# Static Backward Demand-Driven Slicing

Björn Lisper Abu Naser Masud Husni Khanfar
School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
{bjorn.lisper,masud.abunaser,husni.khanfar}@mdh.se

## Abstract

Program slicing identifies the program parts that may affect certain properties of the program, such as the outcomes of conditions affecting the program flow. Ottenstein's Program Dependence Graph (PDG) based algorithm is the state-of-practice for static slicing today: it is well-suited in applications where many slices are computed, since the cost of building the PDG then can be amortized over the slices. But there are applications that require few slices of a given program, and where computing all the dependencies may be unnecessary. We present a light-weight interprocedural algorithm for backward static slicing where the data dependence analysis is done using a variant of the Strongly Live Variables (SLV) analysis. This allows us to avoid building the Data Dependence Graph, and to slice program statements "on-the-fly" during the SLV analysis which is potentially faster for computing few slices. Furthermore we use an abstract interpretation-based value analysis to extend our slicing algorithm to slice low-level code, where data dependencies are not evident due to dynamically calculated addresses. Our algorithm computes slices as sets of Control Flow Graph nodes: we show how to adapt existing techniques to generate *executable slices* that correspond to semantically correct code, where jump statements have been inserted at appropriate places. We have implemented our slicing algorithms, and made an experimental evaluation comparing them with the standard PDG-based algorithm for a number of example programs. We obtain the same accuracy as for PDG-based slicing, sometimes with substantial improvements in performance.

***Categories and Subject Descriptors*** F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.2.6 [*Software Engineering*]: Programming Environments

***Keywords*** Static backward slicing; Unstructured control flow; Data flow equations; Computational complexity; Strongly live variable; Abstract Interpretation

## 1. Introduction

Program slicing refers to a collection of techniques to identify which parts in a program may affect a so called "slicing criterion" that expresses certain properties of a program. The slicing criterion may be, for instance, the possible values of some program variables in some program points. The result of the slicing is usually a new program, formed by extracting certain statements from the original code. Program slicing was first considered by Weiser [44] in the context of debugging. Other applications of slicing have emerged since then, including program comprehension [20], integration [9], testing [5, 16], parallelization of sequential code [45], software maintenance [15], compiler optimization [30] and many more.

Slicing comes in different dimensions. *Static slicing* computes a safe overapproximation of the code that might affect, or be affected by the slicing criterion, whereas *dynamic slicing* considers the statements that will affect (or be affected by) the criterion in different runs. *Intraprocedural slicing* is performed on a single nonrecursive procedure whereas *interprocedural slicing* considers multiple procedures possibly containing multiple call sites. *Syntax-preserving* slices are the subset of the original program statements whereas *amorphous* slicing transforms program code. *Backward slicing* computes the code-segment that affects the slicing criterion whereas *forward slicing* computes the code-segment that is affected by the slicing criterion.

Today, state of the practice in static slicing are algorithms based on the Program Dependence Graph (PDG) [14, 23, 35]. These algorithms first build the PDG for the whole program, and then compute the slices by a simple linear-time graph traversal. This is good for applications like program understanding, where many slices may be taken using different slicing criteria, since then the cost of building the PDG can be amortized over the different slices taken.

However, there are applications that require to compute only few slices. An example is Worst-Case Execution Time (WCET) analysis [47], where supporting analyses to constrain the WCET program flow can benefit greatly from slicing the analysed program with respect to the conditions, thus removing the parts of the program that surely cannot affect the control flow [13, 31, 40]. For such applications it can be advantageous to compute dependencies on the fly, for the parts of the program that actually produce the slice, rather than for the whole program.

In this paper we present such an algorithm for static backward slicing. We start with an intraprocedural analysis, which is subsequently extended to an interprocedural analysis. These analyses work for a high-level view of memory that consists of distinct program variables. Next we extend the analyses to slice code with a low-level view of memory, where reads and writes are made to addresses that may be dynamically computed rather than to program variables of given type and size. This is interesting for applications such as the aforementioned WCET analysis, which often is performed on linked binaries. Finally we show how to generate executable slices from our computed slices, which are sets of Control-Flow Graph (CFG) nodes. These executable slices can be directly translated into executable (or compilable) textual code.

This paper makes the following contributions:



*PEPM '15*, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3297-2/15/01.
http://dx.doi.org/10.1145/2678015.2682538

1. An interprocedural slicing algorithm is developed for high-level code based on the *Relevant Strongly Live Variables* (SLV) dataflow analysis on the CFG representation of the input program. This analysis is a modification of standard SLV analysis, where the generated SLVs carry a dependency relation with the slicing criterion. This algorithm slices program statements on the fly during the analysis of SLVs.
2. We show how to adapt the SLV-based slicing algorithm to slice code with a low-level memory model, by augmenting it with an abstract interpretation-based value analysis that is used to determine addresses for memory accesses in the dataflow analysis.
3. We show how to convert the slices into executable slices, which can be directly translated into textual code with the same semantics as the original, CFG-based slices.
4. We have implemented the SLV-based algorithm and the standard PDG-based slicing algorithm in the WCET analysis tool SWEET [1, 32], and we have performed a comparative evaluation on a number of example programs. The evaluation shows that our algorithm computes the same slices as the PDG-based algorithm, sometimes with a significantly lower execution time.

The rest of this paper is organized as follows. Section 2 reviews some basic theory of data flow analysis, and introduces some notations used in the rest of the paper. Section 3 presents the SLV-based intraprocedural slicing algorithm and discusses its complexity. Section 4 extends the intraprocedural slicing algorithm to interprocedural slicing. In Section 5 we show how the SLV-based slicing can be extended to slice low-level code, and produce executable slices. Experimental results are given in Section 6, Section 7 gives an account for related work, and Section 8 concludes the paper.

## 2. Preliminaries

We now introduce some standard concepts and notation for completeness and clarity. A *control-flow graph* (CFG) [34] is defined as a directed graph $(N, Flow)$ where the nodes in $N$ are labeled either with conditions, assignments, a special label $start$, or ditto $stop$, and $Flow \subseteq N \times N$ is a relation describing the possible flows of execution in the graph. Each CFG contains a unique start node and a unique stop node. The start node has no predecessors, and the stop node no successors. An assignment node has exactly one successor, and a condition node has exactly two successors (labeled *true* and *false*, respectively).

We will sometimes write $[c]^n$ for a node $n$ labelled with the condition $c$, and $[x := a]^n$ for a node $n$ labelled with the assignment $x := a$. This notation makes it easier to define the data flow equations in Sections 2.1 and 3.1.

Conditions, and right-hand sides in assignments, are expressions. We assume that expressions have no side-effects, and that they are simple expressions built from program variables, constants, operators and primitive functions (not user-defined). For simplicity we do not allow pointers and operations on such: all the analyses presented here can however be extended to deal with them. We assume that program variables are unaliased, i.e., an assignment to the program variable $x$ can not affect the value of another program variable $y \neq x$. For an expression $e$, $FV(e)$ denotes the set of program variables that appear in $e$.

Note that we label the nodes by single statements, whereas in compiler literature the nodes often are considered to represent basic blocks. This is for two reasons: first, we want to perform the slicing on statements rather than basic blocks, and similarly it is easier to define data flow analyses and other static program analyses by equations over statements.

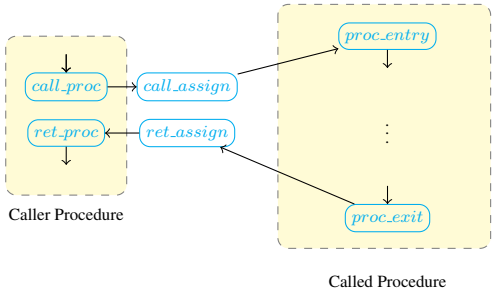


Figure 1: Relations among node types at call site in the CFG

For procedure calls, our CFG representation is based on [34] with some extensions. Some new kinds of nodes are introduced for representing procedure calls and entries: *proc_entry* and *proc_exit* nodes correspond to procedure entry and procedure exit. Without loss of generality, we assume that procedure parameters can be divided into a set of input and a set of output parameters where output parameters may be updated by the procedure. A procedure call is then represented by four types of nodes in the CFG: *call_proc* nodes represent the procedure calls, *call_assign* nodes represent the assignments of actual input parameters to formal input parameters, *ret_assign* nodes represent the assignments of formal output parameters to the actual output parameters, and *ret_proc* nodes represent returns from procedure calls.

There are some rules how procedure call nodes can appear. *call_proc* nodes and *call_assign* nodes always come in pairs, where the *call_assign* node is the unique successor to the *call_proc* and vice versa. The relation is similar for *ret_assign* and *ret_proc* nodes, but the order is reversed. The (unique) successor of a *call_assign* node must be a *proc_entry* node, and every predecessor of a *proc_entry* node must be a *call_assign* node. Similarly the unique predecessor to a *ret_assign* node must be a *proc_exit* node, and every successor of a *proc_exit* node must be a *ret_assign* node. *call_proc* and *ret_proc* nodes belong to the CFG of the caller, *proc_entry* and *proc_exit* nodes belong to the CFG of the callee, and the *call_assign* and *ret_assign* nodes are special nodes which may contain variables (in assignment expressions) that are scoped in both caller and callee procedures. See Fig. 1 and 2 for examples.

*Postdominators* [36] play an important role in slicing. A node $n$ in a CFG is said to postdominate a node $n_0$ if and only if every path from $n_0$ to the stop node goes through $n$. There are many algorithms to compute postdominators, as well as *postdominator trees* which can be used to efficiently represent sets of postdominators for different nodes. Postdominator relations are used to determine *control dependencies*, see Section 2.2.

### 2.1 Data Flow Analysis

An important part of slicing is to compute data dependencies. A data flow analysis [34] is often used for this purpose. Data flow analyses are usually defined over CFGs in the following way. For each node $n$ in the CFG, the analyses compute sets $S_{entry}(n)$ and $S_{exit}(n)$ which are present before and after the node, respectively. The sets represent some kind of data flow information. Depending on the direction of the data flow computed, an analysis is a *forward* or *backward* analysis. The sets are related through equations. We restrict our attention to *bit vector analyses*: for a backward bit-

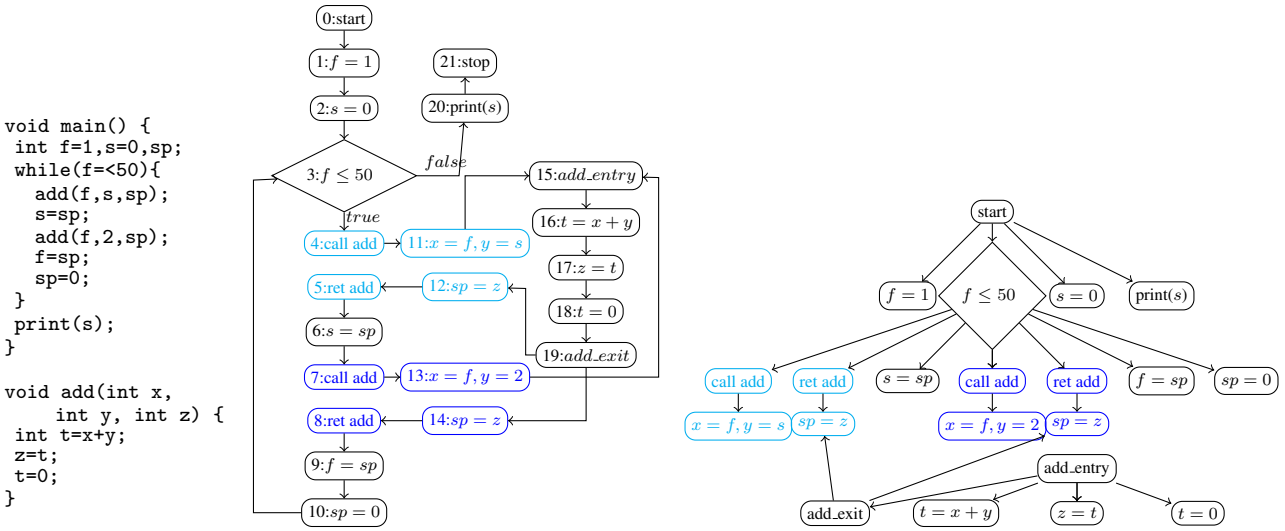


Figure 2: CFG and CDG of the Running Example

vevtor analysis, the form of the equations is as follows:

$$\begin{array}{lcll} S_{exit}(stop) & = & S_{init} & \\ S_{entry}(stop) & = & S_{exit}(stop) & \\ S_{entry}(start) & = & S_{exit}(start) & \\ S_{entry}(n) & = & (S_{exit}(n) \setminus kill(n)) \cup gen(n), & (1) \\ & & \text{where } n \notin \{start, stop\} & \\ S_{exit}(n) & = & \bigcup_{n' \in succ(n)} S_{entry}(n'), & \\ & & \text{where } n \notin \{start, stop\} & \end{array}$$

(The equations for forward bit vector analyses are similar, see [34].) Here, $succ(n)$ is the set of immediate successors to $n$ in the CFG, and $S_{init}$ is some set describing the data flow information that is present at the exit of the program. The equations defining $S_{entry}(n)$ have the form $S_{entry}(n) = f_n(S_{exit}(n))$, where $f_n$ is the *transfer function* of node $n$. Data flow analyses are further divided into may and must analyses. We will only deal with may analyses here: (1) is valid for these.

A classical data flow analysis is Reaching Definitions (RD). It is a forward may analysis which computes sets of pairs $(x, n)$, where $x$ is a program variable and $n$ is a node in the CFG. If $(x, n)$ belongs to the set associated with node $p$ then the value of $x$ that was assigned at $n$ may still reside in $x$ at $p$, and then there is a possible data flow from $n$ to $p$.

RD can be used to compute def-use pairs of variable assignments and possible uses. The def-use pairs for a program constitute its *Data Dependence Graph*, which is used in PDG-based slicing: see Section 2.2.

(1) yields a system of $2 \cdot |N|$ equations, where the unknowns are the sets $S_{entry}(n)$ and $S_{exit}(n)$ for the different nodes $n$ in the CFG. The classical way to solve the system is by *fixed-point iteration*: all unknown sets are initialized to $\emptyset$, and then the system of equations is iterated until no more sets change. The underlying theory of complete lattices [34] ensures that this procedure converges, and always yields the least (most precise) solution. There is a generic worklist (or work set) algorithm to perform fixed-point iteration [34] that handles both forward and backward analyses. It computes an array $S$ of sets, indexed by the nodes in the CFG $(N, Flow)$, where each element $S[n]$ equals $S_{entry}(n)$ for forward analyses, and $S_{exit}(n)$ for backward analyses.

The number of fixed-point iterations can be at most $h \cdot |N|$, where $h$ is the height of the lattice iterated over and $|N|$ is the number of nodes in the CFG. For dataflow analysis, $h$ equals the size of the largest possible set of data flow information. Thus the maximal execution time is $O(t \cdot h \cdot |N|)$, where $t$ is an upper bound for the time needed to perform one fixed-point iteration.

### 2.2 PDG-based Slicing

The standard algorithms for static backward slicing use the Program Dependence Graph (PDG) [14, 23, 35]. The PDG is the union of two graphs, whose nodes are the same as those in the CFG: the Data Dependence Graph (DDG) and the Control Dependence Graph (CDG). The DDG is usually computed using the aforementioned RD dataflow analysis. The CDG has edges from conditions to CFG nodes, where there is an edge from condition $c$ to node $n$ if the outcome of $c$ possibly can affect whether $n$ is executed or not (i.e., $n$ is *control dependent* on $c$). This can be formalised through postdominators: we say that $n$ is control dependent on $c$ if there is a path from $c$ to $n$ in the CFG and if $n$ does not postdominate $c$. The CDG can be computed efficiently using postdominator trees. Fig. 2 shows the CFG and CDG for a given example program.

Once the PDG is built, the backward slicing can be performed by a simple graph search for the nodes in the PDG that are backward reachable from the slicing criterion. This operation is linear in the number of nodes and edges of PDG that are part of the sliced code, and can thus be performed quickly for different slicing criteria.

## 3. An Intraprocedural Slicing Based on Relevant SLV Analysis

As mentioned in the introduction, we do not want to compute all the data dependencies before the slicing. Rather we want to discover the relevant data dependencies on demand during slicing. In the following, we describe the Relevant Strongly Live Variables analysis which allows us to perform slicing concurrently with the data dependence analysis.

### 3.1 The Relevant Strongly Live Variables Analysis

The Strongly Live Variables (SLV) analysis (Exercise 2.4 in [34]) is an alternative data flow analysis for computing data dependencies. Given some sets of variables in some different program points, representing "uses" of these variables (like being written to some out-

put device), it computes for each program point a set of "strongly live variables" whose values in that program point might (transitively) reach a use of some variable. It is a backward may analysis, with *kill*, *gen* and $S_{init}$ sets defined as follows:

$$\begin{array}{rcl} S_{init} & = & \emptyset \\ kill([x := a]^n) & = & \{x\} \\ gen([x := a]^n) & = & FV(a) \\ kill([c]^n) & = & \emptyset \\ gen([c]^n) & = & FV(c) \end{array} \quad (2)$$

The transfer functions for SLV analysis are set up according to (1), however with the following modification (for $n$ being an assignment or a condition):

$$S_{entry}(n) = \begin{cases} (S_{exit}(n) \setminus kill(n)) \cup gen(n), \\ \quad \text{if } kill(n) \subseteq S_{exit}(n) \\ S_{exit}(n), \text{ otherwise} \end{cases} \quad (3)$$

The above defines the "standard" SLV analysis, as known from the literature. It is based on the following assumptions on variable uses:

- variables are not used at the end of a program,
- variables in conditions are always used, and
- variables are not used in any other program points.

For slicing, a slicing criterion corresponds to a set of variable uses. To allow for a more flexible specification of slicing criteria, not necessarily adhering to the assumptions on variable uses above, we introduce *Relevant SLVs* defined as follows:

**Definition 3.1** (Relevant Strongly Live Variables)**.** *The Relevant Strongly Live Variables at any program point are those strongly live variables on which the variables at the slicing criterion have data dependency.*

So, relevant SLVs are a kind of SLVs that can affect some slicing criterion through data dependencies. They are defined through the following, modified equations. Let the slicing criterion be specified by a (possibly empty) set $S_{crit}(n)$ for each node $n$ in the CFG. The *kill*, *gen* and $S_{init}$ sets for Relevant SLV analysis are then defined according to (2) with the following modifications:

$$\begin{array}{lcl} S_{init} & = & S_{crit}(stop) \\ gen([x := a]^n) & = & FV(a) \cup S_{crit}(n) \\ gen([c]^n) & = & S_{crit}(n), \quad c \text{ is a condition} \end{array} \quad (4)$$

The second case in (3) is modified accordingly:

$$S_{entry}(n) = S_{exit}(n) \cup S_{crit}(n), \text{ if } kill(n) \not\subseteq S_{exit}(n) \quad (5)$$

For brevity we will refer to Relevant SLVs as SLVs below.

### 3.2 A Worklist Algorithm for Slicing on the Fly

In each program point, the relevant strongly live variables are the ones that can carry a dependence to some slicing criterion. Thus, for an assignment $x := a$, if $x$ is strongly live at exit then there is a possible data dependence from the assignment to some slicing criterion, and it can be immediately put into the slice. We call this "slicing on-the-fly" as it slices during the SLV analysis, in contrast to data dependence analysis with RD where an explicit data dependence graph has to be built first. We now describe an algorithm to perform static backward slicing based on slicing on-the-fly. The slicing algorithm in Fig. 3 computes data dependencies according to the Relevant SLVs defined in Section 3.1, it traces control dependencies, and it performs slicing-on-the-fly. It is an adaptation of the generic worklist algorithm for data flow analysis [34] modified in the following way:

**Algorithm 1:** $SLVSlicing(N, F, C, S_{crit}, \langle f_n | n \in N \rangle)$

```
   /* Initialization */
1  W := ∅ ;
2  N_slice := ∅;
3  forall n ∈ N do
4  |  S[n] := S_crit[n];
5  |  if S_crit[n] ≠ ∅ then
6  |  |  N_slice := N_slice ∪ {n};
7  |  |  forall n' where (n, n') ∈ F do
   |  |  W := W ∪ {(n, n', CFG)};
8  |  |  forall n' where (n', n) ∈ C do
   |  |  W := W ∪ {(n, n', CDG)};
9  |  end
10 endfor
   /* Iteration */
11 while W ≠ ∅ do
12 |  (n, n', T) := Select(W)      /* Pick new element
   |  from W */;
13 |  W := W \ {(n, n', T)};
14 |  if T = CFG ∧ f_n(S[n]) ⊈ S[n'] then  /* S[n'] has
   |  changed: add new work item to W */
15 |  |  if n' is an assignment x := a and x ∈ f_n(S[n])
   |  |  then
16 |  |  |  N_slice := N_slice ∪ {n'};
17 |  |  |  forall n'' where (n'', n') ∈ C do
   |  |  |  W := W ∪ {(n', n'', CDG)};
18 |  |  end
19 |  |  S[n'] := S[n'] ∪ f_n(S[n]) ;
20 |  |  forall n'' where (n', n'') ∈ F do
   |  |  W := W ∪ (n', n'', CFG);
21 |  end
22 |  if T = CDG then      /* The edge is a control
   |  dependence */
23 |  |  S[n'] := S[n'] ∪ FV(c(n')) ;
24 |  |  if S[n'] ≠ S[n'] \ FV(c(n')) or n' ∉ N_slice then
   |  |  /* to avoid potential infinite loop */
25 |  |  |  N_slice := N_slice ∪ {n'};
26 |  |  |  forall n'' where (n', n'') ∈ F do
   |  |  |  W := W ∪ (n', n'', CFG);
27 |  |  |  forall n'' where (n'', n') ∈ C do
   |  |  |  W := W ∪ {(n', n'', CDG)};
28 |  |  end
29 |  end
30 end
   /* Finalization */
31 return N_slice
```

Figure 3: Slicing-on-the-fly with SLV analysis

- The worklist $W$ holds two types of edges, which are tagged to keep them apart: $(n, n', CFG)$ for edges in the CFG, and $(n, n', CDG)$ for edges in the CDG.
- The algorithm uses the explicit CDG $C$ to perform slicing due to control dependencies. $F$ is the set of reversed edges $Flow^{-1}$.
- A set $N_{slice}$ of sliced CFG nodes is maintained by the algorithm.
- A node $[x := a]^n$ is included in the slice whenever $x \in S[n]$. This takes care of slicing due to data dependencies.
- If $n$ is included in the slice, and if $(n', n) \in C$ (where $n'$ is a conditional node with condition $c$), then $n'$ is included in the slice as well, $FV(c)$ is added to $S[n']$, for each edge $(n'', n') \in C$, $(n', n'')$ is added to the worklist $W$, and if $S[n']$ changes then all edges $(n', n'') \in F$ are added to the worklist as well.
- The array $S$ is initialized to an array $S_{crit}$ containing the slicing criteria (sets of variables in certain program points). Furthermore, $N_{slice}$ is initialized to the set of nodes $n$ where $S_{crit}[n] \neq \emptyset$.
- For any node $n$, the transfer function $f_n$ is defined according to (4) and (5).

In Algorithm 1 $select(W)$ picks non-deterministically an element from $W$, and for any conditional nodes $n$ in the CFG $c(n)$ denotes the condition of $n$.

Each SLV set can contain at most $v$ elements, where $v$ is the number of variables in the program. Thus, the total size of the SLV sets is $O(|N| \cdot v)$, where $|N|$ is the number of nodes in the CFG. The height of the lattice is also $v$ and thus the time to do fixed-point iteration for the standard SLV analysis is $O(t \cdot |N| \cdot v)$ where $t$ is an upper bound to the time required to perform one fixed-point iteration. The complexity for Algorithm 1 is essentially the same, since the termination criterion for the worklist iteration is the same as well as the lattice of SLV sets. The factor $t$ will of course change, but its order should be the same as for the original SLV analysis since the same set operations are used.

### 3.3 Slicing on the Fly With Respect to all Conditions

As mentioned in Section 1, in the flow analysis phase of WCET analysis it is often interesting to slice with respect to all the conditions in the program. In the SLV slicing algorithm, the slicing criterion will thus consist of all conditions in the program plus their sets of variables. As the conditions thus are contained in the slice from the beginning, the control dependencies can be disregarded. This is since the sole function of computing the control dependencies is to determine which conditions to include in the slice: thus, if all conditions are already included, the control dependencies will have no use and the CDG need not even be generated.

Algorithm 1 can be simplified accordingly, by removing lines 8, 17 and 22-29. Also the worklist elements will not have to be tagged as CFG or CDG edges as no CDG is needed. This is close to the original SLV algorithm as described in Section 3.1. The worst-case complexity will be the same as for the general SLV slicing algorithm, but the real execution time can be expected to be lower since the handling of the CDG is eliminated. As mentioned the CDG does not even have to be generated, which eliminates another phase in the slicing. This results in an algorithm that works directly on the CFG and does not require any other additional data structures than the worklist and the SLV sets.

## 4. Interprocedural Slicing

Any interprocedural slicing technique needs to deal with two problems. The first one is how to handle context-sensitivity: a very context-sensitive slicing will be precise but can be time-consuming, whereas a less context-sensitive approach will be faster but less precise. The second problem is how to precisely trace the dependencies between the input and output arguments of procedure calls. This is non-trivial since procedure calls may be arbitrarily nested and program variables might be aliased.

For PDG-based slicing, a solution has been developed where the PDG is extended into the *System Dependence Graph* (SDG) that captures also the interprocedural dependencies [23]. The SDG allows to handle both the context-sensitivity and the tracing of dependencies [23, 38]. As we avoid building the PDG (and thus also the SDG), we have developed a solution where these problems instead are handled on-the-fly. Our solution is context-sensitive and works for programs without (direct or indirect) recursive procedures. This may yield a costly algorithm: however, it turns out that we can mitigate this problem in a manner that is similar to how it is handled in the SDG-based algorithm. Note that even though our method works for non-recursive procedures, it is possible to extend it to handle recursive procedures as well by using some of the optimizations discussed in Section 4.1.

It is worth noting that Weiser's interprocedural slicing algorithm [42, 44], which is not SDG-based and has some similarities with our algorithm, does not handle these two problems and thus is less precise.

As mentioned in Section 2, a call site is represented by four distinct nodes in our CFGs: a $call\_proc$, a $call\_assign$, a $ret\_assign$, and a $ret\_proc$ node. The call site nodes are connected to a $proc\_entry$ and $proc\_exit$ node representing the entry and exit of the called procedure.

Some control dependencies are added in the interprocedural case. Every node in the CFG of a procedure $P$ including the $proc\_exit$ node is control dependent on its $proc\_entry$ node. So, if any node in $P$ is sliced then its $proc\_entry$ node is sliced as well. The $call\_proc$ and the $ret\_proc$ nodes correspond to the same program statement, and the $call\_assign$ and the $ret\_assign$ nodes correspond to the assignments of actual arguments into formal arguments. So, the $call\_assign$ and the $ret\_assign$ nodes in a procedure call are control dependent on the $call\_proc$ and the $ret\_proc$ nodes respectively. Moreover, the $ret\_assign$ node is control dependent on the corresponding $proc\_exit$ node. Note that the sole purpose of these control dependencies is to include some relevant call site nodes into the slice. When a procedure containing procedure calls is being sliced by computing the SLV sets for each of its nodes, as soon as any $ret\_proc$ node is reached during the backward traversal of the CFG the slicing of the current procedure is postponed, and instead slicing continues to the called procedure (with some exceptions described later in this section). The called procedure is then sliced with respect to the SLV set obtained at the $proc\_exit$ node from the caller. This SLV set can be considered as the slicing criterion for the called procedure. Once the called procedure has been sliced, the slicing continues at the original call site.

**Example 4.1.** Let us consider the CFG in Fig. 2. Suppose the SLV analysis reaches node 15 (the *add_entry* node). Now the next visited node can be (i) node 11 corresponding to the first call site of procedure *add*, or (ii) node 13 corresponding to the second call site of the same procedure. If node 15 is reached from node 12 during the SLV analysis then the SLV analysis should continue to node 11, otherwise node 13.

We solve the calling-context problem by introducing a unique ghost variable $g_n$ into the set $S[n]$ for the $ret\_assign$ node $n$ in each call site. The transfer functions for the call site nodes are

defined as follows.

$$S_{entry}(n)=\begin{cases} S_{exit}(n) & n:T \\ \bigcup_i S_{exit}(n)|_{x_i\to FV(e_i)}\cup\{g_n\} & n:ret\_assign \\ \bigcup_i S_{exit}(n)|_{x_i\to FV(e_i)}\setminus\{g_{rasgn(n)}\} & n:call\_assign \\ S_{exit}(n)\cap[vars(proc(n))\cup G_V] & n:proc\_exit \\ S_{exit}(n)\cup[S_{exit}(retp(n))\setminus kill(rasgn(n))] & n:call\_proc \end{cases} \tag{6}$$

Here, $n:X$ represents that node $n$ is of type $X$, $T$ can be $ret\_proc$ or $proc\_entry$, and $proc(n)$ is the procedure containing the node $n$, $vars(P)$ returns the set of variables in the scope of procedure $P$, $retp(n)$ and $rasgn(n)$ return the $ret\_proc$ and $ret\_assign$ nodes respectively of the corresponding node $n$ belonging to the same call site, $g_n$ is the unique ghost variable for the call site containing the $ret\_assign$ node $n$, and $G_V$ is the set of all ghost variables. Furthermore we define

$$S_{exit}(n)|_{x_i\to FV(e_i)}=\begin{cases} S_{exit}(n)\setminus\{x_i\}\cup FV(e_i) & \text{if } x_i\in S_{exit}(n) \\ S_{exit}(n) & \text{otherwise} \end{cases}$$

The $call\_assign$ and $ret\_assign$ nodes contain lists of concurrent assignments $x_1=e_1,\ldots,x_m=e_m$ for some variables $x_i$ and expressions $e_i$. As mentioned in Section 2, the $call\_assign$ node contains the assignments of actual input parameters into the formal input parameters, and the $ret\_assign$ node includes the assignments of formal output parameters into the actual output parameters.

Now suppose $n_1,\ldots,n_k$ are the $call\_assign$ nodes corresponding to $k$ call sites calling any procedure $P$ in the input program. The $proc\_entry$ node $n_0$ of procedure $P$ is the successor node of each $n_i$. If the SLV analysis described in Section 3 reaches node $n_0$, the worklist will be updated by $W=W\cup\{(n,n_j,CFG)\}$ where $g_j\in S[n_j]\cap G_V$ represents the ghost variable corresponding to the call site $j$. That means the previous slicing of the caller procedure of $P$ is resumed at call site $j$ that has been postponed before. However, if $S[n_j]\cap G_V=\emptyset$, it means that the slicing of the current procedure is not invoked by any of its callers. Since all the caller procedures are backward reachable from the current procedure, they should be sliced as well. In such case, the worklist should be updated by $W=W\cup\{(n,n_i,CFG)\mid 1\le i\le k\}$. These conditions can be included in line 20 of Algorithm 1.

Note that introducing the ghost variable in each call site makes the interprocedural data flow analysis very context sensitive. Different call patterns generate distinguishing sequences of ghost variables representing call strings obtained from the corresponding $proc\_entry$ nodes. Since every call site has a distinguishing $ret\_assign$ node, introducing a unique ghost variable at this node distinguishes calls from different call sites. Moreover, since we do not allow recursion, if the $Select$ function in Algorithm 1 picks the most recent element from the worklist, it does not merge data flow from different call sites. This is because even though ghost variables are inserted into the $ret\_assign$ nodes, they are discarded from the $call\_assign$ nodes. So, (6) merges contexts where relevant but splits contexts where appropriate.

### 4.1 Avoiding Unnecessary Multi-Pass Analysis of Procedures

If a procedure is called several times from multiple call sites, it may be required to slice a procedure more than once possibly with different slicing criteria. Here, the slicing criterion for a procedure corresponds to the SLV set of the *proc_exit* node in the CFG of the called procedure. If the *proc_exit* node of any procedure obtains a slicing criterion during the SLV analysis and it has been sliced before with the same slicing criterion, then the called procedure does not need to be analyzed again for slicing. Instead, the SLV set in the corresponding *proc_entry* node of the called procedure can be reused to continue the slicing in the caller procedure. This essentially requires that we save the SLV set at the *proc_exit* and *proc_entry* node of each procedure. If the newly generated slicing criterion is a superset of the previously generated slicing criterion, the previous analysis results can be reused in the extended analysis of the called procedure. This will reduce the negative impact of the context-sensitivity on the performance of the slicing.

Suppose procedure $P$ is already sliced with criterion $C_1$. If it needs to be sliced again with criterion $C_2$ such that $C_2\supset C_1$, then intuitively, it is enough to slice $P$ with the criterion $(C_2\setminus C_1)$. This may reduce the running time of the slicing, as a smaller slicing criterion on average should yield faster convergence of the fixed-point iteration in Algorithm 1. The intuition behind this optimization is as follows. Suppose a CFG node of the form $[x:=e]^n$ is sliced during slicing the procedure $P$ with the slicing criterion $C_2$. Then either $x\in C_2$, or $x\in S[n']$ and $n'$ already belongs to the slice of $P$. If $x\in C_2$, then either $x\in C_1$ or $x\in(C_2\setminus C_1)$ and hence node $n$ should be sliced during slicing the procedure $P$ with criterion either $C_1$ or $(C_2\setminus C_1)$. However, when $x\in S[n']$ for some node $n'$ already in the slice during slicing with criterion $C_2$, it can be proven inductively that $n'$ belongs to the sliced set of $P$ when it is sliced either with $C_1$ or with $(C_2\setminus C_1)$. This intuition suggests that if slicing $P$ generates the SLV set $S_{C_1}[k]$ and $S_{C_2\setminus C_1}[k]$ when slicing $P$ with criterion $C_1$ and $C_2\setminus C_1$ at the $proc\_entry$ node $k$ of $P$, then the SLV set $(S_{C_1}[k]\cup S_{C_2\setminus C_1}[k])$ should be generated at $k$ when $P$ is sliced with criterion $C_2$. We leave a formal proof of this intuition as future work.

If $C_2\subseteq C_1$ then $P$ does not need to be sliced again at all, which definitely saves slicing time. But in order to continue slicing the caller of $P$ without slicing $P$ again, the $call\_proc$ node at the call site must obtain the right SLV set. This requires to keep the data dependence relation from the output parameters to the input parameters of the already analyzed procedure. During the update of any SLV set $S_{entry}(n)$ for any node $n$ by the transfer function (3), we update the dependence relation $\mathbb{H}_P$, from procedure variables to procedure output parameters of $P$, which is initially empty. For any node $n$, suppose $kill(n)\subseteq S_{exit}(n)$. Then for all $v\in gen(n)$, we update first $\mathbb{H}_P[v]$ and then $\mathbb{H}_P[t]$ as follows:

$$\begin{aligned} \mathbb{H}_P[v]= &\quad \mathbb{H}_P[v]\cup\{x\mid x\in kill(n), x\in out(proc(n))\}\cup \\ &\quad \{x\mid x\in\mathbb{H}_P[t], t\in kill(n), t\notin out(proc(n))\} \\ \mathbb{H}_P[t]= &\quad \emptyset \text{ for all } t\in kill(n), t\notin out(proc(n)) \end{aligned} \tag{7}$$

where $out(P)$ is the set of output arguments of $P$. Moreover, any set $S[n]$ for the conditional node $[c]^n$ is updated due to the control dependency edge $(n,n')$ in the CDG where node $n'$ is already in the slice. In such case, for all $v\in FV(c(n))$, we update $\mathbb{H}_P[v]$ as follows:

$$\begin{aligned} \mathbb{H}_P[v]= &\quad \mathbb{H}_P[v]\cup\{x\mid x\in S_{exit}(n'), x\in out(proc(n'))\}\cup \\ &\quad \{x\mid x\in\mathbb{H}_P[t], t\in S_{exit}(n'), t\notin out(proc(n'))\} \end{aligned} \tag{8}$$

Note that $\mathbb{H}_P$ keeps all the data and control dependences from procedure variables to formal output parameters. In order to obtain the dependences from formal output parameters to formal input parameters, we need an inverse relation. For any $v\in out(proc(n_k))$ at the $proc\_entry$ node $n_k$, the inverse data dependence relation $\mathbb{H}^{-1}[v]$ is obtained according to the following equation

$$\mathbb{H}_P^{-1}[v]=\{x\mid v\in\mathbb{H}_P[y], x\in rec(\mathbb{H}_P[y])\}$$

where $rec(\mathbb{H}_P[y])$ is defined as follows

$$rec(\mathbb{H}_P[y])=\begin{cases} y & \text{if } y\in in(proc(n_k)) \\ rec(\mathbb{H}_P[z]) & \text{if } y\notin in(proc(n_k)), y\in\mathbb{H}_P[z] \end{cases}$$

where $in(P)$ is the set of input arguments of procedure $P$.

**Theorem 1.** *If the SLV analysis of procedure $P$ with the slicing criterion $C$ generates the SLV set $S$ in the proc_entry node $n_k$ (i.e. $S = S_{entry}(n_k)$), then $S = \{w \mid w \in \mathbb{H}_P^{-1}[v], v \in C\}$*

*Proof.* "⇒": Suppose $x \in S$. $x$ can be included into $S$ due to (1) data, or (2) control dependences. Case (1): there exists a CFG node $[t = e]^n$ of $P$ where $x \in FV(e)$, there exists a path from $n_k$ to $n$ in the CFG of $P$ and there is no node in between $n_k$ and $n$ in such path which defines $x$. That means $t \in kill(n)$, $x \in gen(n)$ and $kill(n) \subseteq S_{exit}(n)$ for which $x$ is added to $S$ by the transfer function. Also $x \in in(P)$ as it is not defined before its first use. If $t \in C, t \in \mathbb{H}_P[x]$ and $x \in \mathbb{H}_P^{-1}[t]$ as $C$ contains only formal output variables. If $t \notin C$, suppose $t_1 \in \mathbb{H}_P[x]$ for some variable $t_1$. Then we have a sequence of updates of $\mathbb{H}_P$ on variables $t_1, t_2, \ldots, t_j = t$ at different iterations of the fixpoint computation such that $t_1 \in C$, and $t_i \in \mathbb{H}_P[t_{i+1}]$ for all $1 \leq i \leq j-1$. This implies that $x \in \mathbb{H}^{-1}[t_1]$. Case (2): there exists a path from $n_k$ to some conditional node $[c]^n$ such that $x \in FV(c(n))$ and $x$ is not defined by any node in this path. Otherwise it would not be the case that $x \in S$. Then according to equation (8), $t \in \mathbb{H}_P[x]$ for some $t \in out(P)$. According to the construction of $\mathbb{H}_P$ in equations (7 and 8), any $t \in out(P)$ can be included in $\mathbb{H}_P$ if $t \in S_{exit}(n')$ for some node $n'$. Since $C = S_{exit}(n'')$ where $n''$ is the $proc\_exit$ node of $P$, any $t \in out(P)$ and $t \in S_{exit}(n')$ is possible only if $t \in C = S_{exit}(n'')$ is propagated backward to $S_{exit}(n')$ by the iterations in Algorithm 1. So, $t \in C$, and $x \in \mathbb{H}_P^{-1}[t]$.
"⇐": Suppose $w \in \mathbb{H}_P^{-1}[v], v \in C$. This implies that $v \in out(P)$, $w \in in(P)$, and $v \in \mathbb{H}_P[w]$. Case (1): according to the construction of $\mathbb{H}_P$ in (7), there exists a node $n$ in the CFG of $P$ such that $kill(n) \subseteq S_{exit}(n)$, $w \in gen(n)$, and either (1) $v \in kill(n)$ or (2) $v \in \mathbb{H}_P[t]$ for some $t \in kill(n)$. In any case, $w$ is added to $S_{entry}(n)$ by the transfer function. Case (2): according to the construction of $\mathbb{H}_P$ in (8), there exists a conditional node $[c]^n$ such that $w \in FV(c(n))$ and hence $w \in S_{entry}(n)$. We can assume without loss of generality that there exists a path from $n_k$ to $n$ in the CFG of $P$. If $w \notin kill(n')$ or $kill(n') \not\subseteq S_{exit}(n')$ for any node $n'$ in the path from $n_k$ to $n$, $w \in S_{entry}(n)$ implies that $w \in S_{entry}(n_k) = S$. However, if $w \in kill(n')$ and $kill(n') \subseteq S_{exit}(n')$, we argue that node $n'$ does not exist with this condition. Let's assume that node $n'$ have the assignment $w = e$. For any $l \in FV(e)$[1], $w \notin \mathbb{H}_P[l]$ as $w \in in(P)$. So, according to equation 7, $\mathbb{H}_P[w]$ is copied into $\mathbb{H}_P[l]$, $v \in \mathbb{H}_P[w]$ implies $v \in \mathbb{H}_P[l]$, and then $\mathbb{H}_P[w]$ is reset to empty set. As $v \notin \mathbb{H}_P[w]$ anymore due to this reset, $v \notin \mathbb{H}_P[w]$ at the $proc\_entry$ node $n_k$ which is a contradiction. So, no such node $n'$ exists such that $w \in kill(n')$ and $kill(n') \subseteq S_{exit}(n')$. □

Consider the situation where we slice procedure $P$ first with slicing criterion $C$, and then with $C'$ such that $C' \subseteq C$. As noted above, in this situation $P$ does not need to be sliced again for $C'$ as the slice will be contained in the slice obtained for $C$. However, the SLV set $S$ for the $proc\_entry$ node of $P$ must still be generated. Assume that $\mathbb{H}_P^{-1}$ was computed when slicing $P$ with respect to $C$. According to Theorem 1, $S$ can then be computed directly from $\mathbb{H}_P^{-1}$ as $\{w \mid w \in \mathbb{H}_P^{-1}[v], v \in C'\}$. Thus, the SLV dataflow analysis need not be applied to $P$ anew.

Further improvements of efficiency in slicing are possible by delaying the slicing of any called procedure. Suppose a procedure $P$ receives multiple requests for slicing with the slicing criteria $C_1, \ldots, C_k$. Then instead of slicing $P$ $k$ times, it is possible to slice $P$ just once by combining all the slicing criteria $C = \bigcup_{1 \leq i \leq k} C_i$. This will yield the same slice as slicing with the different $C_i$. Let $\mathbb{H}^{-1}$ is generated when slicing with respect to $C$. Then, similarly, Theorem 1 allows us to compute the SLV sets for the respective $proc\_entry$ nodes for the slicing criteria $C_i$ directly from $\mathbb{H}^{-1}$. The handling of the worklist in Algorithm 1 can be tuned to create opportunities for this optimization.

[1] We are ignoring the case that $FV(e) = \emptyset$ for any input parameter $w$ before its first use as this is unusual and it will complicate the proof and the definition of $\mathbb{H}_P$, however it is doable.

**Example 4.2.** Consider the program in Fig. 2. Assume that we wish to slice it on the statement "print(s)". This statement corresponds to node 20 in the CFG and the slicing criterion is $\{s\}$. The CFG is traversed backwards, the sequence of visited nodes are $3, 10, 9, 8, 14, 19$, and the SLV sets are $S[3] = S[10] = S[9] = S[8] = \{s\}$, $S[14] = S[19] = \{g_2\}$ where $g_2$ is the ghost variable at the second call site of procedure *add*. At node 19, the slicing criterion for procedure *add* is basically empty as $g_2 \notin vars(add)$, and slicing this procedure can be stopped. The $proc\_call$ node at this call site is node 7 and $S[7] = \{s\}$ according to (6). Next, traversing the CFG, the generated SLV sets are $S[6] = S[5] = \{sp\}$, $S[12] = \{z, g_1\}$, $S[19] = \{z, g_1\}$ and the slice set is $N_{slice} = \{6, 3\}$ where $g_1$ corresponds to the first call site of procedure *add*. Node 6 is sliced due to data dependency and as it is control dependent on node 3, 3 is also sliced. During slicing the procedure *add*, the SLV sets are $S[18] = \{z, g_1\}$, $S[17] = \{t, g_1\}$, $S[16] = S[15] = \{x, y, g_1\}$, $\mathbb{H}_{add}[x] = \mathbb{H}_{add}[y] = \{z\}$, $\mathbb{H}_{add}^{-1}[z] = \{x, y\}$, and $N_{slice} = \{6, 3, 5, 12, 17, 16, 15, 19\}$. $g_1 \in S[15]$ implies that the next visited node will be 11. This procedure continues until a fixpoint is reached. Note that procedure *add* does not need to be sliced again, instead when the control reaches node 19 and $z \in S[19]$ implies that $S[15]$ will include $x$ and $y$ as $\mathbb{H}_{add}^{-1}[z] = \{x, y\}$. The fixpoint is reached in 2 iterations for this program and the sliced program contains all nodes except nodes 10, 18, and 21.

## 5. Slicing Low-level Code

The SLV-based slicing algorithm presented so far relies on a high-level model of memory. The data dependence analyses, whether separate or integrated with the slicing, are based on program variables that are distinct and non-overlapping: an assignment to a variable $x$ can thus not affect the value of any other variable $y \neq x$. Even if pointers to program variables are introduced, as long as this assumption holds it is well-known how to modify the data dependence analyses to cope with the possible aliasing that ensues.

However, for low-level code the memory model is different. Here a memory access is typically done to a numerical address, accessing a certain number of bits. Both the addresses and the sizes of accesses may vary dynamically (an example of the latter is block transfer instructions, where whole memory blocks are copied). Thus, it is not entirely straightforward to decide whether two memory accesses may overlap or not. Here we will sketch a way to analyze memory references to decide this. As the data flow analyses under consideration are may analyses, we are interested in approaches that can safely decide when accesses can not overlap: if they surely don't, then they cannot carry a data dependence. On the other hand, if there is a possible overlap then we will assume that there is a possible data dependence. This will yield a safe analysis where all data dependencies surely are included in the result.

### 5.1 The Memory Model

We will assume the memory model of the language ALF [17, 19], which is the language analyzed by the WCET analysis tool SWEET [1, 32]. ALF is designed to be able to faithfully represent both high- and low-level code, and its memory model,which is similar to the monolithic memory model [37], is chosen accordingly. Memory in ALF is organized into frames. Each frame is a separate memory area. An ALF address consists of a so-called "frameref",

which can be seen as a symbolic base pointer to the frame, and a numerical offset. This memory model is close to the one for unlinked code, where base addresses are not yet resolved. It is assumed that memory accesses to different frames never overlap, whereas accesses to the same frame may do. This memory model supports both the high- and low-level view. If a high-level language is translated to ALF, then distinct variables are preferably mapped to single, distinct frames with the same number of bits as the variables they represent. For low-level code larger memory areas might be mapped to frames: e.g., for executable binaries, the data memory may be modeled by a single frame. Memory accesses in ALF use an address as above, and a specified size. Thus an access can be represented by a triple $(f, o, s)$, where $f$ is a symbolic frameref, $o$ is an offset (non-negative, in bits) and $s$ is a size (same). The semantics is that the bits $o$ to $o + s - 1$ are accessed in frame $f$. Two memory accesses $(f, o, s)$ and $(f', o', s')$ will thus overlap iff $f = f'$, and $[o, o + s - 1] \cap [o', o' + s' - 1] \neq \emptyset$.

### 5.2 Abstract Domains

Data flow analysis on ALF code must be based on memory access triples rather than program variables. Also, to decide def-use chains, possible overlap of triples must be considered rather than program variables. Thus, the data flow analysis should be preceded by a value analysis that yields safe overapproximations to the memory access triples $(f, o, s)$. Such analyses are standard, and can be developed within the framework of abstract interpretation [12, 34]. The set of possible triples $T$ for a memory access in the code will then be approximated by an "abstract triple" $T^{\#}$ in an abstract domain, where $T^{\#}$ represents a set of triples that surely contains $T$.

The scenario described above is somewhat simplified. Since not only addresses depend on values, but also values may depend on addresses, it is in general not possible to first perform a value analysis and then an address analysis. Thus, the address and value analyses have to be combined into a joint analysis where approximated addresses and values are computed concurrently. What is needed is a combined abstract domain for these where the analysis can be carried out by a fixed-point iteration using standard methods.

If $\mathbf{F}$ is the set of framerefs in the program under analysis, and $\mathbf{Int}$ is the domain of integer intervals, then two possible abstract domains for memory access triples are $\mathcal{P}(\mathbf{F}) \times \mathbf{Int} \times \mathbf{Int}$ and $\mathcal{P}(\mathbf{F} \times \mathbf{Int} \times \mathbf{Int})$. The first domain represents set of triples by "abstract triples" $(F, I_o, I_s)$ where $F$ is a set of framerefs, and $I_o$ and $I_s$ are intervals surely containing the possible offsets and sizes, respectively. The second domain represents set of triples by finite set of abstract triples $\{(f_1, I_{o1}, I_{s1}), \cdots, (f_n, I_{on}, I_{sn})\}$ where $f_1, \cdots, f_n$ are frame-refs and $I_{ok}, I_{sk}$ are intervals surely containing the possible offsets and sizes for the accesses to frame $f_k$. The second domain allows for more precise representations than the first, but is also potentially more costly. Both abstract domains are standard within abstract interpretation, and it is well-known how to implement value analyses that compute abstract values in these domains for different memory accesses in a program.

### 5.3 Abstract Operations

If abstract values are computed for memory accesses by a value analysis, safe tests for overlaps will look as follows. With the domain $\mathcal{P}(\mathbf{F}) \times \mathbf{Int} \times \mathbf{Int}$, $(F, [l_o, u_o], [l_s, u_s])$ and $(F', [l'_o, u'_o], [l'_s, u'_s])$ represent possibly overlapping accesses if $F \cap F' \neq \emptyset$, and $[l_o, u_o + u_s - 1] \cap [l'_o, u'_o + u'_s - 1] \neq \emptyset$. For $\mathcal{P}(\mathbf{F} \times \mathbf{Int} \times \mathbf{Int})$, $\{(f_1, [l_{o1}, u_{o1}], [l_{s1}, u_{s1}]), \cdots, (f_n, [l_{om}, u_{om}], [l_{sm}, u_{sm}])\}$ and $\{(f'_1, [l'_{o1}, u'_{o1}], [l'_{s1}, u'_{s1}]), \cdots, (f'_n, [l'_{on}, u'_{on}], [l'_{sn}, u'_{sn}])\}$ represent possibly overlapping accesses if there exists $i, j$ such that $f_i = f'_j$ , and $[l_{oi}, u_{oi} + u_{si} - 1] \cap [l'_{oj}, u'_{oj} + u'_{sj} - 1] \neq \emptyset$. These overlap tests can be used when checking for possible data dependencies in the PDG-based slicing.

| if | $\mathbf{M_1} \setminus \mathbf{M_2}$ | $\mathbf{M_1} \cup \mathbf{M_2}$ | $\mathbf{M_1} \cap \mathbf{M_2}$ |
|---|---|---|---|
| $f_1 \neq f_2$ | $\{M_1\}$ | $\{M_1, M_2\}$ | $\emptyset$ |
| $(l_{o2} > sup_1 \vee l_{o1} > sup_2)$ $\wedge f_1 = f_2$ | $\{M_1\}$ | $\{M_1, M_2\}$ | $\emptyset$ |
| $l_{o1} \leq l_{o2} \leq sup_1 \wedge$ $sup_2 \geq sup_1 \wedge f_1 = f_2$ | $\{M_3\}$ | $\{M_4\}$ | $\{M_7\}$ |
| $l_{o1} \leq l_{o2} \leq sup_1 \wedge$ $sup_1 \geq sup_2 \wedge f_1 = f_2$ | $\{M_3, M_5\}$ | $\{M_4\}$ | $\{M_2\}$ |
| $l_{o2} \leq l_{o1} \leq sup_2 \wedge$ $sup_2 \leq sup_1 \wedge f_1 = f_2$ | $\{M_5\}$ | $\{M_6\}$ | $\{M_8\}$ |
| $l_{o2} \leq l_{o1} \leq sup_2 \wedge$ $sup_1 \leq sup_2 \wedge f_1 = f_2$ | $\emptyset$ | $\{M_6\}$ | $\{M_1\}$ |

Table 1: Abstract operations of two memory accesses $M_1 \equiv (f_1, [l_{o1}, u_{o1}], [l_{s1}, u_{s1}])$ and $M_2 \equiv (f_2, [l_{o2}, u_{o2}], [l_{s2}, u_{s2}])$.
Let
(i) $M_3 \equiv (f_1, [l_{o1}, l_{o1}], [l_{o2} - l_{o1}, l_{o2} - l_{o1}])$
(ii) $M_4 \equiv (f_1, [l_{o1}, max(sup_{max} - s_{max} + 1, l_{o1})], [s_{min}, s_{max}])$
(iii) $M_5 \equiv (f_1, [sup_2 + 1, sup_2 + 1], [sup_{max} - sup_2, sup_{max} - sup_2])$
(iv) $M_6 \equiv (f_1, [l_{o2}, max(sup_{max} - s_{max} + 1, l_{o2})], [s_{min}, s_{max}])$
(vi) $M_7 \equiv (f_1, [l_{o_{max}}, l_{o_{max}}], [sup_{min} - l_{o_{max}}, sup_{min} - l_{o_{max}}])$
(vii) $M_8 \equiv (f_1, [l_{o1}, l_{o1}], [sup_2 - l_{o1}, sup_2 - l_{o1}])$
where $sup_1 = u_{o1} + u_{s1} - 1$, $sup_2 = u_{o2} + u_{s2} - 1$, $l_{o_{max}} = max(l_{o1}, l_{o2})$ $sup_{min} = min(sup_1, sup_2)$, $sup_{max} = max(sup_1, sup_2)$, $s_{min} = min(l_{s1}, l_{s2})$, and $s_{max} = max(u_{s1}, u_{s2})$

Set operations for the abstract domains above can be built on top of the operations on $\mathbf{F} \times \mathbf{Int} \times \mathbf{Int}$ that are defined in Table 1. For instance, if $S_1, S_2 \in \mathcal{P}(\mathbf{F} \times \mathbf{Int} \times \mathbf{Int})$ then $S_1 \setminus S_2 = \bigcup_{M_2 \in S_2} \bigcup_{M_1 \in S_1} M_1 \setminus M_2$. Other set operations can be computed accordingly. However, for $\mathcal{P}(\mathbf{F}) \times \mathbf{Int} \times \mathbf{Int}$, the conditions $f_1 = f_2$ and $f_1 \neq f_2$ should be replaced by $F_1 \cap F_2 \neq \emptyset$ and $F_1 \cap F_2 = \emptyset$, respectively, in the definitions in Table 1.

### 5.4 Producing Executable Slices

Low-level code often contains *jump* or *goto* statements. The SLV-based slicing algorithm introduced in the previous sections does not include these statements in the slice, as it is based on a CFG model where the nodes are conditions and assignments, and jumps are implicitly represented as edges. Also if explicit *goto* nodes are present in the CFG they will typically not be included by a data-dependence based slicing, since they do not perform any reads or writes.

If *gotos* are not properly included, then that might change the semantics of the slice. Informally the semantics of the slice should be such that the same values are computed for the variables in the slicing criterion, at the respective program points, as in the original program. So, for instance, if the slicing criterion is within a loop then the loop structure will typically have to be preserved even if most of the loop is sliced away.

An example is shown in Fig. 4a, where the slicing criterion is the variable $x$ at `L3` and the boxed statements constitute the slice. In this example, the slicing criterion is located within a loop where `x` is incremented for each iteration. In the computed slice the loop structure is removed, and `x` will only be incremented once. This is clearly not the original semantics for the slicing criterion. To restore this semantics, the statements `goto L1` and `goto L3` must be included to form the loop structure anew.

Fig. 4b gives another example, with slicing criterion `x` at `L2`. Here the order of executing the statements can be garbled if the proper *goto* statements are not preserved in the slice. In the original program `x` will be incremented if `c` is true and decremented if `c` is false, but if no *gotos* are included then `x` will always be incremented.

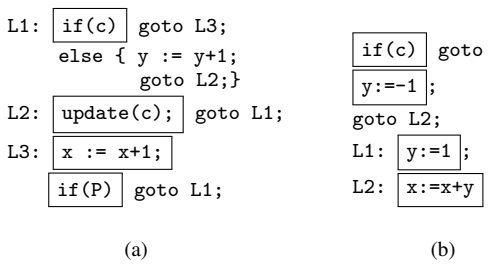


Figure 4: Illustrating the importance of preserving (a) loop structure, and (b) flow order in the sliced program

Thus, in order to preserve the loop structure and the flow order in the sliced program, it is important to include the relevant *goto* statements and (when relevant) their associated control predicates. Also, at the same time we would like to keep the sliced program minimal. In particular, our assumption is that it is sufficient to include the following nodes from the CFG of the given program into the sliced program that preserves the loop structure and flow order:

- We include all *strongly live goto* nodes according to the definition 5.3 below
- We include all control structures that become live due to strongly live goto nodes.

In order to define *strongly live goto* nodes, we borrow the concept of *lexical successor tree*.

**Definition 5.1** (Lexical Successor Tree [2])**.** *Two statements $s$ and $s'$ in the program code constitute a parent node and its immediate successor node respectively in the lexical successor tree if during the execution of the program, whenever control reaches $s'$, then replacing $s'$ by skip yields a program flow to $s$.*

Note that in the presence of unstructured code, the CFG representation loses the information of whether two statements in the program code are consecutive or not. We can retrieve such information by using the lexical successor tree which can be constructed in a purely syntax directed manner [2]. Given a set of nodes $S$ in the lexical successor tree $T$, two nodes $n_1, n_k$ are immediate successors with respect to $S$ if there is a path $n_1, n_2, \cdots, n_k$ in $T$ and $S \cap \{n_2, \cdots, n_{k-1}\} = \emptyset$, and we say that $(n_1, n_k) \in T_S$. If $(n_1, n_k) \notin T_S$, we say that $n_1, n_k$ are non-lexically adjacent.

Given the CFG $(N, Flow)$ and the set of nodes $S \subseteq N$ that does not include any goto nodes, an edge is considered live according to the following definition.

**Definition 5.2** (Live Edges [21])**.** *Given a set of nodes $S$ in the CFG $(N, Flow)$, an edge from a branch node $b \in N$ to its non-syntactic successor (e.g. the first statement in the else block of an if-then-else) is live only if $b \in S$. All other CFG edges are always live.*

A goto node $g$ is considered strongly live with respect to the given set $S$ according to the following definition.

**Definition 5.3** (Strongly Live Goto Node)**.** *A goto node $g \in N$ becomes live with respect to $S$, if (1) there exists a node in $S$ that is reachable from $g$ by traversing live edges containing at least one non-lexically adjacent edge, and (2a) $g$ is reachable from any node in $S$ by traversing live edges or (2b) $g$ is the descendant of a node reachable via live edges that does not have any lexical predecessor.*

The PDG based slicing algorithm can be adapted according to the 'Strong, syntax-preserving, Ottenstein-more" slicing algorithm in [21] in order to preserve the termination behaviour and flow-order. Our SLV-based slicing Algorithm 1 can be extended to produce similar slices as follows:

1. First, two boolean variables $Live[n]$ and $NonLex[n]$ is maintained for each node $n$ in the CFG. These two variables represent whether the node $n$ is reachable and non-lexically adjacent by at least one edge respectively from any node in $S$ by traversing backward live edges. Initially, $Live[n] = yes$ for all $n \in N_{slice}$ and $no$ otherwise, and $NonLex[n] = no$ for all $n$ in the CFG. An auxiliary workset $W'$ is also maintained which is initially empty and the iteration of algorithm 1 terminates when $W \cup W' = \emptyset$. A *lexical successor tree* $T$ is maintained that contains all the successor relations.
2. Next, for each node $n$ in the CFG, a set $G[n]$ is maintained which contains the potential goto nodes to be included in the slice if $n$ is included in the slice as it meets the condition (2a) of Definition 5.3. Initially all $G[n]$ are empty.
3. During the iteration of the algorithm, for any $(n, n', CFG) \in W$, $Live[n']$ is updated according to the following equation:

$$Live[n'] = \begin{cases} yes & \text{if } Live[n] = yes \ \wedge \ (n, n') \in F \\ & \text{is live w.r.t } N_{slice} \\ no & \text{otherwise} \end{cases}$$

   $NonLex[n']$ is updated as follows:

$$NonLex[n'] = \begin{cases} yes & \text{if } (NonLex[n] = yes \vee (n', n) \\ & \notin T_{N_{slice}}) \wedge n' \notin N_{slice} \\ no & otherwise \end{cases}$$

4. For any $(n, n', CFG) \in W$, if $Live[n'] \wedge NonLex[n'] = yes$ and $n'$ is a *goto* node, it is a potential *goto* node to be included in the slice as it meets the condition (1) of definition 5.3. So, we update $G[n'] = G[n'] \cup G[n] \cup \{n\}$. If $n'$ is not a *control-predicate* and $(n, n') \in F$ is live w.r.t $N_{slice}$, $G[n'] = G[n'] \cup G[n]$; $G[n']$ is unchanged in all other cases. If $Live[n'] \wedge NonLex[n'] = no$, $W'$ is updated by $W' = W' \cup \{(n, n', CFG)\}$.
5. During the iteration of Algorithm 1, if $n \in N_{slice}$ and $G[n]$ is not empty, add all the goto nodes in $G[n]$ to the slice as it meets condition (2a) of definition 5.3. Any conditional node $n$ is also included in the slice if $G[n]$ is not empty and there exists $n' \in G[n]$ which is already included in the slice as the conditional structure controls a live goto node included in the slice.
6. At the end of the iteration, check for any node $n$ such that $G[n]$ is not empty and it does not have any lexical predecessor node. Add $n$ and all $n' \in G[n]$ into the slice (as condition (2b) of definition on 5.3 is satisfied).

In order to prove the correctness (as defined in [29]) of the above sketch, it needs to be proved that (i) Definition 5.3 of strongly live goto nodes has *semantic effect* (Definition 0 in [29]) on $N_{slice}$, and (ii) any goto node other than the strongly live goto nodes in the program does not have any semantic effect on $N_{slice}$. Intuitively, any strongly live goto node with respect to $N_{slice}$ causes program flow to a node in $N_{slice}$ and thus have a semantic effect on $N_{slice}$ if it is removed from the sliced program. Furthermore, removing lexically adjacent strongly live goto nodes keep $N_{slice}$ minimal. We leave the proof of correctness of the above sketch as future work.

## 6. Experimental Evaluation

We have implemented the the SLV-based and the PDG-based slicing in SWEET, and run them on a number of benchmark programs

| | | $P_{BV}$ | | $P_{PDG}$ | | $P_{SLV}$ | |
|---|---|---|---|---|---|---|---|
| **Benchmarks** | $N_{CFG}$ | $N_{BV}$ | $T_{BV}$ | $N_{PDG}$ | $T_{PDG}$ | $N_{SLV}$ | $T_{SLV}$ |
| loop | 39 | 26 | 2 | 26 | 67 | 26 | 35 |
| expint* | 139 | 59 | 0 | 51 | 1 | 51 | 1 |
| cnt | 145 | 55 | 0 | 21 | 0 | 21 | 0 |
| bmp | 161 | 105 | 3 | 40 | 4 | 40 | 4 |
| edn* | 639 | 455 | 1 | 56 | 4 | 56 | 3 |
| edn2 | 1391 | 1300 | 19 | 24 | 39 | 24 | 14 |
| fir | 1569 | 800 | 1 | 48 | 6 | 48 | 5 |
| nsichneu | 1860 | 5 | 6 | 4 | 19 | 4 | 4 |
| esab_mod | 2349 | 817 | 2 | 802 | 57 | 802 | 54 |
| bmp2 | 3725 | 3273 | 348 | 41 | 306 | 41 | 16 |
| arrayloop | 12044 | 12029 | 14 | 33 | 33 | 33 | 27 |

Table 2: Comparison among three slicing algorithms. $N$ represents number of nodes, and $T$ represents time measured in seconds. Examples marked with $*$ are generated from ARM7 binaries.

obtained mostly from the Mälardalen WCET benchmark suite [18]. The original benchmark programs are written in C, and have been transformed into ALF for subsequent analysis by SWEET. In order to evaluate the effectiveness of our analysis for low-level code we have first compiled some benchmarks into ARM7 binaries (obtained by the GCC ARM7 compiler version 4.6.1), and translated the resulting binaries into ALF. Other benchmarks have been manipulated in order to make the memory model more low-level, by putting several variables into the same struct or array. Since the ALF translator that we have used then generates a single frame for them, the data dependence analysis will have to decide offsets and sizes of memory accesses to resolve the dependencies as described in Section 5.3. All experiments have been performed on a 1.7 GHz Intel Core i7 processor with 8GB RAM.

The current implementation of the SLV-based slicing algorithm is interprocedural, and several of the benchmark codes contain function calls. However, none of the optimization described in Section 4.1 are implemented in the current version. The data dependency analysis is based on abstract operations over the interval domain as described in Section 5. So, each element in the SLV sets as well as the $gen$ and $kill$ sets contains the memory access triples $(f, o, s)$ where $o$ and $s$ are intervals. Two versions of the PDG-based slicing have been implemented in SWEET. They differ in the RD analysis for computing the data dependencies: both use bit vectors to represent the RD sets, but the first implementation only distinguishes frames whereas the second also considers offsets and possible overlaps within frames as described in Section 5.3. The first version is thus less precise but faster, whereas the second should compute the same slices as our SLV-based algorithm. The control dependencies are computed using post-dominators. All the analyses use Steensgard's points-to analysis [41] for checking possible memory aliasing between frames. No further optimizations are implemented in any of the slicing algorithms, and none of them implement the generation of executable code (the computed slices are sets of nodes in the CFG).

Table 2 compares the results of the three slicing algorithms: the PDG-based slicing algorithm that operates on frame level, $P_{BV}$, the PDG-based slicing that also considers offsets and possible overlaps within frames, $P_{PDG}$, and the SLV-based slicing algorithm, $P_{SLV}$. The slicing criteria have in all runs been either the loop conditions or the conditional statements. These are typically the slicing criteria used when slicing is performed in the context of WCET analysis [13, 40], which is our primary client application of slicing. Note that the optimizations described in Section 3.3 have not been applied even though we used loop conditions or conditional statements as slicing criterion. So, our experimental results should not be biased to this kind of slicing criterion.

The algorithms have been run on the example programs several times, with the same slicing criteria, and the running times have been averaged and rounded afterwards. $N_{CFG}$ is the number of nodes in the CFG of the analysed ALF program. $N_{BV}$, $N_{PDG}$ and $N_{SLV}$ stand for the number of nodes in the sliced programs obtained from the $P_{BV}$, $P_{PDG}$, and $P_{SLV}$ slicing algorithms respectively. $T_{BV}$, $T_{PDG}$, and $T_{SLV}$ are the running times, measured in seconds, of the corresponding algorithms. Note that there is a small variation in the number of nodes in the slices obtained for $P_{SLV}$ and $P_{PDG}$. This is due to the fact that the slices computed by $P_{SLV}$ consist of CFG nodes whereas those computed by $P_{PDG}$ consist of PDG nodes. The call site nodes in the CFG and PDG in SWEET have slightly different representations, which yields the difference. When this discrepancy is compensated for, the number of nodes in the slices are the same for the two algorithms for all benchmarks and the slices indeed represent the same code in all cases, as expected. It is the "cleaned" number of nodes that is given in Table 2.

It can be observed from the above table that $N_{BV} \geq N_{PDG}$ and $N_{BV} \geq N_{SLV}$ for all benchmark programs. This illustrates that $P_{BV}$ is much less precise than $P_{SLV}$ for slicing the low-level code. Sometimes, this loss of precision by the $P_{BV}$ algorithm is very significant compared to the other two algorithms. For example, in the *edn2* benchmark, the number of nodes in the slice computed by $P_{BV}$ is 1391 compared to 24 nodes by $P_{SLV}$. This result is obvious as $P_{BV}$ is loosing a lot of precision in computing the SDG. This is due to the fact that if a statement writes in a frame and another statement reads from the same frame but from another location, $P_{BV}$ considers that they are data dependent which is not necessarily the case.

Regarding the execution time of the slicing algorithms, $T_{SLV} \leq T_{PDG}$ in all examples. Thus, $P_{SLV}$ is faster than $P_{PDG}$ for computing a single slice for these benchmark codes. This is because the construction of the SDG in $P_{PDG}$ is slower compared to running a single SLV analysis. For small examples, the timing difference is not significant. However, for large examples like *bmp2* the difference in execution time is large: for instance, $P_{SLV}$ takes 16 seconds whereas $P_{PDG}$ takes approximately 5 minutes for the *bmp2* program. This indicates that $P_{SLV}$ scales better than $P_{PDG}$ when computing single slices.

$P_{BV}$ is faster than $P_{PDG}$ or $P_{SLV}$ in almost all examples as it does not require the value analysis to resolve data dependencies within frames. But this speed advantage comes at the cost of seriously deteriorated precision in the computed slices.

## 7. Related Work

Since the seminal work of Weiser [44, 46], there has been much work on different kinds of program slicing. Different approaches to slicing include static [35, 44, 46], quasi-static [43], conditioned [10], dynamic [28], and amorphous [20] slicing, of which static slicing techniques are the ones that are most comparable to our approaches. Solving the dataflow equations, the reachability problem in a dependence graph, and using information-flow relations are the most dominant static slicing approaches found in the literature. Our SLV-based slicing technique is similar to the approach of Weiser [22, 44, 46] and Lyle [33]. The computational complexity of Weiser's approach for the intraprocedural slicing is $O(v \times |(N + Flow)|)$ [42] which is higher than our intraprocedural SLV-based slicing algorithm. Our contemporary work [27] also uses the SLV analysis to do slicing on the fly, but it is based on a different way to solve the data flow equations that works on well-structured intraprocedural code.

Ottenstein and Ottenstein [35] proposed that program slicing can be viewed as a reachability problem on the program dependence graph. Following this work, reachability-based slicing techniques that work on PDGs or variants of them have been used by many researchers [6, 23–25, 39]. The emphasis of these works have been on properties such as precision, complexity, applicability, and scalability. All these approaches compute all the dependencies of the given program before the slicing, which is good when the same code is sliced many times (since the cost for computing the dependencies then can be amortized over the slices). There have been several empirical studies and survey papers [7, 8, 42, 48] that compare the reachability-based slicing techniques. Unstructured program slicing was considered by Ball and Horwitz [4], Choi and Ferrante [11], Agrawal [2], Kumar and Horwitz [29]: all these techniques are based on the PDG or its variants. The approaches of Lyle [33], Gallagher and Lyle [15], and Jiang et al. [26] are based on solving dataflow equations, and the solutions are either conservative or incorrect. We are not aware of many efforts for slicing low-level code except CodeSurfer/x86 [3]. CodeSurfer/x86 implements the PDG-based slicing on x86 executables, and data dependencies among memory accesses are computed using a value-set analysis ("VSA") which is based on congruences and intervals. Our approach to slicing of low-level code uses a simpler value analysis (intervals) but attempts to use information about the ALF frames to increase the precision. This is somewhat comparable to using debug information from a compiler.

## 8. Conclusion and Future Work

Program slicing extracts the program parts that can possibly affect certain slicing criteria. In this paper, we have developed a lightweight interprocedural slicing technique that performs better when few slices of a given program are required. We perform slicing on-the-fly during a variant of the SLV data flow analysis on the CFG representation, and thus we avoid computing an explicit data dependence graph. Our slicing technique handles low-level code using abstract interpretation based value analysis, and we also show how it can be extended to produce executable slices. Comparing our approach with PDG based slicing, we obtain exactly the same accuracy as in PDG-based slicing on a set of benchmark codes and our algorithm is at least as fast, and sometimes much faster, than the PDG-based algorithm on this set of benchmark codes. Our results indicate that our algorithm scales better since we obtain the largest speedups for larger benchmark codes.

Future works include implementing all the features of SLV-based slicing that have been described here, and make an empirical evaluation in order to compare our approach with other slicing approaches over a set of larger programs. We would like to know about the program patterns for which the SLV-based slicing performs better. For example, when slicing the same code many times, with different slicing criteria then we would like to know the break-even point where SLV-based slicing gets worse than PDG-based slicing. We would also like to implement various optimization techniques in order to improve the performance of SLV-based slicing. For example, the performance of the slicing can most likely be improved by a judicious choice of iteration order [34].

## Acknowledgments

The research presented in this paper is supported by the European Commission through the Marie Curie IAPP 251413 APARTS project, by the Swedish Foundation for Strategic Research under the SYNOPSIS project, and by the Knowledge Foundation through the TOCSYC project. We would like to thank Linus Källberg for his help with the SWEET WCET tool, Niklas Holsti for providing us the ALF code generated from ARM7 binaries, and the anonymous reviewers for their comments that helped us to improve the paper.

## References

[1] SWEET home page, 2011. URL `http://www.mrtc.mdh.se/projects/wcet/sweet/`.

[2] H. Agrawal. On slicing programs with jump statements. In *Proc. ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 302–312, New York, NY, USA, 1994. ACM.

[3] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, Aug. 2010.

[4] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proc. First International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pages 206–222, London, UK, UK, 1993. Springer-Verlag.

[5] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 384–396, New York, NY, USA, 1993. ACM.

[6] D. Binkley. Precise executable interprocedural slices. *ACM Lett. Program. Lang. Syst.*, 2(1-4):31–45, Mar. 1993.

[7] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proc. International Conference on Software Maintenance*, ICSM '03, pages 44–, Washington, DC, USA, 2003. IEEE Computer Society.

[8] D. Binkley and M. Harman. A survey of empirical results on program slicing. In *Advances in Computers*, volume 62 of *Advances in Computers*, pages 105 – 178. Elsevier, 2004.

[9] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1): 3–35, Jan. 1995.

[10] G. Canfora. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, Dec. 1998.

[11] J.-D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, July 1994.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, proc. POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[13] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proc. Seventh International Workshop on Worst-Case Execution Time Analysis, (WCET2007)*, July 2007.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[15] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, Aug. 1991.

[16] R. Gupta, M. Jean, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proc. Conference on Software Maintenance*, pages 299–308. IEEE Computer Society Press, 1992.

[17] J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. ALF - a language for WCET flow analysis. In N. Holsti, editor, *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET09)*. OCG, June 2009.

[18] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks - past, present and future. In *Proc. 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010. URL `http://www.es.mdh.se/publications/1895-`.

[19] J. Gustafsson, A. Ermedahl, and B. Lisper. ALF (ARTIST2 language for flow analysis) specification. Technical report, Oct. 2011. URL `http://www.es.mdh.se/publications/1138-`.

[20] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. In *Software Focus*, pages 70–79. IEEE Computer Society Press, 1997.

[21] M. Harman, A. Lakhotia, and D. Binkley. Theory and algorithms for slicing unstructured programs. *Information and Software Technology*, (7):549–565, 2006.

[22] H. R. H.K.N. Leung. Comments on program slicing. *IEEE Transactions on Software Engineering*, SE-13, 12:1370–1371, 1987.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[24] C. R. C. J. C. Hawang, M. W. Du. Finding program slices for recursive procedures. In *Proc. 12th Annual International Computer Software and Applications Conference*, COMPSAC '88, pages 220–227, Chicago, 1988.

[25] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *Proc. 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 2–10, New York, NY, USA, 1994. ACM.

[26] J. Jiang, X. Zhou, and D. J. Robson. Program slicing for C — the problems in implementation. In *Proceedings, Conference on Software Maintenance 1991*, pages 182–190, Sorrento, Italy, Oct. 15–17, 1991. IEEE Computer Society Press.

[27] H. Khanfar. Objects-based slicing. Technical Report MDH-MRTC-284/2014-1-SE, School of Innovation, Design and Engineering, Mälardalen University, May 2014.

[28] B. Korel. Dynamic program slicing. In *Information Processing Letters, 29 Oct*, 1988.

[29] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Proc. 5th International Conference on Fundamental Approaches to Software Engineering*, FASE '02, pages 96–112, London, UK, UK, 2002. Springer-Verlag.

[30] J. R. Larus, S. Ch, and R. Y. Using tracing and dynamic slicing to tune compilers. Technical report, University of Wisconsin Computer Sciences Department, 1993.

[31] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In J. Gustafsson, editor, *Proc. Third International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 77–80, July 2003.

[32] B. Lisper. SWEET – a tool for WCET flow analysis. In B. Steffen, editor, *Proc. $6^{th}$ International Symposium on Leveraging Applications of Formal Methods (ISOLA'14)*, lncs, pages 482–485, Korfu, Greece, oct 2014. Springer-Verlag.

[33] J. R. Lyle. *Evaluating Variations on Program Slicing for Debugging (Data-flow, Ada)*. PhD thesis, College Park, MD, USA, 1984.

[34] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[35] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes*, 9(3):177–184, Apr. 1984.

[36] R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138, New York, NY, USA, 1959. ACM.

[37] Z. Rakamarić and A. J. Hu. A scalable memory model for low-level code. In N. D. Jones and M. Müller-Olm, editors, *Proc. 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 290–304. Springer, Jan. 2009.

[38] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 11–20, New York, NY, USA, 1994. ACM.

[39] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Proc. POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[40] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET flow analysis by program slicing. In *Proc. ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES2006)*. ACM, June 2006.

[41] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[42] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[43] G. A. Venkatesh. The semantic approach to program slicing. In *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 107–119, New York, NY, USA, 1991. ACM.

[44] M. Weiser. Program slicing. In *Proc. 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[45] M. Weiser. Reconstructing sequential behavior from parallel behavior projections. *Inf. Process. Lett.*, 17(3):129–135, 1983.

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

[47] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3): 1–53, Apr. 2008.

[48] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.