not important for us now, since we focus only on the operations on the graph (although the one-pass algorithm could be implemented in our framework too). Without the auxiliary structures, the fixpoint would be reached anyway – it may just take more than one iteration.

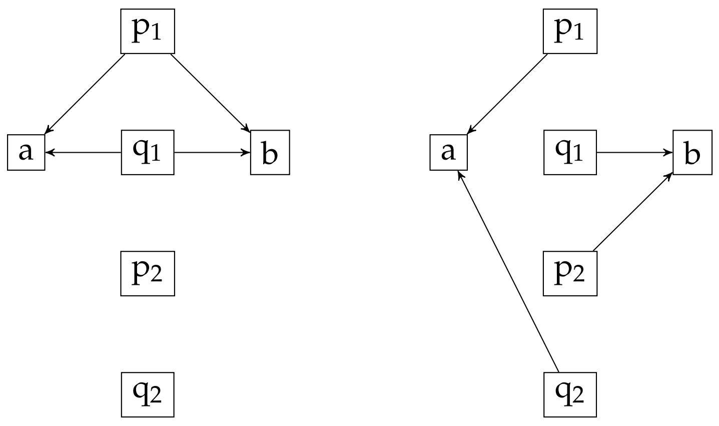


Figure 3.8: A difference between storage shape graphs for the program from Figure 3.5 as computed by our flow-insensitive (left) and flow-sensitive (right) pointer analysis

the rules always use the p_1 node for the pointer p and the q_1 node for the pointer q, thus making the analysis flow-insensitive. In contrary, the flow-sensitive analysis use different "versions" of p and q in the rules [8]. This is how the rules look like when applied:

Statements	Flow-insensitive	Flow-sensitive
`p = &a`	$p_1 \rightarrow a$	$p_1 \rightarrow a$
`q = &b`	$q_1 \rightarrow b$	$q_1 \rightarrow b$
`q = p`	$\forall x.p_1 \rightarrow x : q_1 \rightarrow x$	$\forall x.p_1 \rightarrow x : q_2 \rightarrow x$
`p = &b`	$p_1 \rightarrow b$	$p_2 \rightarrow b$

The only difference between flow-insensitive and flow-sensitive analysis is in the selection of nodes we provide to the rules. Moreover, if we would merge nodes while adding edges, we would get Steensgaard-like analysis. This view allows to implement different analyses by defining few simple functions. These functions do all the necessary work transparently on the background, allowing the analysis abstract from the real implementation. The analysis only calls functions to get

8. Variables a and b do not have different versions, because they are not pointers and therefore it is safe to have only one copy of them.

nodes and to add edges, and the implementations of these functions take care of returning the nodes that are relevant for the statement that is being processed and for adding new edges to the right nodes (and possibly modify the underlying graph).

As the first step in our pointer analysis, we build a *pointer subgraph (PS)* for the analyzed LLVM bitcode (Figure 3.9). The pointer analysis runs only on this subgraph. A PS is built from a CFG by removing all nodes that are not relevant to the pointer analysis[9]. The instructions that need to be in a PS are namely: pointer assignments and loading pointers to top-level variables, getting pointers to elements of structures and arrays, pointers casting, phi instructions of pointer type, and also every memory allocation. Further, in a PS must be all global variables (as in LLVM every global is a pointer to static memory) and return instructions that may return a pointer or that are important for the flow of control in the PS (as is the `ret 0` node in the PS in Figure 3.9). We also use no-operation nodes which only simplify a generation of a PS and are not important for the analysis. Since LLVM is typed, we can identify these nodes syntactically with the only exception: pointers casted to an integer and back. In the case that a pointer is casted to an integer, we track and transitively add all the uses of the integer into the PS.

We handle interprocedural analysis by inlining called procedures into PS (parameter passing is modeled by creating a *phi* node for each pointer argument; the phi node gathers the pointers passed to this argument, see Figure 3.9). If a procedure is called via a function pointer, we dynamically re-build the PS during the analysis.

All dynamic memory allocations are represented by (and thus summarized to) the call site of the function that allocates the memory (e.g. calls to `malloc`, `calloc` or `realloc` functions).

Our PS is generic – it is independent from LLVM and could be generated from any other program representation (like C or assembler), therefore our pointer analysis can be used for any language (as long as we can generate a PS for it).

9. Lyle and Binkley use a *pointer state subgraph*, which is also a subgraph of a CFG, but captures only assignments to pointers [10]. Hind et al. use a sparse evaluation graph [23], which, in this case, is that same as our pointer subgraph. However, a sparse evaluation graph is a more generic notion [16].

```
define void @init(i32* %arr, i32 %i) {
  %1 = getelementptr inbounds i32* %arr, i32 %i
  store i32 %i, i32* %1
  ret void
}

define i32 @main() {
  %i = alloca i32
  %array = alloca [10 x i32]
  store i32 0, i32* %i
  br label %1

; <label>:1
  %2 = load i32* %i
  %3 = icmp slt i32 %2, 10
  br i1 %3, label %4, label %9

; <label>:4
  %5 = getelementptr [10 x i32]* %array,
        i32 0, i32 0
  %6 = load i32* %i
  call void @init(i32* %5, i32 %6)
  br label %7

; <label>:7
  %8 = add nsw i32 %6, 1
  store i32 %8, i32* %i
  br label %1

; <label>:9
  ret i32 0
}
```

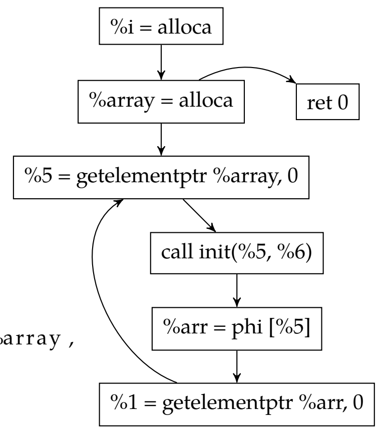


Figure 3.9: A LLVM bitcode (allocation of an array with 10 integers and initializing each integer to its index using a function), and the corresponding PS (the names of nodes are truncated).

To address the field-sensitivity, we represent a pointer with a pair *(n, off)*, where *n* is the node from a PS where the memory was allocated (a node associated with an `alloca` instruction or with a call to a function that allocates dynamic memory), and *off* is the offset into the memory represented by a 64-bit number [10]. Using offsets change an SSG: every node of an SSG is split into parts that correspond to different offsets in the memory (see Figure 3.10). To keep the points-to

10. In the following text, we assume that the width of a pointer is 8 bytes and the width of an integer is 4 bytes.

```
%struct.A = type {i32 *, i32 *}

%a = alloca i32
%c = alloca i32
%r = alloca i32*
%s = alloca %struct.A
store i32* %a, i32** %r
%1 = load i32*, i32** %r
%2 = getelementptr %struct.A ,
      %struct.A* %s, i32 0, i32 0
store i32* %1, i32** %2
%3 = getelementptr %struct.A ,
      %struct.A* %s, i32 0, i32 1
store i32* %c, i32** %3
```

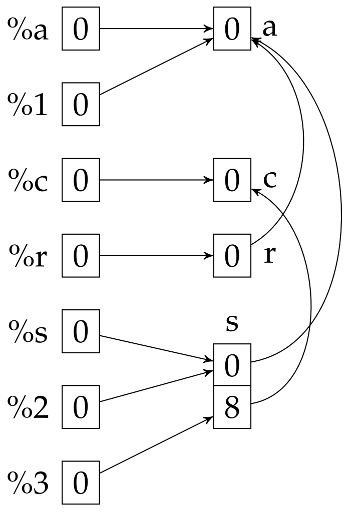


Figure 3.10: A simple program with pointers and the corresponding SSG extended with offsets. Top-level variables point to memory and never have more that one part (they can have only an offset 0) nor any incoming edge.

information for each offset in an SSG node, we use a sparse map rather than an explicit enumeration of all possible offsets, since the range of possible offset values in a node can be huge.

When we perform a pointer analysis on a C code, every pointer is also a variable, thence it can be referenced and there can be an edge in the SSG between any two nodes. This is not true in LLVM, where top-level variables always have only one part (offset 0), and they have no incoming edges, because they are not a piece of memory and we can not point to them.

Ergo, top-level variables do not fit much into storage shape graphs, since they are "only" a value of the memory or a computation at some point in the program. The same top level variables that are in the SSG are also in the PS of a program. In our framework, we do not create nodes for top-level variables in the SSG. Instead, we use the corresponding nodes of the PS (so PS nodes can have a points-to set), and to keep a track of pointers in memory, we build an SSG aside. The analysis works with PS nodes and performs queries and modification to the SSG when needed (e.g. loading or storing a pointer from/to memory). The results are stored in PS nodes that correspond to pointers in LLVM. Such mixed graph is depicted in Figure 3.11.

The given semantics of a pointer – that it points to the PS node where the memory was allocated (the allocation site) – has two consequences. First, all edges from the SSG go to PS nodes. Second, every allocation site has a self-loop in the PS – it points to the allocation site where the memory was allocated.

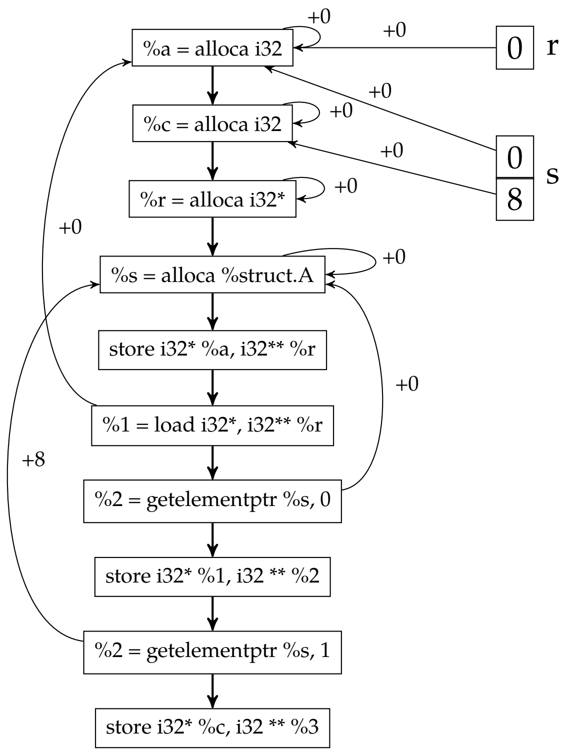


Figure 3.11: The pointer subgraph for program from figure 3.10 and the SSG extended with offsets are combined together in our analysis. Thick edges are control flow edges and the thin edges are points-to relation (an edge from A to B with label +c means that A points-to to the memory allocated at B with the offset c).

This two-graph structure perfectly fits into the approach we described at the beginning of this section: the pointer analysis is processing PS nodes and according to the operation associated with a node selects relevant nodes of the SSG that bear the actual points-to information and apply the rules for adding edges. Our pointer analysis framework use a virtual method `getMemoryObjects` that is overridden by the actual analysis. This method returns the nodes (we call them memory objects from the obvious reason) from the SSG that bear the points-to information relevant to the PS node that is being processed. The rules for adding edges are then implemented using this method. As an example, we give a simplified pseudo-code for loading a pointer from memory in Algorithm 3.

Algorithm 3 Loading a pointer from memory.

```
processLoad(stmt: n = LOAD(p)):

for all (target, offset) $\in$ pt-set(p) do
    objects $\leftarrow$ getMemoryObjects(stmt, target)
    for all o $\in$ objects do
        for all x $\in$ pt-set(o[offset]) do
            n->addEdgeTo(x)
```

Lets run this pseudo-code on the example from Figure 3.11. There is only one load: `%1 = load i32*, i32** %r`. Following the pseudo-code, we take every pointer where `%r` points-to. `%r` points only to itself (to memory allocated at `%r` with offset 0). Then we get nodes from the SSG (the memory objects) that keep the relevant points-to information at point in the program *stmt* (the flow-insensitive analysis ignores the *stmt* and always returns the same node for `%r`). We get one object that on offset 0 points-to `%a` with offset 0 – we copy that pointer to `n` and we are done.

The pseudo-code is simplified. Apart from what is shown, we take a care of a special value of an offset there: *UNKNOWN_OFFSET*. We use *UNKNOWN_OFFSET* when we know that a pointer points to some memory, but we do not know exactly where (this arises for example when accessing an array via a variable index that is read from outside). When we load a pointer from memory (an SSG node) that has a record for *UNKNOWN_OFFSET*, we must copy even all the pointers in that record. When we have a load with a pointer that points to some memory with *UNKNOWN_OFFSET* we must copy all pointers from that memory, because any of them is covered in the *UNKNOWN_OFFSET*. We also use the *UNKNOWN_OFFSET* to keep points-to sets smaller. When we have a lot of pointers to a memory with concrete offsets, we can replace them with the *UNKNOWN_OFFSET* which is a sound approximation.

To correctly handle nested structures, we must perform the addition of offsets. Consider this code snippet:

```
%1 = getelementptr i8* %s , 0
%2 = getelementptr i8* %1, 16
%3 = getelementptr i8* %2, 8
```

In this code, the top-level variable %1 points to $(s, 0)$, the top-level variable %2 points to the same memory as %1, but 16 bytes further and the top-level variable %3 points to the same memory as %2 but 8 bytes further. That is:

$$
\begin{aligned}
\%1 &\rightarrow (s, 0) \\
\%2 &\rightarrow (s, 16) \\
\%3 &\rightarrow (s, 24)
\end{aligned}
$$

From the example, we can see that a sufficient implementation of offsets addition is: "take every pointer (p, o) where can the node point to and add the offset to o", and we indeed use it like this. However, adding offsets this way has a drawback. In the flow-insensitive analysis, we represent each pointer by one node for whole program, thus all points-to information for the pointer is accumulated in this node. Consider this LLVM code:

```
1. %array = alloca [10 x i32]
2. %p = alloca i32*
3. store %array , %p;
4. %1 = load %p
5. %2 = getelementptr i32* %p, 1;
6. store %2, %p;
```

After the first iteration, we get that $p \rightarrow \{(\text{array}, 0), (\text{array}, 4)\}$ and that something changed, so we did not reach fixpoint and go for another iteration. In the next iteration, lines 5 and 6 make the analysis take all pointers where p points to and add 4 to the offsets (shift by 1 element in the array), thus we get that $p \to \{(\text{array}, 0), (\text{array}, 4), (\text{array}, 8)\}$ and in the another iteration the analysis adds the offset 12, and then 16, etc. This way we add all the offsets $4k\ (mod\ 2^{64})$, for k a natural number. With such many pointers in a points-to set (and such many iterations), the pointer analysis can not perform well. To prevent this, we keep track of the size of allocated memory (when possible), and once we add a pointer with the offset outside of the memory to a points-to set S, we replace all pointers to that memory with a concrete offset in S with the pointer to that memory with the *UNKNOWN_OFFSET*. Adding a pointer to a memory with concrete offset when we already have a pointer to the same memory with the *UNKNOWN_OFFSET* has no effect (we do not add such pointers to points-to sets), this solves our problem. The same situation can occur in flow-sensitivie analysis, but only when such pointer assignment (like on the line 6) is inside a loop. Note that we can forbid using concrete offsets and by using *UNKNOWN_OFFSET* everywhere, we can get field-insensitive analysis.

In the flow-sensitive case, the pointer analysis performs classical data-flow analysis in the background, therefore we do not gain on efficiency with our approach. What is the contribution of this framework is that it distills the common behavior of pointer analyses and allows us to define different pointer analyses by defining only a few operations. Moreover, the framework is highly configurable. Apart from the possibility of using different hybrids of pointers analyses (the implementation of `getMemoryObjects` defines whether the analysis is flow-sensitive or insensitive, under-approximation or over-approximation or any combination of these, e.g. flow-sensitive only in some parts of a code and flow-insensitive everywhere else, etc.) we can tune field-sensitivity by giving restrictions on possible offsets. Other aspects of program analyses, like sparesness, could be covered with our approach too. However, we do not support them yet, therefore we omit the description of these.

Except for `Store` instruction, calls to functions like `memcpy` can manipulate with pointers too. We have a direct support for the `memcpy` in our framework. However, it could be modeled as a sequence of

`Load`, `GetElementPtr` and `Store` instructions, and that is what we do for other similar instructions (e.g. for the `InsertElement`).

Our analysis has a support for an *unknown* pointer. The *unknown* pointer represents an unknown memory location. It is used, for example, when a call to undefined function returns a pointer.

Except for the unified pointer analysis framework we just described, we still keep a support for an old pointer analysis we used initially in the slicer. It is an implementation of the flow-insensitive Andersen's analysis extended with offsets in the same manner as described in this section. There are two differences, though. First, the old analysis runs on nodes of a dependence graph (it iterates over the nodes of the dependence graph and stores points-to sets also into the nodes), and second, it has worse support for some features like casting pointer to integer and back (in this case, the old pointer analysis treat the pointer as unknown), or `mempcy` instruction.

3.2.2 Reaching definitions and Def-Use analysis

Once we have the points-to information for a program, we know what parts of memory can be read or modified by which statement. Simply adding an edge from every node that writes to a memory location m to every node that reads values from the memory location m is too liberal (but sound) approach. What we really want is to add an edge between reads from a memory location and reaching definitions of that memory location, as described in Section 2.2. In programs without pointers, every redefinition of a variable is a *strong* update – we forget (kill) the old definition, since it is definitely overwritten by the new one. In the presence of pointers, a redefinition may not be a strong update. If we know that on a statement we define a memory location l_1 or a memory location l_2 we can not kill the old definition, since we do not know which of l_1 or l_2 is really defined during a runtime. We must perform a so-called *weak* update – we only add the definition to the old one without killing the old one. We may perform a strong update only in the case that the pointer information we have is the *must* information.

A write to an unknown memory (a write using the *unknown* pointer), is taken as a definition of any variable and is always a weak update.

The first step of our reaching definitions analysis is to build a subgraph of a CFG (similarly as during the pointer analysis), that contains only instructions that write to memory (store instructions), call instructions, return instruction (the ones that are important for the flow of control in the subgraph) and every memory allocation. Call instructions are important from two reasons: first, when we know the definition of the called function, we must include the nodes from the function into the subgraph (like in the pointer analysis, we inline the called functions), second, when the function is undefined, we must assume that memory pointed to by pointers passed into the function via parameters may be defined there. The only exception is when the undefined function is `malloc` (or alike) or LLVM's intrinsic function like `memcpy` or `memmove`. Although these functions are undefined, we know their semantics and we can handle them more mercifully. Memory allocations serve as the operands for other nodes. Moreover, we create a no-operation nodes that serve for easier generation of the subgraph, or as join nodes where needed. See Figure 3.12 for an example of a reaching definitions subgraph.

Since our pointer analysis works with offsets, the reaching definitions analysis must use offsets too. To describe a definition, we use a triple *(n, off, num)*, where *n* is the node where was the memory that is defined allocated, *off* is the offset into the memory and *num* is the number of bytes that are overwritten by the definition. *UNKNOWN_OFFSET* is a valid value for both the offset and the number of bytes in the triple. When we have a definition d_1, we can overwrite it with a definition d_2 (strong update) only when d_2 overwrites the whole memory that is defined by d_1 (and, of course, d_2 must be a definite definition, not only a possible definition).

Two sets of definition triples are associated to every node of the subgraph. One set is the set of definitions produced by the node (the definitions set) and the other is a set of definitions killed at the node (the kill set). For strong updates, these sets are equal. For weak updates, the kill set is empty. We use even the setup when the definitions set is (possibly) empty and the kill set is not – we kill definitions of local variables that may not be propagated to callers (their address

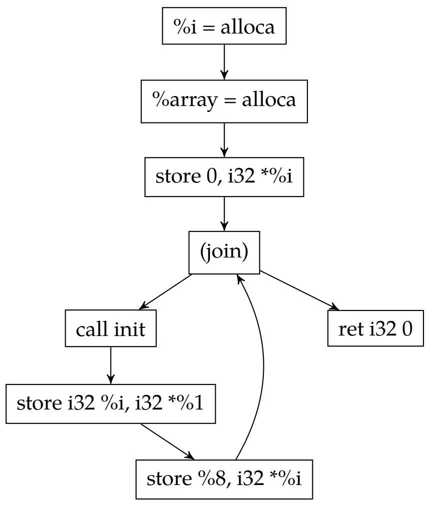


Figure 3.12: A reaching definitions subgraph for the LLVM bitcode from Figure 3.9 (the names of nodes are truncated).

is never assigned to a pointer) at return points from procedures. The reaching definitions analysis then proceed as usual data-flow analysis: for every node, it gathers reaching definitions from predecessors, adds reaching definitions generated by the node to them and remove all the definitions that are killed on the node [36]. This is repeated until a fixpoint is reached. An output of our reaching definitions analysis on the reaching definitions subgraph from Figure 3.12 is depicted in Figure 3.13.

Def-use analysis is the final step in building a dependence graph. It copies all def-use chains of top-level variables from LLVM to a dependence graph and also connects nodes representing a possible use of a memory location with nodes representing (possible) reaching definitions of that memory location. Since we have not implemented the two-pass algorithm of Horwitz et al. yet, we have easy work with parameters. For the top-level variables passed to a procedure, we create parameters and connect them with the variables using dependence edge (the parameter is a new use of the top-level variable) and we also interconnect the actual and formal parameters using parameter edges as described in Section 2.2. However, for the indirect definitions via

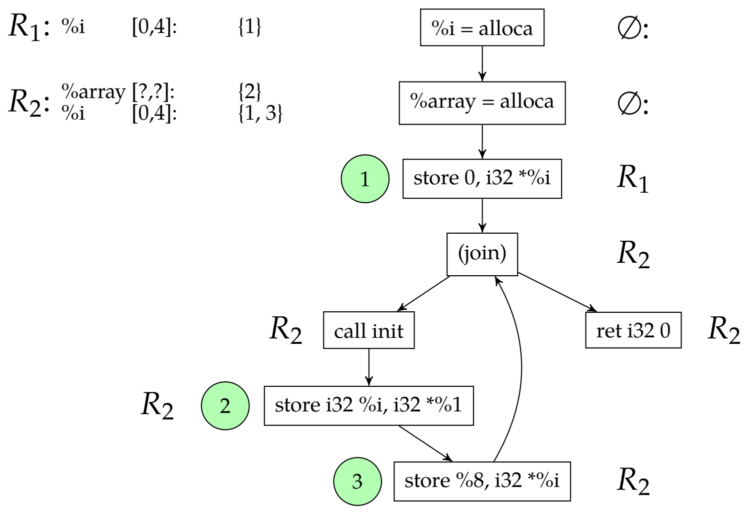


Figure 3.13: A reaching definitions computed for the program from Figure 3.9. The analysis computed one of $\emptyset$, R_1 or R_2 sets of reaching definitions for each node. For example, %i [0,4]: {1, 3} *means that the stores (1) and (3) write 4 bytes to memory %i on offset 0. The* UNKNOWN_OFFSET *is abbreviated with '?'.*

pointers we do not add any new parameteres. Instead, we add interprocedural dependence edges that do not go through parameters, but directly between procedures (the naive interpocedural dependence graph from Section 2.7.) An example of a whole dependence graph can be found in Figure 3.14.

3.3 Slicing a Dependence Graph

Slicing a dependence graph proceeds in two steps. In the first step, we mark the nodes that form the slice. As we have not implemented the two-pass algorithm yet, we mark the nodes using only the backward reachability. In the second step, we remove basic blocks and instructions from dependence graph as well as from the LLVM module.

While slicing, we first remove the basic blocks that do not contain any instruction from the slice, and already then we remove the rest of instructions. Removing a block includes these steps:

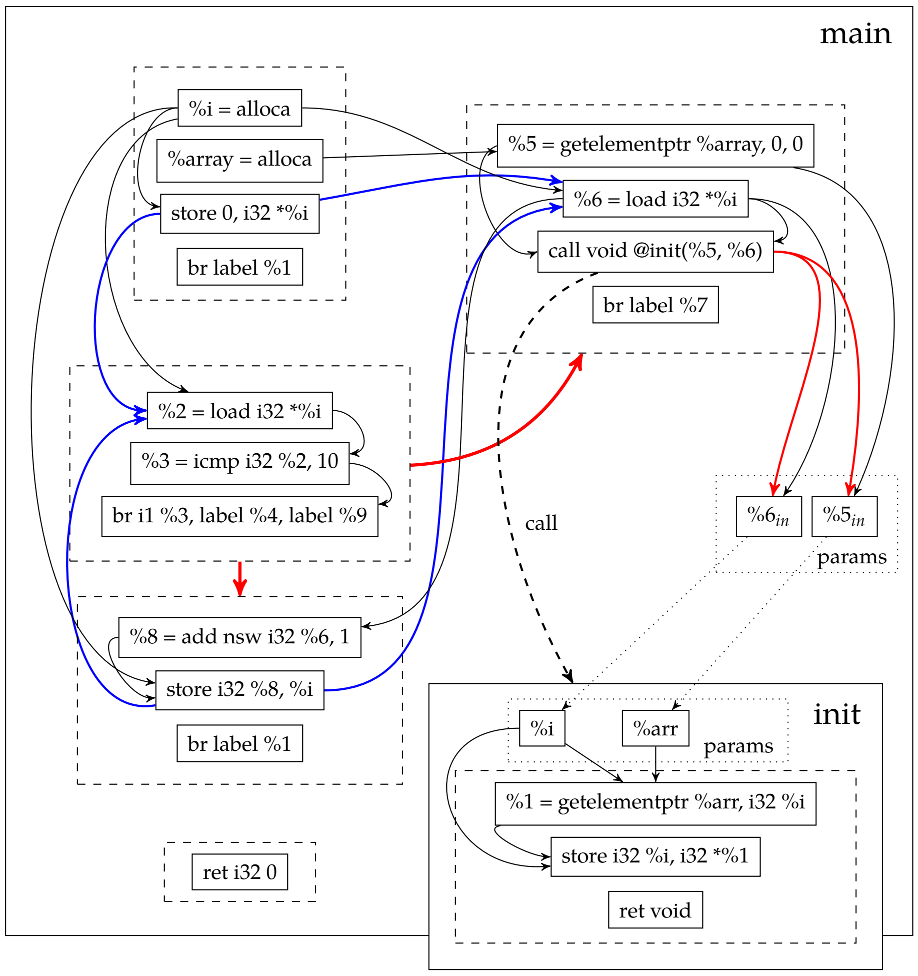


Figure 3.14: A dependence graph for the LLVM bitcode from Figure 3.9 (the names of nodes are truncated). Black solid edges are top-level data dependencies. Blue edges are indirect (memory) data dependencies, red edges are control dependencies. The dashed edge is a call edge of the function `init` *and dotted edges are parameter edges.*

- Remove corresponding LLVM block from a module
- Remove corresponding LLVM block from *phi* instructions that has a reference to it
- Cut the block away from a dependence graph

The first two steps are straightforward. Cutting the block away from a dependence graph is more complicated. We must reconnect the edges that go from/to the block that is being removed so that the labels on edges are correct: we always keep the labels that go to the block that is being removed, and forget the label that go from the block that is being removed [5]. Removing a block may temporarily result in another block to have more that two successors. After removing all blocks, though, every block has at most two successors. See Figure 3.15.

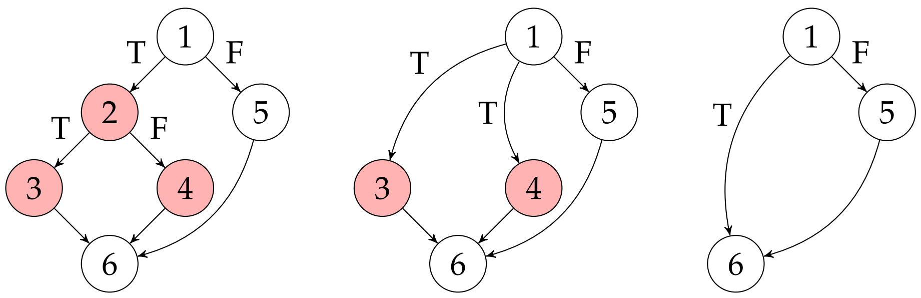


Figure 3.15: Removing basic blocks from a dependence graph requires correct reconnecting of labeled edges.

After we reconnected all edges between basic blocks in a dependence graph, we do a post-processing of the blocks, so that when we mirror the sliced dependence graph to LLVM, we get a valid bitcode. For example, we transform basic blocks which have two successors, but both are the same basic block, to a block with just one successor (thus we make it an unconditional jump). Then we reconnect the blocks in LLVM (which now contain dangling references to deleted blocks) according to the block in the dependence graph. Finally, we create a new entry basic block in the functions where the original entry block was sliced away.

The rest of slicing is trivial – we remove all the nodes from dependence graph and the corresponding instructions from LLVM that are not in the slice.

Testing

We have over a hundred of tests for slicing. Every test is a short program that calls an `assert` when executed. The program is closed (it does not take any inputs) and there is only one execution path through the program. Each test program is sliced and after slicing we check that the `assert` was called and that it passed. When we need test slicing of a program with calls of undefined functions, we link these functions after slicing.

Unsupported features

Our slicer do not implement the extensions for correct slicing of non-terminating loops (therefore it may make an unreachable code reachable), nor the extensions for interprocedural control flow (exceptions, calling of `exit` and alike from procedures). We also can slice only sequential programs. It supports most of LLVM IR. The exceptions are vector instructions and we currently have problems with some types of arithmetic operations on pointers converted to integers (e.g. multiplication by a constant) in which case we conservatively use the *UNKNOWN_OFFSET* or (in the worst case) the *unknown* pointer.

For a comparsion, the old slicer incorrectly slices some cycles because of an insufficient information in the pointer analysis. It does not support concurrent programs, exceptions and correct slicing of non-terminating loops as well. It also fails some of our tests for pointers casted to integers and `memcpy` handling.

4 Experiments

In this chapter, we compare the impact of the new slicer on results of Symbiotic and we also compare the speed and efficiency of the new and the old slicer.

We did a set of experiments on benchmarks from the SV-COMP competition [1]. SV-COMP is an international competition of software verifiers, where verification tools compete in finding bugs in computer programs (benchmarks) that are divided into categories according to the type of faults they may contain (e.g. a signed integer overflow or memory leaks). We used C benchmarks from reachability categories (these benchmarks are C source codes where the aim is to verify whether a call of a `__VERIFIER_error` function, that represents an error, is reachable). Machines with 8 64-bit cores, 1600 MHz each, and 8GB of RAM were used to take measurements. The experiments were run in parallel on 9 such machines, 4 experiments at a time, each on its own core.

First, we ran Symbiotic (git commit `d2e634e`) with the old slicer and with the new slicer on these benchmarks. The new slicer was used in three different setups: with the flow-insensitive analysis (further in the text, we refer to it as *FI*), with the flow-sensitive analysis (*FS*), and with the old flow-insensitive analysis (*OLD*). Possible results of a run of Symbiotic on a benchmark are: *true*, *false*, *unknown*, *timeout* or *error*. A result is an answer to the question: "Is the benchmark without bugs?". Therefore the answer *true* means that Symbiotic did not find a bug, *false* means that a bug was found, and *unknown* means that Symbiotic does not know what to answer (Symbiotic did not find a bug, but is not sure that there is none). Symbiotic returns the *error* result whenever anything during verification went wrong. The *timeout* result is self-explanatory. Timeout for Symbiotic was set to 300 s.

With the new slicer (with any setup), we get much less incorrect answers and much more correct answers than with the old slicer. The new slicer with the *OLD* analysis has the most correct *true* answers, while it gives similar number of correct *false* answers in all three setups. It is because the *FI* and *FS* pointer analyses abort when they run into an unhandled instruction. The *OLD* analysis, however, returns an

1. `http://sv-comp.sosy-lab.org/2016/benchmarks.php`

unknown pointer if possible in such case, therefore slicing proceeds without an error. The difference between *true* results between the *OLD* and the *FI* (*FS*, respectively) analyses roughly corresponds to the difference between *error* results between these analyses. The most incorrect results is with the *FS* analysis. We currently do not know whether the incorrect answers are caused by a bug in the slicer or in some other component of Symbiotic (the verification without slicing timeouts for every timeout we tried). The measurements are depicted in Table 4.1.

slicer	true		false		unknown	timeout	error
	ok	nok	ok	nok			
old	653	89	255	36	86	2505	2568
new + FI	994	4	498	11	137	2667	1881
new + FS	999	13	518	12	139	2707	1804
new + OLD	1318	5	489	10	132	2824	1414

*Table 4.1: Results of Symbiotic with the old (*old*) slicer, the new slicer with the flow-insensitive analysis (*new + FI*), the new slicer with the flow-sensitive analysis (*new + FS*), and the new slicer with the old flow-insensitive analysis (*new + OLD*).* ok *means that the answer returned by Symbiotic was correct and* nok *means that it was incorrect. A timeout was set to 300 s.*

We ran the slicers also alone on the same set of benchmarks. In this set of experiments, every benchmark was compiled using the *clang* to a LLVM bitcode[2]. Afterward, a definition of the `__VERIFIER_error` function was linked to the bitcode (the definition only calls `assert(0)`). The slice was taken with respect to the call site of the `assert`, as in the case when the slicer runs in Symbiotic[3]. After linking, we once sliced the code as it was, and once we performed some conservative

2. In the previous set of experiments, the benchmarks were compiled automatically by Symbiotic, also by *clang*.
3. The old slicer use call sites of the `assert` as the slicing criterion, so it is easier to provide the definition of the `__VERIFIER_error` function than modifying the old slicer to slice with respect to `__VERIFIER_error` call sites. The new slicer can take the slicing criterion as an argument.

optimizations provided by LLVM (constant propagation, dead code elimination, promoting read-only arguments to uses without copying to a local variable, arithmetic instructions combination and a CFG simplification). The reason for these optimizations is that these optimizations are used also in Symbiotic before slicing, so we wanted to investigate the impact of the optimizations on slicing. Moreover, the optimizations can change the code in many ways and thus can reveal unhandled features in slicers.

In following tables, we use *success* to say that slicing proceed without errors (in the other case we use *error*), and *timeout* to say that slicing timeouted. Timeout was set to 120 s.

The optimizations strongly help the new slicer. This is because the optimizations may remove the instructions that are currently unsupported by the new slicer, therefore it has less *error* results. The opposite effect have the optimizations on the old slicer. It produce much more errors than without the optimizations. In a random sample of benchmarks, all errors of the old slicer were caused by an invalid bitcode produced by the old slicer (LLVM automatically checks the validity of a bitcode when writing it to a file and aborts on any error). The bitcode contained corrupted *phi* instructions that had dangling references to deleted values. The optimizations probably optimize the *phi* instructions to a shape that is not handled by the old slicer. The new slicer with the *OLD* pointer analysis has the best results. The reason is the same as the reason why this setup has the most correct *true* answers when used in Symbiotic: the *OLD* analysis does not abort on any unhandled instruction, but returns the *unknown* pointer where possible. This approach leads to much less *error* results and it gains on the speed of the analysis (on behalf of the precision). The measurements can be found in Table 4.2, and some more statistics are in Table 4.3.

We removed the *error* results from the measurements in Table 4.2, and from the rest we took some statistics. The new slicer with the *OLD* analysis is much faster (on average) than the old slicer and also than the new slicer with other analyses. We assume that it is because the other analyses in the new slicer perform a lot of demanding computations in the cases where the old slicer only say that the pointer is *unknown* (the *FI* and *FS* analyses support more instructions than the old analysis). It is only a guess and we need to perform a deeper analysis to be sure.

slicer	w/o optimizations			with optimizations		
	success	timeout	error	success	timeout	error
old	4349	733	1110	2981	670	2541
new + FI	3399	875	1918	4047	694	1451
new + FS	3508	839	1845	4181	648	1363
new + OLD	4159	206	1827	4861	188	1143

Table 4.2: Slicers ran alone on the benchmarks with timeout 120 s. w/o optimizations *means that the bitcode was not optimized by LLVM before slicing,* with optimizations *means that it was optimized.*

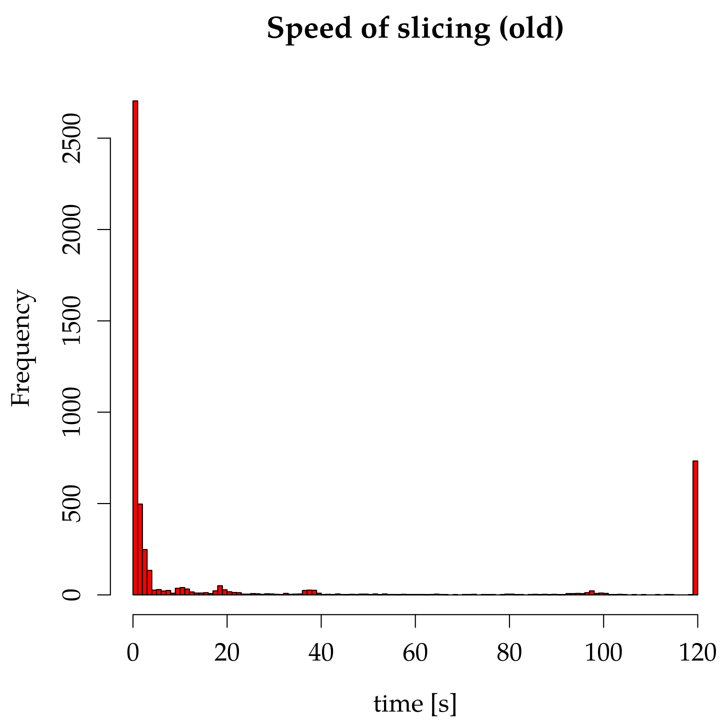
slicer	w/o optimizations		with optimizations	
	avg(time)	avg(size)	avg(time)	avg(size)
old	23.05 (6.71)	0.67	26.06 (4.95)	0.65
new + FI	28.66 (5.11)	0.65	21.59 (4.74)	0.63
new + FS	27.75 (5.66)	0.64	20.68 (5.26)	0.62
new + OLD	7.46 (1.88)	0.63	6.27 (1.89)	0.64

Table 4.3: Statistics of slicing of the benchmarks with a timeout 120 s, excluding error *results. In parenthesis is the average slicing time excluding* timeout *results.* w/o optimizations *means that the bitcode was not optimized by LLVM before slicing,* with optimizations *means that it was optimized.*

From Table 4.3 it may seem that slicing is quite slow. The opposite is true – slicing is very fast in most cases. The average times are influenced by a few benchmarks that take long to slice. To get better understanding of the numbers, see Figure 4.1 that shows a distribution of running times, and Figure 4.2 that shows a distribution of relative sizes of sliced bitcodes. In both cases, the higher are the columns on the left, the better.

In Figure 4.1 are displayed measurements of slicing of bitcodes without optimizations (with optimizations, they look similarly). Measurements for the new slicer and *FI* and *FS* analyses are omitted – those look very similarly to the measurement with the old slicer. In Figure 4.2 are distributions of relative sizes of a sliced bitcode without optimizations. In Figure 4.3 are measurements of the new slicer with *FS* and *FI* analyses on bitcodes with optimizations. We omitted the *old* slicer and *OLD* analysis measurements with optimizations, since those distributions do not change much.

Worth of a remark is that the sizes of sliced bitcodes may be greater than 100 %. It is because both the new and the old slicer include new labels and possibly entry or exit basic blocks into the code when reconnecting the successors of basic blocks. Since the sizes are displayed as percents, these additions may be non-negligible when the original code is tiny.



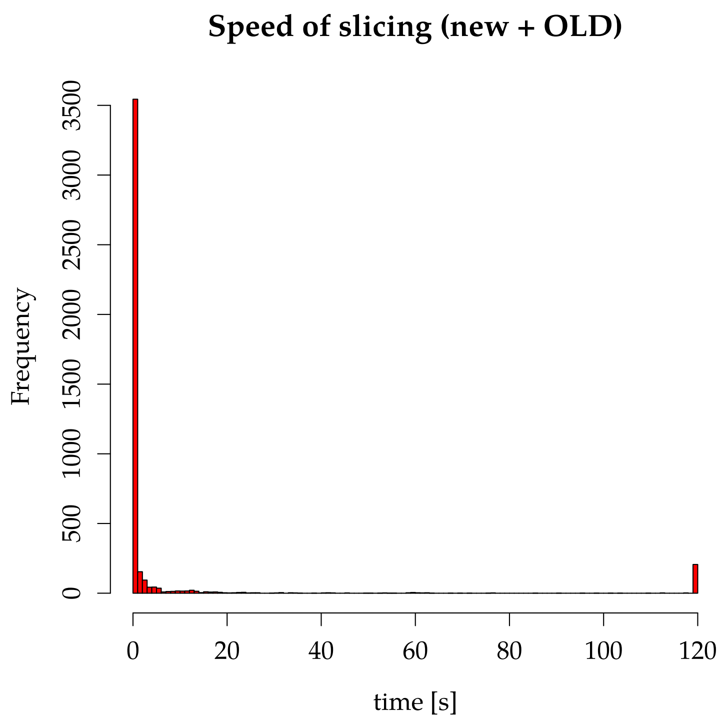
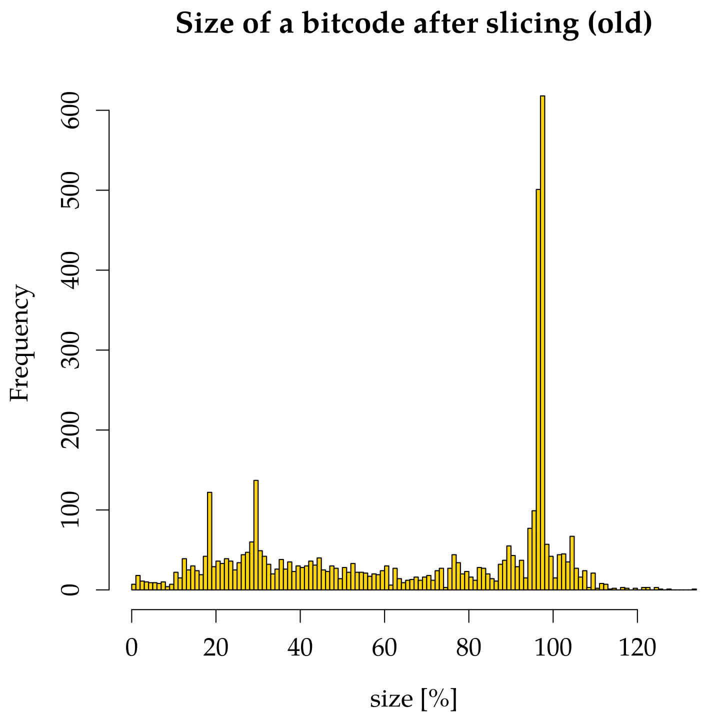
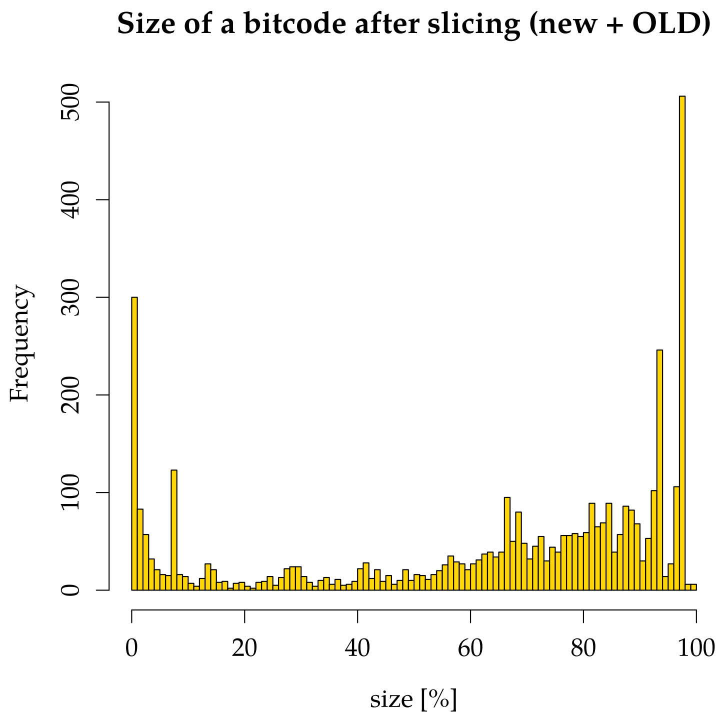
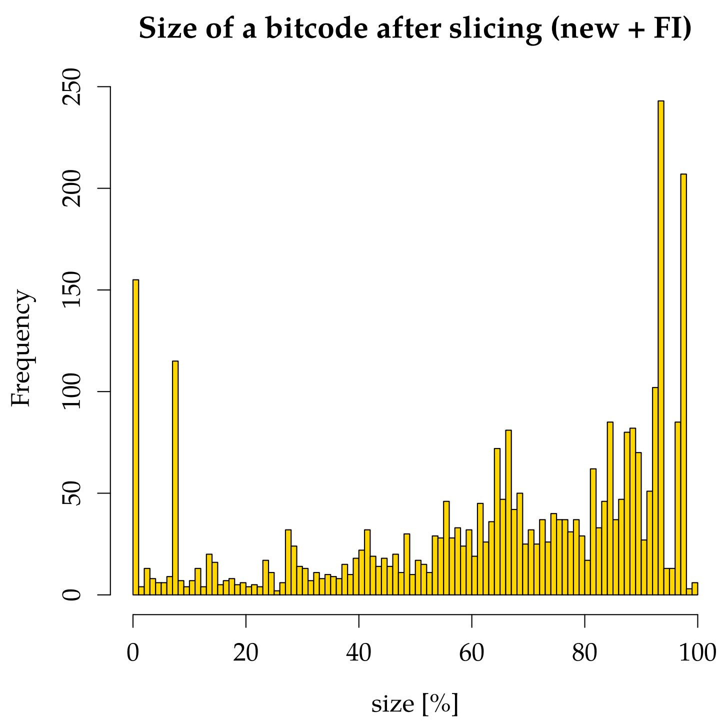


Figure 4.1: Distribution of times of slicing bitcodes (without optimizations).


Size of a bitcode after slicing (old)
Frequency
0
100
200
300
400
500
600
0
20
40
60
80
100
120
size [%]


Size of a bitcode after slicing (new + OLD)
Frequency
0
100
200
300
400
500
0
20
40
60
80
100
size [%]



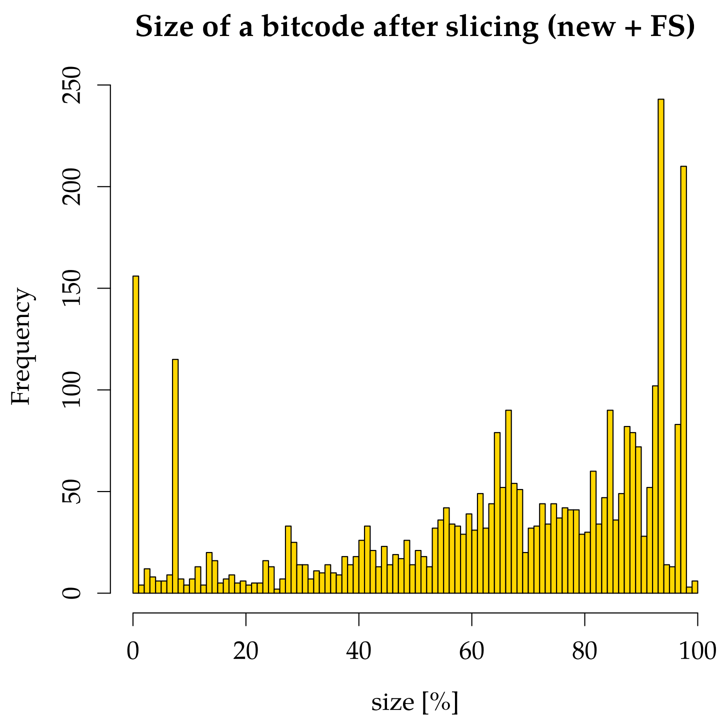
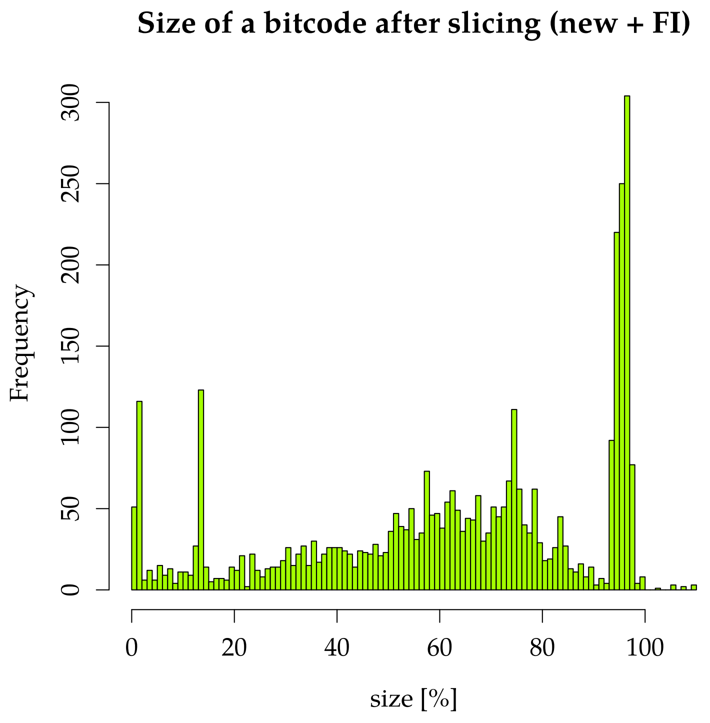


Figure 4.2: Relative sizes of sliced bitcodes (without optimizations).



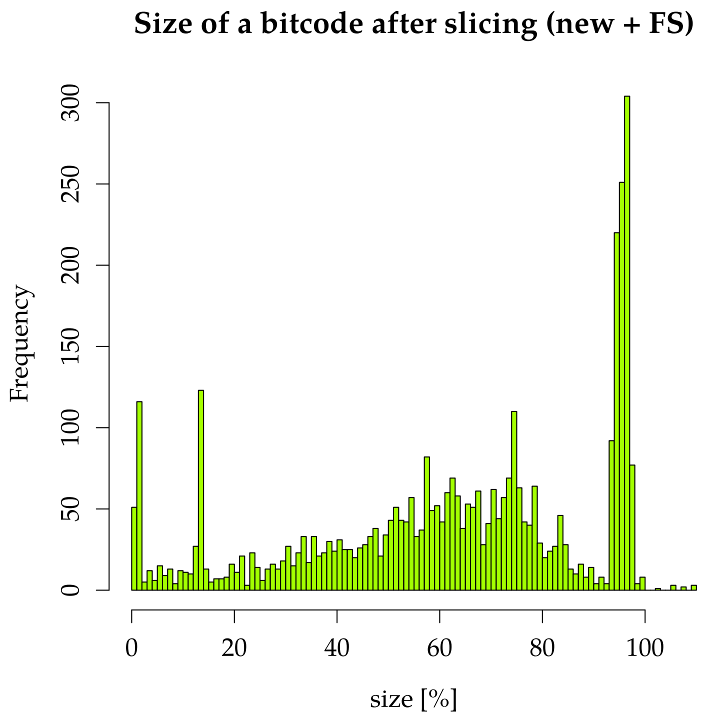


Figure 4.3: Relative sizes of sliced bitcodes (with optimizations).

5 Conclusion

In this thesis we described the Weiser's algorithm for slicing programs, and the slicing algorithms based on dependence graphs. Then we looked into extensions of the dependence-graph based slicing algorithms to support modern progamming languages features like pointers or unstructured control flow. We created a library *dg* that implements the dependence-graph based slicing algorithm for LLVM and analyses important for building a dependence graph. The new slicer was integrated into the tool Symbiotic that use program slicing to speed-up program verification instead of an old slicer that use the Weiser's algorithm. Evaluation of an impact that the new slicer has on Symbiotic's performance was given.

6 Further development

The new slicer still lacks a support for some important features, like the two-pass algorithm or slicing programs with exceptions or threads. In this chapter, we describe some improvements we would like to focus on in the future.

New Control-Dependence Algorithm

A classical definition of a control dependence do not work on control flow graphs that do no have an *EXIT* node or that contain blind ends. We created a new algorithm for determining control dependencies in a directed graph. In this section, we sketch how the algorithm works.

Since the classical definition of a control dependence use the *EXIT* node, we need to use a more generic notion of a control dependence.

Definition 10. *We say that a node* m *is in control scope of a branching node (a node that has more than one successor)* n *iff* m *is reachable from* n *and there exists a path starting in* n *that does not contain* m*.*

By a path we mean a path in a graph – that is without repetition of nodes. If we would allow the nodes repetition, the control scope relation would have slightly different properties (that may be useful too). We define control dependence as a cover of the control scope relation:

Definition 11. *A node* m *is control dependent on a branching node* n*, iff* m *is in control scope of* n *and there is no branching node* d *in the control scope of* n *that contains* m *in its control scope.*

We can see that our definition is less restrictive than the classical one – we consider any path from a branching node, whereas the classical definition use only paths that go to *EXIT* node. It is easy to show that a set of statements from a slice computed using the classical control dependence is a subset of a set of statements from a slice computed using our control dependence. The difference is that our control dependence handles non-terminating loops, blind ends and graphs without *EXIT* and *ENTRY* nodes.

We based our new algorithm on finite state machines and regular expressions. The idea is very simple: if we properly label the edges of a CFG, it is basically a finite state machine. In our case, we label the edges by nodes of the CFG (when an edge goes to a node n, it gets a label n), so unrolling the automaton gives us all possible walks through the CFG. That would not be beneficial without a concise representation of all these path. Therefore we transform the labeled CFG to a regular expression and then we generate the *abstract syntax tree* (AST) of the expression. We evaluate the tree from leaves upward and for each node of the tree that has children, we compute a set of nodes through which we always go when we descent into the subtree, and a set of nodes that we may visit only on some paths when we descent into the subtree. To derive control dependencies for a branching node b, we identify all occurrences of b in the AST, and by traversing the AST from each occurrence of b to the right (and up) we find a descriptions of all paths starting in b. Instead of descending into subtrees, we use the pre-computed sets. This way we identify which nodes lay on every path from b and that lay only on some path from b.

Pointer subgraph and pointer analysis optimizations

We include auxiliary nodes into a pointer subgraph when building it. For example, we create a no-operation node as an single entry and another no-operation node as a single exit for a procedure. The objective of these nodes is solely to make generation of a PS easier. Such nodes can be harmlessly removed without any effect on the result of the analysis. Removing such useless nodes from a PS is fast and it could have positive impact on the pointer analysis performance.

Another optimization we plan to implement is reducing a number of nodes based on an equivalence of pointers: when two nodes are proven to have the same point-to sets, we can keep only one of them and use it as a representative of both (i.e. we put them into the same equivalence class).

For example, consider this LLVM code:

```
%1 = load [10 x i32]* %array, i32 0, i32 0
%2 = getelementptr i32* %1, i32 0
%3 = bitcast i32* %p to *i8
```

All the pointers *%1*, *%2*, and *%3* are equal: they point to (%array, 0), therefore we can represent them with a node *%1;2;3* during a pointer analysis. Afterwards, when a user tries to look-up a pointer information for any of those pointers, we return *%1;2;3*.

We could gain further speed up of the pointer analysis with smarter iteration over a PS and by its pre-analysis. For example, we could find loops or statements that lead to the offset blowup problem (Section 3.2.1) and set the *UNKNOWN_OFFSET* directly. That could save us a lot of iterations that the analysis makes until a node run out of the size of allocated memory and fall-back to *UNKNOWN_OFFSET*. Another pre-analysis could detect regions in the code that could be analyzed independently (that do not share any memory, and the pointers can not point between these regions). These regions could be analyzed in parallel [52].

The last but not the least is the implementation of sparse pointer analysis. In fact, our framework fits nicely to the idea of sparse pointer analysis. The analysis just use the nodes it gets from *getMemoryObjects* and it does not care how the pointer information was handled in the background – whether it use sparse or dense propagation.

Other

We though of many other wards of future research and development. Some of them are: a lazy dependence graph building using demand-driven analyses, implementation of hybrid flow-insensitive pointer analysis that would be selectively flow-sensitive (for function pointers and some regions of the code where it is easy and cheap to maintain the flow-sensitive information), support for exceptions, slicing concurrent programs, and, of course, the implementation of the two-pass algorithm of Horwitz et al.

Bibliography

[1] Hiralal Agrawal. "On Slicing Programs with Jump Statements". In: *SIGPLAN Not.* 29.6 (June 1994), pp. 302–312. ISSN: 0362-1340.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. "Detecting Equality of Variables in Programs". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 1–11. ISBN: 0-89791-252-7.

[3] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. DIKU, University of Copenhagen, 1994.

[4] Min Aung et al. "Specialization Slicing". In: *ACM Trans. Program. Lang. Syst.* 36.2 (June 2014), 5:1–5:67. ISSN: 0164-0925.

[5] Thomas Ball and Susan Horwitz. "Slicing Programs with Arbitrary Control-flow". In: *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93, Linköping, Sweden, May 3-5, 1993, Proceedings*. 1993, pp. 206–222.

[6] John P. Banning. "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables". In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. San Antonio, Texas: ACM, 1979, pp. 29–41.

[7] Richard W. Barraclough et al. "A Trajectory-based Strict Semantics for Program Slicing". In: *Theor. Comput. Sci.* 411.11-13 (Mar. 2010), pp. 1372–1386. ISSN: 0304-3975.

[8] Gianfranco Bilardi and Keshav Pingali. "A Framework for Generalized Control Dependence". In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 291–300. ISBN: 0-89791-795-2.

[9] David Binkley. "Precise Executable Interprocedural Slices". In: *ACM Lett. Program. Lang. Syst.* 2.1-4 (Mar. 1993), pp. 31–45. ISSN: 1057-4514.

[10] David W. Binkley and James R. Lyle. "Application of the pointer state subgraph to static program slicing". In: *Journal of Systems and Software* 40.1 (1998), pp. 17–27. ISSN: 0164-1212.

[11] David Binkley and Keith Brian Gallagher. "Program Slicing". In: *Advances in Computers* 43 (1996), pp. 1–50.

[12] Marek Chalupa et al. "Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings". In: ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. Chap. Symbiotic 3: New Slicer and Error-Witness Generation, pp. 946–949. ISBN: 978-3-662-49674-9.

[13] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. "Analysis of Pointers and Structures". In: *SIGPLAN Not.* 25.6 (June 1990), pp. 296–310. ISSN: 0362-1340.

[14] Feng Chen and Grigore Roşu. "Parametric and Termination-sensitive Control Dependence". In: *Proceedings of the 13th International Conference on Static Analysis*. SAS'06. Seoul, Korea: Springer-Verlag, 2006, pp. 387–404. ISBN: 3-540-37756-5, 978-3-540-37756-6.

[15] Jingde Cheng. "Slicing Concurrent Programs: A Graph-Theoretical Approach". In: *Lecture Notes in Computer Science, Automated and Algorithmic Debugging*. Springer-Verlag, 1993, pp. 223–240.

[16] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. "Automatic Construction of Sparse Data Flow Evaluation Graphs". In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '91. Orlando, Florida, USA: ACM, 1991, pp. 55–66. ISBN: 0-89791-419-8.

[17] Jong-Deok Choi and Jeanne Ferrante. "Static Slicing in the Presence of Goto Statements". In: *ACM Trans. Program. Lang. Syst.* 16.4 (July 1994), pp. 1097–1113. ISSN: 0164-0925.

[18] R. Cytron et al. "An Efficient Method of Computing Static Single Assignment Form". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 25–35. ISBN: 0-89791-294-2.

[19] Sebastian Danicic et al. "A unifying theory of control dependence and its application to arbitrary program structures". In: *Theoretical Computer Science* 412.49 (2011), pp. 6809–6842. ISSN: 0304-3975.

[20] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925.

[21] Ben Hardekopf and Calvin Lin. "Flow-sensitive pointer analysis for millions of lines of code". In: *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. 2011, pp. 289–298.

[22] Mark Harman and Sebastian Danicic. "A New Algorithm for Slicing Unstructured Programs". In: *Journal of Software Maintenance* 10.6 (Nov. 1998), pp. 415–441. ISSN: 1040-550X.

[23] Michael Hind et al. "Interprocedural Pointer Alias Analysis". In: *ACM Trans. Program. Lang. Syst.* 21.4 (July 1999), pp. 848–894. ISSN: 0164-0925.

[24] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs". In: *ACM Trans. Program. Lang. Syst.* 12.1 (Jan. 1990), pp. 26–60. ISSN: 0164-0925.

[25] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782.

[26] Jens Krinke. "Advanced slicing of sequential and concurrent programs". PhD thesis. University of Passau, 2004. ISBN: 978-3-8364-7545-7.

[27] Jens Krinke. "Context-sensitive Slicing of Concurrent Programs". In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-11. Helsinki, Finland: ACM, 2003, pp. 178–187. ISBN: 1-58113-743-5.

[28] David J. Kuck et al. "Dependence Graphs and Compiler Optimizations". In: *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*. 1981, pp. 207–218.

[29] Sumit Kumar and Susan Horwitz. "Better Slicing of Programs with Jumps and Switches". In: *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*. FASE '02. London, UK, UK: Springer-Verlag, 2002, pp. 96–112. ISBN: 3-540-43353-8.

[30] Arun Lakhotia and Jean-christophe Deprez. *Precise slices of block-structured programs with goto statements*. 1997.

[31] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and OptimizAation (CGO'04)*. Palo Alto, California, 2004.

[32] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Trans. Program. Lang. Syst.* 1.1 (Jan. 1979), pp. 121–141. ISSN: 0164-0925.

[33] Mangala Gowri Nanda and S. Ramesh. "Slicing Concurrent Programs". In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '00. Portland, Oregon, USA: ACM, 2000, pp. 180–190. ISBN: 1-58113-266-2.

[34] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. *Understanding Data Dependences in the Presence of Pointers*. Tech. rep. 2003.

[35] Karl J. Ottenstein and Linda M. Ottenstein. "The Program Dependence Graph in a Software Development Environment". In: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*. 1984, pp. 177–184.

[36] Hemant D. Pande, William Landi, and Barbara G. Ryder. *Interprocedural Reaching Definitions in the Presence of Single Level Pointers*. Tech. rep. 1992.

[37] Thomas W. Reps. "Program Analysis via Graph Reachability". In: *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*. 1997, pp. 5–19.

[38] Thomas W. Reps and Wuu Yang. "The Semantics of Program Slicing and Program Integration". In: *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages*. TAPSOFT '89. London, UK, UK: Springer-Verlag, 1989, pp. 360–374. ISBN: 3-540-50940-2.

[39] Thomas Reps et al. "Speeding Up Slicing". In: *SIGSOFT Softw. Eng. Notes* 19.5 (Dec. 1994), pp. 11–20. ISSN: 0163-5948.

[40] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Program-*

ming Languages. POPL ’88. San Diego, California, USA: ACM, 1988, pp. 12–27. ISBN: 0-89791-252-7.

[41] Marc Shapiro and Susan Horwitz. “Fast and Accurate Flow-insensitive Points-to Analysis”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’97. Paris, France: ACM, 1997, pp. 1–14. ISBN: 0-89791-853-3.

[42] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. “System-dependence-graph-based Slicing of Programs with Arbitrary Interprocedural Control Flow”. In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE ’99. Los Angeles, California, USA: ACM, 1999, pp. 432–441. ISBN: 1-58113-074-0.

[43] Jiří Slabý, Jan Strejček, and Marek Trtík. “Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution - (Competition Contribution)”. eng. In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013*. Berlin, Heidelberg: Springer, 2013, pp. 630–632. ISBN: 978-3-642-36741-0.

[44] Bjarne Steensgaard. “Points-to Analysis in Almost Linear Time”. In: *Conference Record of POPL’96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 1996, pp. 32–41.

[45] *The LLVM Compiler Infrastructure*. URL: `llvm.org` (visited on 03/23/2016).

[46] Frank Tip. “A survey of program slicing techniques”. In: *J. Prog. Lang.* 3.3 (1995).

[47] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. “Efficient Flow-sensitive Interprocedural Data-flow Analysis in the Presence of Pointers”. In: *Proceedings of the 15th International Conference on Compiler Construction*. CC’06. Vienna, Austria: Springer-Verlag, 2006, pp. 17–31. ISBN: 3-540-33050-X, 978-3-540-33050-9.

[48] Mark Weiser. “Program Slicing”. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0-89791-146-6.

[49] Mark Weiser. “Program Slicing”. In: *IEEE Trans. Softw. Eng.* 10.4 (July 1984), pp. 352–357. ISSN: 0098-5589.

[50] Mark Weiser. “Programmers Use Slices when Debugging”. In: *Commun. ACM* 25.7 (July 1982), pp. 446–452. ISSN: 0001-0782.

[51] Baowen Xu et al. “A Brief Survey of Program Slicing”. In: *SIGSOFT Softw. Eng. Notes* 30.2 (Mar. 2005), pp. 1–36. ISSN: 0163-5948.

[52] Sen Ye, Yulei Sui, and Jingling Xue. “Static Analysis: 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings”. In: ed. by Markus Müller-Olm and Helmut Seidl. Cham: Springer International Publishing, 2014. Chap. Region-Based Selective Flow-Sensitive Pointer Analysis, pp. 319–336. ISBN: 978-3-319-10936-7.