Static Backward Program Slicing for Safety-Critical Systems

Husni Khanfar(✉), Björn Lisper, and Abu Naser Masud

School of Innovation, Design, and Engineering,
Mälardalen University, SE-721 23 Västerås, Sweden
{husni.khanfar,bjorn.lisper,masud.abunaser}@mdh.se

Abstract. *Static program slicing* is a technique to detect the program parts (i.e. the "slice") of the given program possibly affecting a given property. The technique is of interest for analysing safety-critical software, since it can identify the program parts that may affect various safety properties. Verification efforts can then be directed towards those parts, leading to a more efficient verification process.

We have developed a novel method for static backward program slicing. The method works for well-structured programs, as commonly demanded by coding standards for safety-critical software. It utilises the program structure to obtain a highly efficient slicing process, where control dependencies are inferred from the program structure, and the slicing is done on-the-fly concurrently with the data dependence analysis.

We have evaluated our method experimentally. For applications that require few slices to be taken, like checking for a set of safety properties, we obtain large speedups as compared with the standard method for static backward program slicing. We have also investigated how the speedup varies with various parameters such as code size, size of the slice relative to the full program, and relative frequency of conditions in the code.

Keywords: Program slicing · Dataflow analysis · Strongly live variable · Program dependency graph

1 Introduction

Program slicing refers to a collection of techniques to find the parts of the given program (so-called "slices") that can affect, or be affected by, a certain property [23]. The property is often abstracted into a *slicing criterion*, which typically consists of some program variables in some specific program points. Program slicing was first introduced by Weiser [24] in the context of debugging and parallel processing. It has since been applied in areas like software testing, software measurement, program comprehension, program integration, and software maintenance.

A particular slicing technique is *static backward program slicing*. For a given slicing criterion, backwards slicing computes a slice consisting of all the statements, conditions, and inputs to the program that can possibly affect the values

J.A. de la Puente and T. Vardanega (Eds.): Ada-Europe 2015, LNCS 9111, pp. 50–65, 2015.
DOI:10.1007/978-3-319-19584-1_4

of the variables in the slicing criterion in the respective program points. These values can be affected in two ways: either through *data dependencies*, where data produced by some statement are being read by some other statement, or through *control dependencies* where a condition may affect the possible execution of a statement. Fig. 1 shows a piece of code, and its backward slice (in boldface) with respect to the slicing criterion $\{z\}$ located at the last line.

```
p();
read(x); read(y); read(z);
z = y;
u = x + z;
if (x > 0) then x = 5 else z = 9;
y = x/z;
```

Fig. 1. Backward slicing example

Backward program slicing can be used to identify the program parts that can possibly affect value-related safety properties, like, for instance, whether memory accesses may be out of bounds. In the example in Fig. 1, there may be a possible division by zero at the last line. The corresponding safety property can be abstracted into the aforementioned slicing criterion $\{z\}$. Thus, the slice computed w.r.t. this criterion is the part of the program that can possibly affect whether a division with zero will occur or not. From this slice we can deduce that the formation of test data to test for division of zero should focus on the inputs for x and y, since the read statement for z and the initial value of u are not included in the slice and thus cannot possibly influence the value of z at the last line.

Backward slicing can also be used to find opportunities for early testing before the software is complete. This can be valuable, since it helps catching bugs early. In the example, the call to the procedure p is not included in the slice. This means that this call cannot influence whether or not a division by zero occurs, regardless of the code for p. Consequently we can replace p with a stub, and test for division by zero before the actual code for p is written.

The standard method for static backward slicing is based on the *Program Dependence Graph* (PDG) [9,20]. The nodes of the PDG are the nodes in the control flow graph (CFG) of the program, and it is the union of the *control dependence graph* (CDG) and the *data dependence graph* (DDG). The CDG is computed from the CFG by a control dependence analysis computing the strongest post-dominators for the conditions in the CFG. The DDG is computed using some data flow analysis, typically Reaching Definitions [19], from which the def-use pairs that constitute the edges in the DDG can be built. Once the PDG is built, slicing w.r.t. some slicing criterion can be done by a simple backwards traversal of the PDG from the criterion. The PDG-based slicing is intra-procedural. It can be extended to the inter-procedural case by instead considering the *system dependence graph* (SDG) [22].

A potential problem with PDG-based slicing is that the whole PDG has to be built before the actual slicing is performed. This can be wasteful especially in situations where the resulting slice is small relative to the full program, and it may also cause problems with scalability. We have developed such a method for static backwards slicing that avoids this problem. Our experiments show that it performs considerably better than the PDG-based method when few slices are computed, and it is also more space efficient. Our method computes slices concurrently with the dependence analysis, avoiding the construction of any dependence- or flow graphs. It works for well-structured, jump-free code: such code is prescribed by coding standards for safety-critical applications such as MISRA-C and SPARC 2014.

The rest of the paper is organized as follows. Section 2 reviews some background that is needed for the reading of the rest of the paper. In Section 3 we give the distinguishing features of our algorithm, and in Section 4 we define the underlying program representation that it uses. Section 5 explains more in detail how the algorithm works. In Section 7 we give an account for our experimental evaluation, comparing our approach with the PDG-based algorithm. Section 8 gives an account for related work, and Section 9 concludes the paper.

2 Preliminaries

2.1 A Model Language

Our work considers well-structured programs. Our algorithm is defined for a minimal, well-structured model language with the following abstract syntax:

$$s \quad ::= \quad x := a \quad | \quad skip \quad | \quad s'; s'' \quad | \quad \texttt{if}\ c\ \texttt{then}\ s'\ \texttt{else}\ s'' \quad | \quad \texttt{while}\ b\ \texttt{do}\ s'$$

We assume that all variables are numerical, and that arithmetic expressions a and conditions c have no side-effects.

Adding features like procedures, pointers, non-numerical data types, structured types such as arrays and records, and procedures, is straightforward. Our algorithm is interprocedural, and handles procedures with input and output parameters (see Section 6). It can be extended to deal with the other features as well.

2.2 Data Flow Analysis

Data flow analysis is used to compute data dependencies. We give a short account for it here, as our algorithm uses a particular data flow analysis. Details can be found in, e.g., [19].

Data flow analysis is often defined over a flowchart representation of the program; a flowchart is essentially a CFG where the nodes are individual statements, or conditions, rather than basic blocks. (For our model language in Section 2.1 the nodes are assignments, conditions, or *skip* statements.) The edges in the flowchart are the program points: a data flow analysis computes a set of data

flow items for each program point. These data flow items vary with the analysis, but they typically represent some kind of data flows in the program.

A data flow analysis assigns a monotone *transfer function* f_n to each flowchart node n, which is used in an equation relating the sets $S_{entry}(n)$ and $S_{exit}(n)$ for ingoing and outgoing edge of the node, respectively. (For simplicity we neglect that some nodes may have more than one ingoing or outgoing edge.) For a *forward* analysis the equation has the form $S_{exit}(n) = f_n(S_{entry}(n))$, and for a *backward* analysis it has the form $S_{entry}(n) = f_n(S_{exit}(n))$. In both cases we obtain a system of equations for the sets. This system is solved by a standard fixed-point iteration. The resulting sets provide the result of the analysis.

Besides being forward or backward, data flow analyses are also classified as *may* or *must* analyses. We will only be concerned with may analyses here: these compute data flows that will never underapproximate the real data flows.

PDG-based slicing uses the Reaching Definitions (RD) analysis [19]. It is a forward may analysis that computes sets of pairs (x, n), where x is a program variable and n is a CFG node. If (x, n) belongs to the set associated with the edge p, then the value of x assigned at n may still be present in x when at p and thus, if x is possibly used at the node n' immediately succeeding p, there is a possible data flow from n to n'.

Our slicing algorithm uses another data flow analysis known as *Strongly Live Variables* (SLV) [19]. It is a backward may analysis: given some "uses" of some variables in some program points (basically a slicing criterion) it traces backwards the variables that may influence the values of the used variables. Thus it computes in each program point a set of "Strongly Live Variables", whose values in the respective program point may affect the values of the used variables. The SLV sets also represent data dependencies: if x belongs to a SLV set, and the preceding node possibly assigns x, then there may be a data flow from that node to some of the variable uses. For SLV analysis the transfer functions take the following form:

$$\begin{array}{l} S_{entry}(n) = (S_{exit}(n) \setminus kill(n)) \cup gen(n), \text{ if } kill(n) \subseteq S_{exit}(n) \\ S_{entry}(n) = S_{exit}(n), \text{ otherwise} \end{array} \tag{1}$$

where, $S_{entry}(n)$ and $S_{exit}(n)$ represents respectively the SLV set that is present before and after the CFG node n. In the original formulation of SLV analysis [19], the *kill*, and *gen* sets are defined as follows for assignments $x := a$, conditions c, and *skip* statements:

$$\begin{array}{lll} kill(x := a) = \{x\} & kill(c) = \emptyset & kill(skip) = \emptyset \\ gen(x := a) = FV(a) & gen(c) = FV(c) & gen(skip) = \emptyset \end{array} \tag{2}$$

Here, $FV(a)$ denotes the set of program variables that appear in the expression a. The above definitions are according to the original formulation of SLV analysis where it is assumed that variables in the conditions are always used. The definitions can easily be modified to deal with variable uses corresponding to any possible slicing criterion in arbitrary program points [15].

3 An Overview of the Slicing Algorithm

Our algorithm has the following distinguishing features:

- It uses an internal representation of interconnected code blocks, which basically provides a representation of the syntax tree. It can be generated from the syntax tree in time linear in the size of the tree. No CFG is needed.
- For well-structured code the control dependencies are found directly from the syntax. For our model language, the control dependencies are exactly from a branch condition to the statements in the branches of a conditional statement, and from a loop condition to the statements in its loop body. This information is retained in the code blocks: thus, no further control dependence analysis is necessary.
- The data dependence analysis uses SLV rather than RD. If the SLV set succeeding an assignment $x := a$ contains x then we know that some part of the slicing criterion will be dependent on the value of x produced there, and thus $x := a$ can be immediately put into the slice already during the data flow analysis. No DDG has to be built.
- Rather than maintaining a set of SLVs in each program point, and performing a conventional fixed-point iteration where the sets grow until a fixed-point is reached, the algorithm maintains a single set of SLVs for each code block and keeps track of how far each variable has propagated within the block. This reduces the memory requirements, and it can also sometimes eliminate unnecessary iterations through statements that will not be included in the slice anyway.

Thus the algorithm performs the slicing in a dynamic fashion, without the need to build any control flow or dependence graphs.

4 Predicated Code Blocks

We use *Predicated Code Blocks* (PCB) to represent the control flow in the program. A predicated code block, $(s_0, \ldots, s_n)$, consists of a conditional predicate s_0 followed by a sequence of statements $s_1, \ldots, s_n$. Conditional statements, while loops, and procedure bodies are represented by such code blocks, and a set of interconnected blocks represents the whole program. The code blocks are derived for these kinds of statements as follows:

$$\begin{aligned}\texttt{if } c \texttt{ then } s_1;\ldots;s_m \texttt{ else } s'_1;\ldots;s'_n &\rightarrow \{(c, s_1, \ldots, s_m), (\neg c, s'_1, \ldots, s'_n)\}\\ \texttt{while } c \texttt{ do } s_1;\ldots;s_n &\rightarrow \{(c, s_1, \ldots, s_n)\}\end{aligned}$$

A conditional statement is represented by two PCBs, one for each branch. A procedure with body $s_1;\ldots;s_n$ is represented by the PCB $(true, s_1 \ldots, s_n)$.

If some statement within a block B is a compound statement (if or while), then a block is recursively created for that statement. That block will be connected to the "parent" block B by *interfaces*, which are explained below. If the statement is a while, then it is removed from B. If it is a conditional, then it is

```
int Add(int a,int b){
  return a+b;
}
void F(int c){
  int p=0, n=0;
  while(c ≥ 0){
    if(c%2 == 0)
       p=Add(c,p);
    else
       n=Add(c,n);
    c=c-1
 } print(p);
}
```

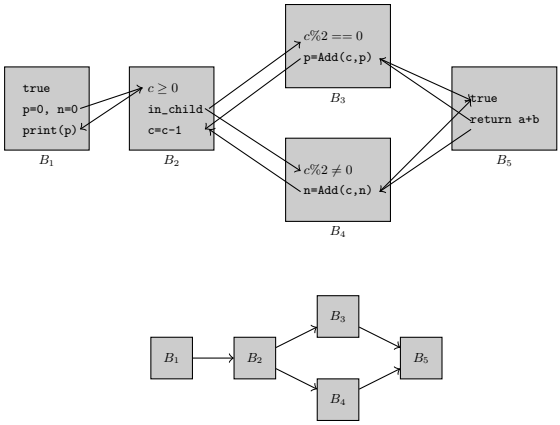

Fig. 2. Running Example (left), PCB representation (above right), immediate successor relation among PCBs (below right)

replaced by a so-called *in-child* statement that acts as a place-holder (as shown in PCB B_2 in Fig. 2). The in-child statements play a certain role in directing the flow of the slicing algorithm, see Section 5. The whole set of PCBs representing the program is generated in a single traversal of the syntax tree, generating new blocks when compound statements are encountered.

Within a PCB the program flow is represented by the sequence of statements. For blocks representing while loops the represented flow is *cyclic*, with a back-edge from the last statement to the condition, whereas the flow represented by PCBs for conditional branches and non-recursive procedure bodies is acyclic.

Program flow between PCBs can be represented by *interfaces*. These are graph edges that connect statements in different blocks: $s_i \rightarrow s'_j$ is an interface if s_i and s'_j belong to different blocks B, B', and there is a possible direct flow of execution from s_i to s'_j. Interfaces will connect blocks representing while and if statements to their "parent" blocks, and also procedure bodies to call sites. Since we consider structured code, there will always be a "forward" interface from parent to child block and a "back" interface in the reverse direction. We obtain the following possible interfaces connecting B and B' (s_0 and s'_0 are the conditions in B and B' respectively):

- $s_i \rightarrow s'_0$: a forward interface where either s_i is a simple statement or an *in_child* statement in B, and code block B' originates from s_{i+1} which is an if or while statement, or s_i is a procedure call and B' is the block for the body of the called procedure.
- $s_0 \rightarrow s'_{i+1}$: a back interface where B originates from s'_i which is a while statement, and s'_{i+1} is a simple statement in B'. (If s'_i is the last statement in the program part represented by B', then we can consider s'_{i+1} to be a dummy statement following s'_i.)

- $s_n \rightarrow s'_{i+1}$: a back interface where either B originates from s'_i, which is an if statement, and s_{i+1} is a simple statement, or B is a procedure body block and s_{i+1} is a call to this procedure in B'. (s_n is the last statement in B.)
- $s_0 \rightarrow s'_0$: a forward interface. s_i and s_{i+1} are statements in a parent block to B and B'. B originates from s_i which is a while staement, and B' originates from s_{i+1} which can be an if or while statement.
- $s_n \rightarrow s'_0$: a forward interface. s_i and s_{i+1} are as above. B originates from s_i which is an if statement, and B' originates from s_{i+1} which can be an if or while statement.

In the following we will sometimes write $B.s$ to emphasise that statement s belongs to block B. We say that B' is an *immediate successor* to B, or $succ(B, B')$, if B' originates from a compound statement in the program part represented by B.

Consider the example in Fig. 2. The C code to the left is represented by the set of interconnected PCBs to the right. Note how the connecting edges represent the inter-PCB control flow, while the intra-PCB control flow is represented by the sequences of statements within the blocks. At the bottom, to the right, the immediate successor relation between the PCBs is shown.

5 The Slicing Approach

We slice a program represented by a set of PCBs by solving local slicing problems for PCBs individually. Initially, the slicing criterion for the global slicing problem is distributed into local slicing criteria for the different PCBs. The local slicing problems are then solved. In this process, new sets of SLVs may be communicated over the backedges of interfaces connecting PCBs. The arrival of such a set will create a new local slicing problem, which subsequently is solved taking the set of SLVs as local slicing criterion. This procedure is iterated until no local problems remain: then the slice can be computed as the union of the locally sliced statements, and the algorithm terminates. A simple work list or job pool can be used to manage the computation.

Solving the local slicing problems amounts to computing both data and control dependencies to identify statements to slice. We use the SLV analysis, which is a backward data flow analysis, for computing data dependencies where the analysis of a block will proceed backwards from the statement(s) of the slicing criterion towards the condition. If the block is acyclic then the local analysis will terminate there; if it is cyclic then it will continue backwards, through the backedge towards the condition.

Our version of the SLV analysis uses the fact that the *kill* and *gen* sets of a statement are constant. This has three consequences:

- a statement s where $v \in kill(s)$ for some SLV v can be immediately sliced,
- the variables in $gen(s)$ can then be generated as SLVs once and for all, not being re-generated at subsequent visits to s, and

– an SLV that is created at statement s, in a cyclic block, need only be propagated until it comes back to s: then it can be safely removed.

Every SLV will therefore be removed sooner or later: either by being killed, or by being removed at the end of a cycle for a cyclic block or at the beginning of an acyclic block, and it can only be generated a finite number of times since the statement generating it will be sliced, and there can only be finitely many such statements. Thus, rather than doing a conventional fixedpoint iteration where SLV sets grow to reach the fixedpoint, we maintain a single set of SLVs for each block, and the iteration continues until this set is empty. To track the movement of SLVs the SLV sets contain *SLV pairs* (i, v), rather than just variables v, where s_i is the first statement in the block to start looking for the definition of the SLV v. Global slicing criteria SC are also sets of SLV pairs: if variable v at statement s_i belongs to the slicing criterion, then $(i, v) \in SC$. For each block B, its set of SLV pairs S_B is initialized to its part of the global slicing criterion before the slicing starts.

If an SLV pair (k, v) is propagated or generated by a statement s_i in B, with an incoming interface edge $s'_j \rightarrow s_i$ where s'_j is in $B' \neq B$, then (k, v) will be propagated into (j, v) at s'_j. This pair will be added to the slicing criterion $S_{B'}$ for the next local slicing problem for B' to be solved.

Control dependencies are handled in the following way. A statement in block B is control dependent on the condition of B as well as the condition of all blocks "above" it: that is, all blocks B' such that $succ^+(B', B)$ where $succ^+$ is the transitive closure of *succ*. The first time a statement is sliced in block B, the condition c of B' will be sliced for each B' where $succ^+(B', B)$, and $\{\,(0, v) \mid v \in gen(c)\,\}$ will be added to the current slicing criterion of B'.

Suppose the given program is represented by the interconnected PCBs $\langle Bs, Rs \rangle$ where Bs is the set of PCBs, and Rs is the set of interface edges connecting the PCBs. The initial set of SLV pairs S_B for all $B \in Bs$ is obtained from the global slicing criterion SC. The SLV set S_B for B is updated during the local SLV data flow analysis and slicing. This is done by repeatedly applying some of the transfer functions $f^j_{i,e,v}$ to the set S_B, visiting the statements in B in the backward direction. The transfer function $f^j_{i,e,v}$ applied to S_B models how (i, v) is successively propagated, and which effect this has on the set of SLVs. j is the current point where the set of SLV pairs is updated according to the SLV transfer function (3) based on the dependency between s_j and v, and e is an "endpoint" where the propagation of (i, v) can be safely terminated. It is defined as follows:

$$f^j_{i,e,v}(S_B) = \begin{cases} S_B \backslash \{(i,v)\} & \text{if } (j = e \wedge v \notin kill(s_j)) \\ & \vee\ (s_j = \text{"}in\_child\text{"}) \vee \\ & (v \in kill(s_j) \wedge s_j \in N_{slice}) \\ S_B & \text{if } j \neq e \wedge v \notin kill(s_j) \\ S_B \backslash \{(i,v)\} \cup \{(j-1,u) | u \in gen(s_j)\} \cup \Phi_B & \text{if } v \in kill(s_j) \wedge s_j \notin N_{slice} \end{cases} \tag{3}$$

Here, $\Phi_B = \{ (0, v) \mid v \in gen(s_0), s_0 \in B, s_0 \notin N_{slice} \}$ is the set of SLV pairs generated due to control dependency.

Algorithm 1. SlicingProgram($\langle B_s, R_s \rangle, SC$)

```
   /* Initialization */
 1 forall (B ∈ B_s) do  S_B := select SLV pairs from SC for B;
 2 forall (s → s' ∈ R_s) do  R_m(s → s') := ∅;
 3 N_slice := ∅;
   /* Slicing PCBs */
 4 while (∃B ∈ B_s. S_B ≠ ∅) do
 5 |   Let B = (s_0, ..., s_n) ;
 6 |   (i, v) := Select(S_B) ;
 7 |   e := i if B is cyclic and n otherwise;
 8 |   j := i;
 9 |   repeat
10 |   |   S_B := f^j_{i,e,v}(S_B);
11 |   |   if (B'.s_l → B.s_j ∈ R_s ∧ v ∉ kill(B.s_j) ∧ v ∉ R_m(B'.s_l → B.s_j)) then
12 |   |   |   S_B' = S_B' ∪ {(l, v)};
13 |   |   |   R_m(B'.s_l → B.s_j) = R_m(B'.s_l → B.s_j) ∪ {v} ;
14 |   |   if (v ∈ kill(s_j) ∧ (s_j ∉ N_slice) then
15 |   |   |   N_slice := N_slice ∪ {s_0, s_j};
16 |   |   |   forall B' such that succ+(B', B) ∧ B'.s'_0 ∉ N_slice do
17 |   |   |   |   S_B' := S_B' ∪ { (0, v) | v ∈ gen(B'.s'_0) };
   |   |   |   |   /* s'_0 is the condition in B' */
18 |   |   |   |   N_slice := N_slice ∪ {B'.s'_0};
19 |   |   |   if (B'.s_l → B.s_j ∈ R_s ∧ v ∉ R_m(B'.s_l → B.s_j)) then
20 |   |   |   |   S_B' = S_B' ∪ { (l, v') | v' ∈ gen(B.s_j) };
21 |   |   |   |   R_m(B'.s_l → B.s_j) = R_m(B'.s_l → B.s_j) ∪ gen(B.s_j) ;
22 |   |   |   break;
23 |   |   if (s_j = "in_child") ∨ (v ∈ kill(s_j) ∧ s_j ∈ N_slice) then
24 |   |   |   break;
25 |   |   j := (j − 1) mod (n + 1);
26 |   until  (j = e);
27 return N_slice;
```

Algorithm 1 describes the steps of slicing based on the SLV data dependence analysis in the given PCBs $\langle Bs, Rs \rangle$. It will, for each block B with nonempty slicing criterion S_B, repeatedly pick a SLV pair (i, v) from S_B and propagate (i, v) in the block until either killed, or the "endpoint" e is reached. If (i, v) is killed at s_j then s_j is sliced, if not already done, and new SLV pairs are generated according to $gen(s_j)$ and added to S_B: the details are found in (3), where the transfer function $f^j_{i,e,v}$ is defined. If a statement $s_j \in B$ is sliced, the condition $s_0 \in B$ is sliced also as s_j is control dependent on the condition s_0 as well as on the conditions for all blocks B' "above" it (lines 16-18).

For each interface $s \to s'$, the set $R_m(s \to s')$ contains the SLVs propagated over the interface so far. This set is used to prevent the same SLV to be propagated several times: if not done, there would be a risk of nontermination due to an SLV being propagated in a cycle through different blocks without ever being killed.

6 Interprocedural Slicing

The slicing algorithm discussed so far is intra-procedural. In this section we sketch how to extend it to an inter-procedural algorithm. We consider an extension of the model language in Section 2.1 with procedures according to the following. The arguments are both readable and writable, and actual arguments are variables. (Basically this is call-by-reference, passing pointers to the actual arguments.) For simplicity we assume that each procedure has a single return statement, at the end of the procedure body. Our approach is easily extended to deal with arguments that are call-by-value, and multiple return statements: indeed, our implementation can handle these features.

Global variables accessed in the procedure body can be considered as actual arguments with the same name as the formal argument. Thus, global variables are handled without further ado.

The PCB representation is easily extended to represent procedures and procedure calls. Each procedure body is represented by a block. Interfaces connect call sites with procedure body blocks: from each call site a "call" interface reaches the entry point of the procedure, and a "return" interface connects the return statement with the call site. This is similar to how procedure calls are represented for CFGs [19]. An example is shown in Fig. 2.

We now describe an approach to inter-procedural slicing that is context-sensitive in that it treats different call sites separately. For a call site $p(\overline{a})$ to procedure p, wich actual arguments $\overline{a}$, there are two problems to solve:

- How will the call contribute to the slice of the procedure body of p?
- What is the transfer function for the call? (I.e., given a SLV pair (i,v) where v is an actual argument to p, which actual arguments when entering p may influence the value of v at exit?

Our solution addresses both these problems. The idea is to slice the procedure body separately for each call site, and for each SLV appearing as actual argument. For each procedure p, and formal argument x of p, we define $\mu_p(x)$ as the set of formal arguments whose value at entry may affect the value of x at return. μ_p can be used to compute transfer functions for calls to p by substituting actual arguments for formal arguments. Our inter-procedural algorithm works as follows when encountering a call site $p(\overline{a})$. Assume that an SLV pair (i, v) appears at the return interface of $p(\overline{a})$:

1. If v is not an actual argument in $\overline{a}$, then the pair is just propagated to the call interface. No slicing of p takes place.

2. If v is an actual argument in $\overline{a}$, for the formal argument x, then we check whether $\mu_p(x)$ is already computed or not:
   - If it is, then $\mu_p(x)$ is used to compute new SLV pairs at the call interface by substituting actual arguments for formal arguments. No slicing is done of the body of p.
   - If it is not, then the slicing of the block holding $p(\overline{a})$ is temporarily halted. Instead the body of p is sliced, with x as slicing criterion at the return statement. During this process, $\mu_p(x)$ can be computed by adding some bookkeeping of SLVs to Algorithm 1. Then the slicing of the block holding $p(\overline{a})$ is resumed, using $\mu_p(x)$ as above.

This demand-driven approach attempts to minimise work by computing $\mu_p(x)$ only for formal arguments x holding some SLV as actual argument, and by reusing already computed sets $\mu_p(x)$ thus avoiding slicing the procedure body anew for the same slicing criterion.

When all call sites to p have been sliced, for all SLVs appearing as formal arguments, the slice of the body of p is computed as the union of the slices computed with slicing criteria formed from the different SLVs. Note that each slicing of the procedure body should be seen as a separate slicing problem, where the original body is sliced: if subsequent slicings are done on already sliced code, then dependences may be different from the original code and the computed $\mu_p(x)$ sets may be incorrect.

7 Experimental Evaluation

We have implemented both the SLV-based and the PDG-based slicing algorithms, and made an experimental evaluation of the results in order to measure the relative correctness, and compare the efficiency of both approaches. Both implementations are done in Microsoft Visual C++ 2013 (MVC), programs are parsed using the "Regular Expression" built-in library in MVC, and experiments have been performed on an Intel Core i5 3320M with 2.66GHz processor, 8 GB RAM, and 64-bit operating system. In addition to our own implementations we have also measured the running times for the CodeSurfer commercial tool[1], which uses PDG-based slicing. The execution times of CodeSurfer are not directly comparable to the execution times for our implementations, as CodeSurfer is implemented in a different framework, but it is still of interest to compare with a state-of-practice tool.

Five factors influence the time and space complexities of the slicing process: number of lines of code (LOC), number of variables, percentage of conditional predicates, size of the slice relative to the size of the sliced program, and the maximum depth of the nested loops. Our experiments study each factor individually. To do so we have used automatically generated, synthetic codes where these factors are systematically varied. These codes can be generated either in the model language of Section 2.1, or as the corresponding C code. Our implementations analyse the model language, and CodeSurfer analyses C code.

[1] From GrammaTech, www.grammatech.com

7.1 Experimental Results

As regards of correctness, the PDG-based and the SLV-based slicing compute the same slices in all our experiments. Otherwise the experiments compare running times and memory consumption, and we compute speedup figures. The parsing time is not included in the running times for any method. For CodeSurfer, which also performs a number of other analyses, the time is measured as the time to build the PDG plus the time to perform the backwards search to form the slice. The resolution of the time reported by CodeSurfer to build the PDG is whole seconds, which limits the precision of the computed execution time. In some experiments the reported time for building the PDG is zero: those cases are marked as "-" in the tables.

Table 1. SLV- vs PDG-based slicing: (a) code size is varied, (b) slice percentage is varied

(a)

	10K	20K	50K	100K	150K	200K
SLV(sec.)	0.015	0.034	0.081	0.165	0.253	0.343
PDG(sec.)	0.676	1.5	4.028	8.183	12.86	16.945
C.Surfer(sec.)	-	2	6	15	31	53
SpeedupPDG	45.06	44.12	49.73	49.59	50.83	49.40
SpeedupCS	-	58.82	74.07	90.90	122.53	154.52

(b)

80%	40%	20%	10%	1.5%
0.172	0.093	0.062	0.031	0.015
8.158	8.127	8.128	8.128	8.112
15.45	15.25	15.155	15.07	15
47.43	87.39	131.10	262.19	540.80
89.8	164.0	244.4	486.1	1000.0

Table 1(a) shows the slicing time of six different source codes varying in size from 10K to 200K. The slicing is performed with respect to a single slicing criterion, and time is measured in seconds. Each example program contains 50 variables, 18% conditional predicates, the nesting of conditionals is at most four, and the relative size of the slices is 70% of the source code. The execution time is computed as an average of the times for five different runs. The execution times of SLV-based slicing, our implementation of PDG-based slicing, and the PDG-based slicing by CodeSurfer are shown in rows 2-4. The fifth and the sixth rows show the speedups obtained by the SLV-based slicing over the PDG-based slicing implemented by our tool, and CodeSurfer respectively. The results in both rows indicate that SLV-based slicing gains a significant speedup compared to the PDG-based method when computing single slices.

The reason for this speedup is that the SLV-based slicing is a demand-driven method that avoids computing unnecessary data dependencies. As the PCB-based representation efficiently captures all control dependencies, detecting control dependencies also becomes inexpensive. PDG-based slicing, on the other hand, requires a complete dependence analysis. This method consumes most of its time in building the PDG. Therefore, for the same source code, decreasing the slice sizes (i.e., slicing w.r.t. a different slicing criterion) does not have a noticeable effect on the execution time for the PDG-based method whereas the SLV-based method runs significantly faster for smaller slices. Table 1(b) gives

execution times when the relative size of the slice varies from 80% to 1.5%, with source code size 100K, containing 50 program variables, 18% conditional predicates, and nesting of conditionals at most four. As can be seen, the speedups are significantly higher for the SLV-based method when the relative size of the slice is smaller.

Increasing the number of variables will on average yield an increased number of data dependencies in the program code. This increases the execution time of both building the PDG, and the sizes of the SLV sets for the SLV-based slicing method. This is shown in Table 2(a), where the slicing time increases with the number of variables for both methods. Still, the SLV-based slicing method is consistently faster than the PDG-based method. We also compare the memory consumption of these methods in this table. The SLV-based slicing consumes considerably less memory than the PDG-based slicing, and the difference increases as the number of variables grows.

Table 2. SLV- vs PDG-based method: (a) varying number of variables, (b) varying number of control predicates. Program size 100K, 50 variables (b), 18% conditional predicates (a), nesting of conditionals at most four, relative slice size 70%.

(a)

	40	80	120	160	200	300	500
SLV(sec.)	0.171	0.359	0.624	1.061	1.622	3.65	12.574
PDG(sec.)	4.321	15.881	33.181	54.288	78.655	141.961	406.5
SpeedupPDG	25.3	44.2	53.2	51.2	48.5	38.9	32.3
C.Surf(sec.)	13.47	32.5	48.59	62	84	151.56	238.6
SpeedupCS	78.8	90.5	77.9	58.4	51.8	41.5	19.0
SLV(Mb)	17.3	23.5	25.8	26.7	29.4	32.9	31.7
PDG(Mb)	96.7	348.5	501.8	763	912	1005	1600
Mem.Save	5.6	14.8	19.4	28.5	31.0	30.5	50.3

(b)

7%	14%	28%
0,078	0,125	0,156
0.671	2.698	28.267
8,60	21,58	181,20
4	10	21
51.28	80.00	134.62
14	16	19.4
49.7	122.5	506.8
3.5	7.6	26.1

Table 2(b) shows the effect on slicing time and memory consumption of varying the percentage of control predicates in the code. The slicing time and memory requirements increase with the number of control predicates for both methods, however much faster for the PDG-based method. For the SLV-based method the number of PCBs will increase, which will affect the execution time some. For the PDG-based method, however, several factors affecting the execution time will increase. First, the size of the CFG will be larger since more control predicates will yield more edges. Second, the number of data dependencies will tend to increase since there will be more paths in the code, increasing the likelihood that different dependencies are carried by the same variable through different paths. This will result in a larger PDG. Third, more fixed-point iterations will be needed in the Reaching Definitions analysis due to the more complex nature of the control and data flow.

The results shown in Table 2(b) confirm that the influence of an increased percentage of control predicates is stronger for the PDG-based method than

for the SLV-based method, both as regards execution time and memory consumption. Both the speedup and the relative memory savings for the SLV-based method grow quickly with the percentage of control predicates.

We also have studied briefly the effects of increasing the depths of loop nests on the slicing time. This is interesting since loops will necessitate conventional fixed-point iterations in the RD data dependency analysis used by PDG-based slicing, whereas our SLV-based method uses a different mechanism to handle loops.

We have obtained increasing speedups over the PDG-based method when the source code contains loops that are nested deeper. For example, we analyzed a program for loop nesting depths one to five, obtaining speedups 2.4, 6.3, 14.7, 27.3, 48.2. When we compared the execution times of our SLV-based method with those of CodeSurfer, we obtained the speedups -, 80.0, 96.0, 92.0, 92.5. (Here, the speedup for the first item could not be computed since the time resolution of CodeSurfer gave zero execution time.)

8 Related Work

Program slicing was first introduced by Weiser [24,25] in the context of debugging. Since then, different approaches to slicing were introduced including static [24,25], conditioned [7], dynamic [13], amorphous [10], semantic [3], and abstract slicing [11,18], of which static slicing techniques are the ones that are most comparable to our approaches.

Ottenstein et. al. [9,20] introduced the PDG, and proposed its use in program slicing. PDG-based slicing has then been the classical program slicing method that has been extended to interprocedural slicing [12,21,22], to produce executable slices [1,2,8], and to handle pointers [16].

Apart from PDG-based slicing, slicing based on data flow equations have been proposed by Weiser [24,25], Lyle [17], and Lisper [15]. Even though some of these approaches can handle unstructured control flow and low-level code, our approach is more time and memory efficient for computing single slices of source codes that does not contain arbitrary jumps. There have been several empirical studies and survey papers [5,6,23,26] that compare different slicing techniques.

Sprite [4] is a slicing approach that divides the information of the code into two levels: hard-to-demand and on-demand. The hard-to-demand information is calculated early, and on-demand information like the data and control dependencies are found during the slicing. The precision of Sprite is tunable, but the algorithm favors less precise slicing. Katana [14] is a slicing approach that uses a database-like program representation: it is fast, but less precise than PDG-based slicing.

9 Conclusions and Future Work

We have proposed a slicing approach for well-structured programs appearing in safety critical systems. The algorithm performs the slicing in a dynamic fashion,

using an internal representation that is efficiently derived from the syntax tree. An experimental evaluation indicates that the algorithm outperforms the current state-of-the practice algorithm by a magnitude, both as regards execution time and memory requirements, when single slices are taken for the same slicing criterion. As future work we would like to extend the slicing approach to handle arbitrary control flow, and make an empirical evaluation for large industrial code. Furthermore we want to investigate how well a parallel job pool implementation performs: its decoupled job structure should make the algorithm very amenable to such an implementation.

Acknowledgments. The research presented in this paper is supported by the FP7 Marie Curie IAPP programme under the project 251413 APARTS, by the Swedish Foundation for Strategic Research under the SYNOPSIS project, and by the KKS Foundation under the project TOCSYC. We also thank GrammaTech for providing access to CodeSurfer.

References

1. Agrawal, H.: On slicing programs with jump statements. SIGPLAN Not. **29**(6), 302–312 (1994)
2. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: Fritzson, P.A. (ed.) Proc. First International Workshop on Automated and Algorithmic Debugging, AADEBUG 1993. LNCS, vol. 749. pp. 206–222. Springer, Heidelberg (1993)
3. Barros, J.B., da Cruz, D., Henriques, P.R., Pinto, J.S.: Assertion-based slicing and slice graphs. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, pp. 93–102. IEEE Computer Society, Washington, DC (2010)
4. Bent, L., Atkinson, D.C., Griswold, W.G.: A qualitative study of two whole-program slicers for C. University of California San Diego, Tech. rep. (2000)
5. Binkley, D., Harman, M.: A large-scale empirical study of forward and backward static slice size and context sensitivity. In: Proc. International Conference on Software Maintenance, ICSM 2003, p. 44. IEEE Computer Society, Washington, DC (2003)
6. Binkley, D., Harman, M.: A survey of empirical results on program slicing. In: Advances in Computers, Advances in Computers, vol. 62, pp. 105–178. Elsevier (2004)
7. Canfora, G.: Conditioned program slicing. Information and Software Technology **40**(11–12), 595–607 (1998)
8. Choi, J.D., Ferrante, J.: Static slicing in the presence of goto statements. ACM Trans. Program. Lang. Syst. **16**(4), 1097–1113 (1994)
9. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
10. Harman, M., Binkley, D., Danicic, S.: Amorphous program slicing. In: Software Focus, pp. 70–79. IEEE Computer Society Press (1997)
11. Hong, H.S., Lee, I., Sokolsky, O.: Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 25–34 (2005)

12. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)
13. Korel, B.: Dynamic program slicing. Information Processing Letters 29 (October 1988)
14. Kraft, J.: Enabling Timing Analysis of Complex Embedded Software Systems. Ph.D. thesis, Mälardalen University Press (August 2010)
15. Lisper, B., Masud, A.N., Khanfar, H.: Static backward demand-driven slicing. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM 2015, pp. 115–126. ACM, New York (2015)
16. Lyle, J.R., Binkley, D.: Program slicing in the presence of pointers (1993) (extended abstract)
17. Lyle, J.R.: Evaluating Variations on Program Slicing for Debugging (Data-flow, Ada). Ph.D. thesis, College Park, MD, USA (1984)
18. Mastroeni, I., Nikolić, D.J.: Abstract Program Slicing: From Theory towards an Implementation. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 452–467. Springer, Heidelberg (2010)
19. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2nd edn. Springer (2005). iSBN 3-540-65410-0
20. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. SIGSOFT Softw. Eng. Notes **9**(3), 177–184 (1984)
21. Reps, T., Horwitz, S., Sagiv, M., Rosay, G.: Speeding up slicing. In: Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1994, pp. 11–20. ACM, New York (1994)
22. Sinha, S., Harrold, M.J., Rothermel, G.: System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 432–441. ACM, New York (1999)
23. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3**, 121–189 (1995)
24. Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering SE-**10**(4), 352–357 (1984)
25. Weiser, M.D.: Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. Ph.D. thesis, Ann Arbor, MI, USA (1979), aAI8007856
26. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005)