Enhanced PCB-Based Slicing

Husni Khanfar and Björn Lisper

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

Abstract. Program slicing can be used to improve software reliability. It enables identification and checking of critical points by removing program parts that do not influence them. The standard slicing method computes the dependencies of an entire program as a prerequisite to the slicing. Demand-driven techniques such as our work Predicate Control Block (PCB)-based slicing computes only the dependencies affecting critical points. This improves the performance for single slices.
This paper extends PCB-based slicing to efficiently compute several slices from the same code. This is done by storing the computed data dependencies in a form of graph to reuse them between individual slices. We also show how PCB-based slicing can be done interprocedurally in a demand-driven fashion. Moreover, we describe a filtering technique that reduces the exploration of irrelevant paths. These two improvements enhance the algorithm performance, which we show using synthetic benchmarks.

Keywords: Program Slicing, Reliable Software, Predicate Control Block

1 Introduction

Backward program slicing extracts the set of statements (so-called "slice") that may affect a slicing criterion. A slicing criterion refers to the value of a particular variable at a program location. In considering critical points in reliable systems as slicing criteria, backward slicing enables us to study many aspects related to those points. For a given slicing criterion, the slicing computes the statements, inputs and conditions that possibly affect the slicing criterion. The effect can appear by two ways: *Control Dependence*, which occurs where a predicate controls the possible execution of statements and *Data Dependence* which occurs when a variable updated at a statement is used in another.

Software programs are getting more complex and it is important for those programs to be reliable. Software reliability refers to the continuity of correct service [14]. To deliver a correct service, the cornerstone is in being fault-free. To find faults, the static analysis methods such as path simulation or the verification methods such as model checking are used. Larson states in [15] that "A major problem with path-based defect detection systems is path explosion". Model checking performs automatic exhaustive testing and Choi et al states in [3] that it might suffer from state-space explosion. The fact that program slicing reduces

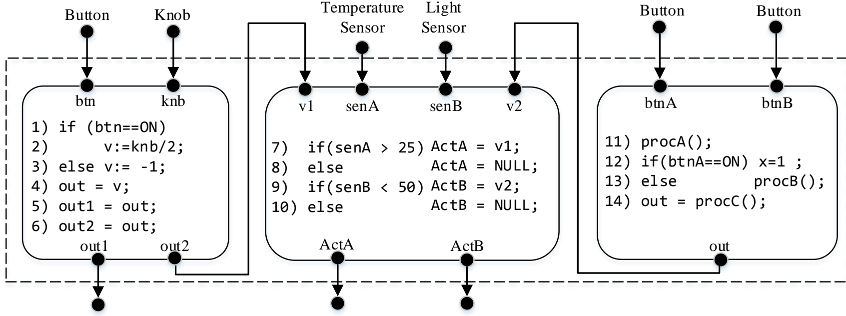

Fig. 1: Simple Control Unit

the program size is important because it can help alleviate the path and state-space explosion.

Program slicing can also be used to check for possible dependencies. consider the simple control unit in Fig. 1. Assume that `ActA` is only to be controlled by `v1` and `senA`, the dependence on any other input being considered a fault. For studying this system with respect to `ActA`, we choose `ActA` as a slicing criterion. The slice taken will show that `senA` may be dependent on v1 and `senA` but is surely not dependent on `senB` or `v2`, ensuring that the system is correct with respect to input dependencies.

The unit in Fig. 1 has six inputs. If we suppose that each button has 2 states, each knob and sensor has 10 states, then we need 32K states to cover all the input combinations. If the aim is to study `ActA`, then it is enough to generate the states relevant to `ActA`. These relevant states are computed from the inputs that `ActA` is dependent on. Notice that the slice of the control unit with respect to `ActA` consists of the program lines: {1,2,3,4,6,7,8} and the program inputs: {`buttonA,Knob1,v1,senA`}. In this case the number of states are: 400. Apparently, the state-space is reduced significantly.

The most common slicing technique is designed on the program representation Program Dependence Graph (PDG) [11]. The PDG consists of nodes and edges which represent statements and direct data and control dependencies respectively. To construct a PDG, a deep comprehensive analysis is performed to compute all dependencies in prior of the slicing. This analysis is reported by many authors [3, 4, 6] as very time and space consuming.

Demand-driven approaches compute data and control dependencies on demand during the slicing operation and not in prior. This gets rid of computing unrelated dependencies which cause unnecessary computations. Our previous work in Predicate Control Block(PCB)-Based slicing [1] is an example of a demand-driven approach. PCB is a basis of program representations that models well-structured, inter-procedural and jump-free programs. This new slicing method is designed for applications that are compliant with MISRA-C and SPARK, which are software development guidelines for writing reliable and safety-critical applications in C and Ada languages respectively.

Demand-driven slicing approaches focus on eliminating unrelated dependencies, but these dependencies are not the only source of unnecessary computations. In computing data dependencies relevant to a particular program point p, all backward or forward paths from p have to be explored. Most of these paths do not include any relevant information. e.g. in Fig. 1 for ActA point, we explore (9) and (10) and this is wasteful.

Sometimes, many critical points in a program need studying. Thus, a slice for each point has to be produced. In demand-driven slicing approaches such as PCB-based approach, when a source code is sliced many times in accordance to many individual slicing criteria, the data dependencies involved in each slice have to be computed from scratch even though some or all of them were already computed for a previous slice. These computations are unnecessary because they already were done before. e.g. in our example, (4) is dependent on (2) and (3) which are dependent on (1). These dependencies affect `out1` and `out2`. If `out1` and `out2` are two individual points of interest, then we must compute these dependencies for each of them.

The contributions of this paper are:

1. Extending the PCB-based algorithm to reuse information from previous slicing, making it more efficient when slicing the same code several times for individual slicing criteria.
2. Extending the PCB-based algorithm to the interprocedural case. This was sketched in [1]: here we explain the full method.
3. Clarifying some parts of the PCB-Based slicing algorithm presented in [1]. This includes the formal definition of how to represent programs by PCBs, and how to handle the communication of data flow information between different program parts efficiently. Thus reducing the exploration of irrelevant paths.

The rest of the paper is organized as follows. Section 2 introduces essential background. In Section 3 we describe PCBs and summarizes PCB-Based slicing approach in [1]. We illustrate how the data dependencies are saved and retrieved between slices in Section 4, while in Section 5 the communications between PCBs are filtered. Section 6 presents the two-modes algorithm. Section 7 onfly interprocedural slicing, In Section 8 we give an account for our experimental evaluation. Section 9 gives an account for related work, Section 10 concludes the paper and Section 11 acknowledgments

2 Preliminaries

2.1 Model Language

WHILE is a simplified model language for languages for safety-critical applications, such as SPARK Ada and MISRA-C. A program is a statement (**s**) consisting of a sequence of statements. WHILE statements are classified into two main types; elementary and conditional statements. Elementary statements are

assignment, *skip*, boolean expressions and the special *in_child* explained later. Conditional statements are *if*, *while* and *ifelse* statements.

We assume that each elementary and conditional statement is uniquely labeled in a program. **Label** is the set of global labels in a program and $\ell \in$ **Label**. Let a denote arithmetic expressions and b boolean expressions. The abstract syntax of the WHILE language is:

$$cs ::= [\texttt{if } [b]^{\ell} \texttt{ then } s']^{\ell'} \mid [\texttt{if } [b]^{\ell,\ell'} \texttt{ then } s' \texttt{ else } s'']^{\ell''} \mid [\texttt{while } [b]^{\ell} \texttt{ do } s']^{\ell'}$$
$$es ::= [x := a]^{\ell} \mid [b]^{\ell} \mid [skip]^{\ell} \mid [in_child]^{\ell}$$
$$s ::= es \mid s'; s'' \mid cs$$

2.2 Strongly Live Variable (SLV) Analysis

A data dependence relation is a relation on program labels relating points of variable definitions with their uses. We write $\ell \xrightarrow{v} \ell'$ to signify that ℓ is data dependent on ℓ', i.e., that the statement labeled ℓ' defines a variable v used by the statement labeled ℓ.

The data dependence relation has two sides: *Definition* and *Use*. The *Definition* is a statement where a variable (v) is updated and then reaches without being redefined to a statement using v (*Use*). Accordingly, *Use* is data dependent on *Definition*. In this context, the data dependence is symbolized by $\rightarrow$ as:

$$\ell \xrightarrow{v} \ell' \subseteq Label \times Var \times Label \tag{1}$$

where ℓ is data dependent on ℓ' in terms of the variable v. Sometimes, v is not specified in this relation.

A variable v is live at program point p if there is a definition free path from a use of v to p. A live variable analysis is a backward analysis [10] that computes the set of live variables associated with each program point. Strongly live variable analysis is a restriction of live variable analysis to an initial set of strong live variables, which are the variables of interest [10]. Thus, a variable is strongly live at program point p if there is a definition free path to a use of v and this use defines another strongly live variable.

SLV is a data flow analysis [10]. It generates a variable used in a particular statement as a SLV, propagates it backward untill reaching a statement defining it where it is killed. Thus, SLV analysis can be utilized to find the definitions that affect the used variables in a particular statement (use). This mechanism which computes from a use the set of definitions affecting it, is the main requirement in computing dynamically data dependence facts by backward slicing.

As a traditional dataflow method, SLV analysis relies on *functions* and *equations*; *gen* and *kill* functions generate and remove SLVs respectively from individual elementary statements, and the equations compose a mechanism to propagate backward the SLVs. Since these equations are not used in PCB-based

slicing, they are not presented here. The functions are defined as follows:

$$\begin{array}{l} kill(x := a) = \{x\}, \quad kill(b) = \quad \emptyset, \quad kill(skip) = \emptyset, kill(in_child) = \textbf{Var} \\ gen(x := a) = FV(a), gen(b) = FV(b), gen(skip) = \emptyset, gen(in_child) = \emptyset \end{array} \tag{2}$$

$FV(a)$ denotes the set of program variables that appear in the expression a. **Var** is the set of variables in a program.

3 PCB-Based Slicing

This section recaptures the notions of Predicate Control Blocks (PCBs), PCB graphs and PCB based slicing as introduced in [1]. With respect to previous work the section contributes by describing the derivation of PCBs and PCB graphs in a more detailed manner.

3.1 Predicate Control Block Graphs

A PCB [1] refers to the encapsulation of a predicate and the set of elementary statements which are controlled directly by this predicate.

$$p ::= \{[b, es_1, \ldots, es_n], type\} \tag{3}$$

In addition to a predicate and a sequence of statements, PCBs carry types, *type*, signifying whether the PCB is linear, L, or cyclic, C. Intuitively, linear PCBs correspond to conditional statements, such as *if*, and cyclic PCBs correspond to iterative statements, such as *while*.

In the following let $+\!\!+$ denote concatenation of sequences and let $:$ denote the standard cons operator, i.e., $b : [es_1, \cdots] = [b, es_1, \cdots]$. Further, we lift sequence indexing to PBCs where $p[0] = b$, and $p[n] = es_n$ for $p = \{[b, es_1, \ldots, es_n], type\}$.

A PCB graph is a pair (ϕ, ϵ), consisting of a map from labels to PCBs, ϕ, and a set of edges, ϵ, represented as pairs of labels. Following [1] we refer to the edges of the PCB graph as interfaces, and write $\ell_1 \hookrightarrow \ell_2$ instead of $(\ell_1, \ell_2) \in \epsilon$, whenever the PCB graph is given by the context.

The top-level translation of a program s to a PCB graph is defined as follows for any ℓ not in s.

$$\lambda(s) = (\phi[\ell \mapsto \{true^{\ell} : es, L\}, \epsilon), \ where\ es, (\phi, \epsilon) = \lambda_{\ell}(s)$$

The bulk of the translation is done by $\lambda_{\ell}(s)$, defined in Figure 2, where $final$ returns the last label in a sequence of elementary statements. Given a statement s the $\lambda_{\ell}(s)$ returns a linearized translation of s together with the PCB graph resulting from the translation. It might be worth pointing out a few key points of the algorithm. First, each PCB inherits the label of its predicate. Second, since *if..else* statements generate two PCBs, their predicates carry two distinct labels. Third, the place of each conditional statement is replaced by a placeholder (*skip*

for *if* or *while*; *in_child* for *if..else*)[1], whose label is the label of the original statement. The translation of compound statements works by first translating the parts, and then joining and extending the results to a new PCB graph.

$$\lambda_{\ell_p}([x := a]^\ell) = [x := a]^\ell, (\emptyset, \emptyset)$$

$$\lambda_{\ell_p}([skip]^\ell) = [skip]^\ell, (\emptyset, \emptyset)$$

$$\lambda_{\ell_p}([\texttt{if } b^\ell \texttt{ then } s]^{\ell'}) = [skip]^{\ell'}, (\phi', \epsilon')$$
$$where\ es, (\phi, \epsilon) = \lambda_\ell(s)\ and\ \phi' = \phi[\ell \mapsto \{b^\ell : es, L\}]$$
$$and\ \epsilon' = \epsilon \cup \{\ell_p \hookrightarrow \ell, final(es) \hookrightarrow \ell'\}$$

$$\lambda_{\ell_p}([\texttt{while } b^\ell \texttt{ do } s]^{\ell'}) = [skip]^{\ell'}, (\phi', \epsilon')$$
$$where\ es, (\phi, \epsilon) = \lambda_\ell(s)\ and\ \phi' = \phi[\ell \mapsto \{b^\ell : es, C\}]$$
$$and\ \epsilon' = \epsilon \cup \{\ell_p \hookrightarrow \ell, final(es) \hookrightarrow \ell'\}$$

$$\lambda_{\ell_p}([\texttt{if } b^{\ell,\ell'} \texttt{ then } s \texttt{ else } s']^{\ell''}) = [in_child]^{\ell''}, (\phi'', \epsilon'')$$
$$where\ es, (\phi, \epsilon) = \lambda_\ell(s)\ and\ es', (\phi', \epsilon') = \lambda_{\ell'}(s')$$
$$and\ \phi'' = (\phi \cup \phi')[\ell \mapsto \{b^\ell : es, L\}, \ell' \mapsto \{\neg b^{\ell'} : es', L\}]$$
$$and\ \epsilon'' = \epsilon \cup \epsilon' \cup \{\ell_p \hookrightarrow \ell, final(es) \mapsto \ell'', \ell_p \hookrightarrow \ell', final(es') \hookrightarrow \ell''\}$$

$$\lambda_{\ell_p}(s; s') = es +\!\!+\, es', (\phi \cup \phi', \epsilon \cup \epsilon')$$
$$where\ es, (\phi, \epsilon) = \lambda_{\ell_p}(s)\ and\ es', (\phi', \epsilon') = \lambda_{\ell'_p}(s')$$
$$and\ \ell'_p = final(es)$$

Fig. 2: Computation of PCB graphs

To illustrate the algorithm consider the program in Figure 3. The algorithm works recursively; in order to translate the top level program, the *while* and the *if* must be translated. In the reverse order of the recursive calls, the *if* is translated first, which gives P_7. No interfaces are created, since the body of the *if* does not contain any compound statements. The resulting PCB graph is returned to the translation of the body of the *while*, and extended with P_4 and interfaces $\ell_3 \hookrightarrow \ell_7$ and $\ell_8 \hookrightarrow \ell_4$. This gives the PCB graph rooted in P_4, which is returned to the top-level translation and the graph. The final result is produced by adding P_0 and interfaces $\ell_1 \hookrightarrow \ell_4$ and $\ell_9 \hookrightarrow \ell_3$.

3.2 PCB-Based Slicing Approach

A slicing criterion is a pair of $< \ell, v >$ where ℓ is a global label and v is a variable. if ℓ belongs to the PCB P, then $< \ell, v >$ is considered as a local problem in P. $< \ell, v >$ is solved in P by propagating it backward among the local statements in P. The propagation starts from ℓ and its aim is to find the statement in P that influences v at ℓ. Since the propagation relies on the order of the internal statements in the PCB, it is important to express the local problem by its local

[1] The reason of using placeholders will be explained in Section 5

index (i) in P rather than its global label ℓ. The local slicing problem in the PCB does not have more than one local solution because each PCB has a unique free-branching path.

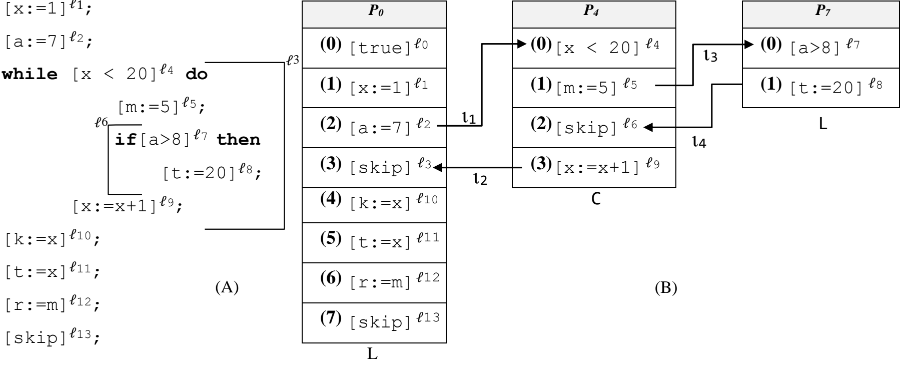

Fig. 3: Example 1

Local slicing problems of the PCB are stored in a single set (S). The PCB works as a standalone process that solves its local problems. Local problems are solved individually. While solving a problem, this problem might be *reproduced* in other PCBs, *killed* and new local problems might be *generated*. This cycle of *reproduction*, *killing*, and *generation* of local problems are iteratively repeated until no more local problem is available in any PCB.

The SLV query is solved by using *kill* and *gen* SLV functions. *kill(s)* gives the variables that s defines. *gen(s)* generates new local slicing problems from s. Therefore and henceforth, local slicing problem are named as SLV query.

Since the PCB represents a branch-free path, there is no use from using conventional fixed point iterations, which is designed to work with tree of branches and requires a set existing at each program point to save the SLVs reaching this point. The PCB has a single set to preserve its SLV queries and each SLV query is solved individually. The PCB encodes the direct control dependency. This encoding enables to capture immediately the control dependencies from the predicate of the PCB and the predicates of parents PCBs.

Suppose $S(P)$ is the single set of P. Each SLV query $< i, v >$ is fetched individually from $S(P)$. The first parameter which should be calculated for this query is its end index e. In linear PCBs, $e = 0$. Otherwise, $e = i + 1$. Then, $< i, v >$ proceeds backward from $P[i]$ toward $P[e]$. In circular PCBs, when $< i, v >$ reaches $P[0]$, it propagates backward by jumping to the last label in P and goes on backward until reaching $P[i + 1]$. In this context, the index of each visited statement in P will be j. Visiting $P[j]$ by $< i, v >$ causes one of these three cases:

case 1: if $v \not\subseteq kill(P[j])$ and $j \neq e$, then $S(P)$ remains as is

case 2: if $v \subseteq kill(P[j])$ then $< i, v >$ is removed from $S(P)$ and does not proceed more. if $P[j]$ was not sliced before, then the variables used in $P[j]$ will be generated as SLV queries and added to $S(P)$. As well, $P[j]$ is sliced.

case 3: if $v \nsubseteq kill(P[j])$ and $j = e$, then $< i, v >$ is removed from $S(P)$ and does not proceed more.

Example 1: In Fig. 3; Suppose $< \ell_{13}, r >$ is a global slicing criterion. It is translated to the SLV query: $< 7, r >$ in $S(P_0)$. Since P_0 is a linear PCB, $e = 0$. So, $< 7, r >$ is solved locally in P_0 by being propagated it from $P_0[7]$ to $P_0[0]$. The first statement visited by $< 7, r >$ is $P_0[7]$. Considering that $kill(P_0[7]) = \emptyset$ $S(P_0)$ remains as is. Next, $< 7, r >$ visits $P_0[6]$. Since $kill(P_0[6]) = r$, $P_0[6]$ is sliced, a new SLV query $< 5, m >$ is generated, $< 7, r >$ is removed from $S(P_0)$ and it no longer proceeds. Similarly, $< 5, m >$ is fetched and it visits the statements from $P_0[5]$ to $P_0[0]$. Since none of them kills m, $< 5, m >$ is removed from $S(P_0)$ after visiting $P_0[0]$□

Notice that the value of m at $P_0[6]$ is affected also by $P_4[1]$. Therefore, a new slicing problem has to be created in P_4 according to the following rule:

Suppose P and P' are two PCBs connected by $P'[j'] \hookrightarrow P[j]$. When an SLV query $< i, v >$ visits $P[j]$ and not being killed at it, $< i, v >$ is reproduced in P' as $< j', v >$.

Example 2: In Fig. 3, P_0 is connected to P_4 by $P_4[3] \hookrightarrow P_0[3]$. When $< 5, m >$ visits $P_0[3]$, it is reproduced in P_4 as $< 3, m >$, which is processed in P_4 by visiting $P_4[3]$, $P_4[2]$ and $P_4[1]$. At $P_4[2]$, it is reproduced in P_7 as $< 1, m >$ and at $P_4[1]$ it is killed. The query $< 1, m >$ in P_7 does not have a local solution □

Each interface $\ell \hookrightarrow \ell'$ associates to a set $R_m(\ell \hookrightarrow \ell')$, which saves the variable part of each query reproduced through $\ell \hookrightarrow \ell'$. R_m works as a blacklist whose elements are not allowed to be reproduced again through. This prevention is important in avoiding a possible non-termination which could occur when an SLV query $< i, v >$ is generated in a cyclic PCB and neither this PCB nor its child defines v. As a result, v would continuously be propagated between the parent and its child.

For if and while conditional statement which exists in other parent blocks, there is an execution path skipping the main body of the conditional statement. Thus, the local analysis in the parent block can neglect the existence of such conditional statements. For if-then-else conditional statement, there is no such skipping path. Therefore, the local analysis in its parent block could not neglect the existence of if-then-else statement. To handle this situation, if-then-else conditional statement is replaced by an *in_child* placeholder. *in_child*, which is designed especially for working as a placeholder for if-then-else statements, does not generate any SLV but it kills any SLV query visits it. It has this property because if-then-else has two branches, which both might kill the same variable.

For obtaining control dependencies; whenever any statement is sliced, then the predicate s_0 in its PCB has to be sliced if it was not sliced before. This routine has to be recursively applied also to the parent predicate until reaching the most outer PCB.

Suppose P is a child of the PCB P', as soon as $P[k]$ is sliced, then $P[0]$ should be sliced if it was not sliced and the variables used in $P[0]$ are generated as SLV queries. In addition, $P'[0]$ should be sliced if it was not sliced before and the variables used in it have to be generated as SLV queries.

Example 3: In Example 2, $P_4[0]$ is sliced due to slicing $P_4[1]$. Thus, $< 3, x >$ is generated in P_4 and solved internally at $P_4[3]$. In addition, $< 2, x >$ is reproduced in P_0 and solved at $P_0[1]$. Based on that, the slice of the global slicing criterion $< 7, r >$ consists of: $\ell_{12}, \ell_5, \ell_9, \ell_4, \ell_1, \ell_0$ □

4 Partial Data Dependency Graph (PDDG)

Sometimes, the source code might be studied from many different perspectives. e.g. the control unit of Example 1 might be studied first with respect to `ActA` and then to `ActB`. Thus, the same code should be sliced many times for individual slicing criteria. In PCB-based slicing, computing many slices suffers from the fact that the same data dependency should be computed from scratch whenever it becomes a requirement. This section shows a novel method to store and retrieve the data dependencies between slices. To do so, the computed data dependencies are stored in a graph form called Partial Data Dependence Graph (PDDG).

4.1 The Organization of a Partial Data Dependence Graph

In backward slicing, it is required to find the labels of definitions that may affect the variables used in the sliced label ℓ. PDDG is designed to store these definitions in order to be retrieved later. To build a PDDG, a special set $\delta(\ell)$ is added to each label (ℓ). $\delta(\ell)$ is initialized to the empty set. In assuming ℓ is sliced, each statement defining any variable used in ℓ should be computed, sliced and its label has to be added to $\delta(\ell)$. Consequently, the data dependencies are organized in PDDG as *use-definitions* form. In this form, the set of *definitions* affecting a particular label are stored in this label, which is a *use* to those definitions. This design enables to retrieve once all the *definitions* of this *use* when it is sliced again for a different slicing criteria.

In subsection 3.2, we saw how the definitions of ℓ are computed by generating an SLV query for each variable used in ℓ and propagate it backward. When the query $< i, v >$ generated from ℓ reaches a statement ℓ' defining v, then this is the best time to capture the data dependency between ℓ and ℓ'. The problem is that the origin of $< i, v >$ is not known because i is changed whenever it is reproduced in a new PCB. Thus, a new field (*src*) is added to the SLV query type, which becomes a triple: $< loc, src, var >$. *src* is assigned to the global label, which the SLV query is generated from. Hence, if $< i, \ell, v >$ is killed at ℓ', then $\ell \xrightarrow{v} \ell'$ which is satisfied by adding (ℓ', v) to $\delta(\ell)$. Based on that, $\delta(\ell)$ is defined as:

$$\delta(\ell) \subseteq \mathcal{P}(LABEL \times VAR) \quad (4)$$

The reason of why $\delta(\ell)$ is not a pool of labels only will be explained in Section 4.2.

4.2 The Hindrance of Interfaces

An interface ι is associated with a set $R_m(\iota)$ in order to prevent any variable from being reproduced more than once. Since the same variable may exist in different SLV queries, the interface black list may cause missing data dependences.
Example 4: In Fig. 3, if $P_0[4]$ and $P_0[5]$ are sliced, then SLV queries $< 3, P_0[4], x >$ and $< 4, P_0[5], x >$ are added to $S(P_0)$. Suppose $< 4, P_0[5], x >$ is fetched first, then it is solved at $P_0[1]$. As well, when $< 4, P_0[5], x >$ visits $P_0[3]$, it is reproduced through ι_2 in P_4 as $< 3, P_0[5], x >$ and $R_m(\iota_2) = \{x\}$. At P_4, it is solved at $P_4[3]$, So, $\delta(P_0[5]) = \{(P_0[1], x), (P_4[3], x)\}$. $< 3, P_0[4], x >$ is solved also at $P_0[1]$ but it could not be reproduced through ι_2 because $x \in R_m(\iota_2)$. Thus, $(P_4[3], x)$ could not be added to $\delta(P_0[4])$. In other words, $P_0[4] \xrightarrow{x} P_4[3]$ is not recognized □

The hindrance of interface is resolved by using a *transition point*, which refers to a label (t) existing in the path from ℓ' and ℓ, where $\ell \xrightarrow{v} \ell'$. The transition point helps in expressing a data dependence relation by two fake data dependencies. Accordingly, $\ell \xrightarrow{v} \ell'$ is represented by $\ell \xrightarrow{v} t$ and $t \xrightarrow{v} \ell'$. To overcome the hindrance of the interface $(\ell_1 \hookrightarrow \ell_2)$, we consider its ingoing side ℓ_2 as a transition point. Hence, if the SLV query $< i, \ell, v >$ visits ℓ_2 and $kill(\ell_2) \neq v$ then the fake data dependence $src \xrightarrow{v} \ell_2$ is created by adding (ℓ_2, v) to $\delta(\ell)$. If $< i, \ell, v >$ is allowed to be reproduced in $\mathcal{M}(\ell_1)$, then it will be reproduced as $< i', \ell_2, v >$, where i' is the index of ℓ_1 in $\mathcal{M}(\ell_1)$.
Example 5: Example 4 is resolved as follows: when $< 4, P_0[5], x >$ reaches $P_0[3]$, then it is reproduced in P_4 as $< 3, P_0[3], x >$ and $(P_0[3], x)$ is added to $\delta(P_0[5])$. When $< 3, P_0[4], x >$ reaches $P_0[3]$, $(P_0[3], x)$ is added to $\delta(P_0[4])$ without being reproduced at P_4. Hence, $P_0[5] \xrightarrow{x} P_4[3]$ is expressed as $P_0[5] \xrightarrow{x} P_0[3]$ and $P_0[3] \xrightarrow{x} P_4[3]$. Similarly, $P_0[4] \xrightarrow{x} P_4[3]$ is expressed as $P_0[4] \xrightarrow{x} P_0[3]$ and $P_0[3] \xrightarrow{x} P_4[3]$ □
Notice that in (4), we defined $\delta(\ell)$ as a pool of pairs rather than a pool of labels. This organization prevents creating non-existing dependencies. See Example 12.
Example 12 Suppose $\iota = \ell_i \hookrightarrow \ell_j$ is in the path of: $\ell_1 \xrightarrow{x} \ell_a$ and $\ell_2 \xrightarrow{y} \ell_b$. If δ was a set of labels only, the following data dependencies would be represented: $\ell_1 \to \ell_j$, $\ell_2 \to \ell_j$, $\ell_j \to \ell_a$ and $\ell_j \to \ell_b$, which means ℓ_1 is data dependent on ℓ_b and this is not correct.

5 SLV Filtering

In PCB-based program representation, interfaces correspond to edges in CFG program representations. Both of them are constructed to model program flows. In this section we explain in depth how the interfaces are implemented in an efficient manner to prevent SLV queries from exploring irrelevant PCBs or paths. This presentation eliminates unnecessary computations.

The cornerstone in making SLV filtration is in associating the interfaces with whitelist sets rather than blacklists which we introduced in Section 3.2. The *White List* is a smart way to implement the black list in a "negative" way,

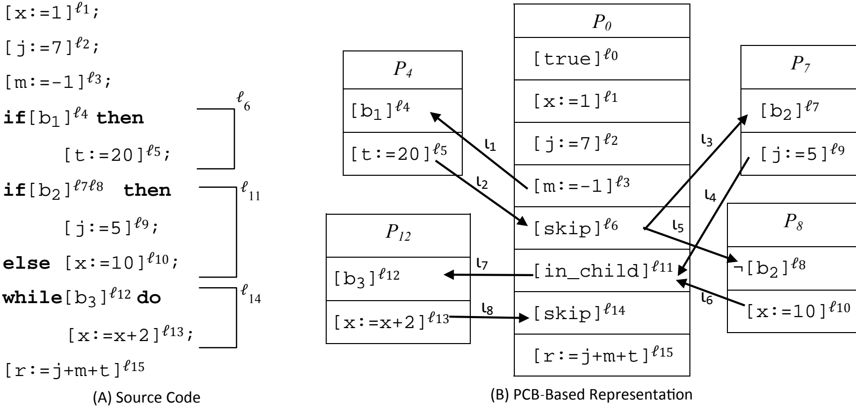

Fig. 4: PCB-Based Representation (Whitelists + Placeholders)

representing what is allowed to go through rather than what is not allowed to go through. The whitelist set associated with an interface $\iota = P.\ell \hookrightarrow P'.\ell'$, where P is a child to P', is initialized to the variables defined in P and its child PCBs. The whitelist associated with ι is denoted $W_m(\iota)$. As soon as any of the variables stored in $W_m(\iota)$ is reproduced through ι, this variable is removed from $W_m(\iota)$ and it becomes no longer allowed to be reproduced again. Based on that, the number of the variables in the whitelist is in the worst case proportional to the number of the variables in P and it is decremented after every reproduction. When the whitelist set becomes empty, then its interface could be deleted to relieve the analysis from the overhead of its existence. This is contrary to the blacklists whose sizes in the worst case are proportional to the number of the variables in the program and they increments after every reproduction. Apparently, changing from blacklists to whitelists makes a significant improvement. (See Table 2 in Section 8.2)

Example 5: Fig.4 is an example of preventing SLV queries from exploring irrelevant paths. Suppose ℓ_{15} is sliced and three SLV queries are generated whose variable components are m, x and t. Notice the t has a definition in P_4. In backward dataflow analysis techniques, t propagates in all statements in P_{12}, P_8, P_7 and P_4 although its unique external solution is in P_4. Similarly, j is reproduced in P_{13}, P_8, P_7 and P_4 although it is defined only in P_7. Figure 4-B shows four ingoing interfaces to P_0: ι_2, ι_4, ι_6 and ι_8. Their whitelist sets are initialized to: $W_m(\iota_2) = \{t\}$, $R_m(\iota_4) = \{j\}$, $W_m(\iota_6) = \{x\}$ and $W_m(\iota_8) = \{x\}$. When t is generated as an SLV from $P_0.\ell_{15}$, it visits ℓ_{14} and ℓ_{11} and neglects P_{12}, P_8 and P_7 because their whitelists do not include t. When it visits ℓ_6, t is reproduced in P_4 because it defines t. To sum up, we can say that t is forwarded directly to P_4.

6 Two-Mode PCB Slicing Algorithm

The PDDG stores and retrieves data dependencies between slices. This mechanism is yielded by designing a slicing method that behaves in two modes. In the first mode the definitions influencing ℓ are retrieved from $\delta(\ell)$ and the second mode generates the variables used in ℓ as SLV queries to reach these definitions. The contents of $\delta(\ell)$ assist the analysis to determine in any mode it should run when ℓ is sliced.

Alg.5a slices the internal labels of the PCB P with respect to SLV queries stored in S_P. These SLVs are fetched individually (line 4) until no more query exists (3). The SLV query $< i, s, v >$ visits local statements from i to e, which is calculated at (7-8). j refers to the index of the current visited statement. j is calculated from the type of P and the current value of j (10).

When $< i, s, v >$ visits $P[j]$, we check whether $P[j]$ kills v. If it does not (14), then v is reproduced if $P[j]$ is an ingoing side of an interface (15). Otherwise (16), $P[j]$ should be sliced (18,19) and added to $\delta(s)$ (17). At this point, there are two modes; if $P[j]$ was already sliced in a previous slice (20) then the definitions stored in $\delta(P[j])$ should be sliced by the procedure TRCK(21). Otherwise, the second mode generates the variables used in $P[j]$ as SLV queries in S_P (23) and in $S_{P'}$ if there is $P'.\ell' \hookrightarrow P[j]$(25) . After every visit, S_P is updated according to the transfer function $f^{j,e}_{(i,s,v)}(S_P)$ shown in Fig.5e. This function has three cases: $< i, s, v >$ is removed from S_P, S_P is not updated and finally, $< i, s, v >$ is removed from S_P and the variables used in the current visited statement $P[j]$ are generated as SLV queries.

TRCK(ℓ, var, N_{slc}) (Fig.5d) traces *use-definition* chains from ℓ. This is performed by first slicing the definitions stored in $\delta(\ell)$. Then TRCK is called recursively to slice the definitions which are stored in each of these labels and so on. In the pool $\delta(\ell)$, we slice every label stored in $\delta(\ell)$ unless ℓ is a placeholder. In this case, we slice only the labels that influence var.

Suppose ℓ that exists in P is sliced. Since ℓ is control dependent on $P[0]$, $P[0]$ should be sliced, which in its turn is control dependent on the predicate of P parent and so on. In Fig. 5d, this hierarchical structure of control dependencies goes on until reaching the PCB representing the most outer PCB.

Fig.5d shows INTFC function. This function reproduces var from ℓ to $P'[k]$ if ℓ is an ingoing side of an interface connecting ℓ with $P'[k]$.

Finally, the role of SELECT function is in fetching individually the SLV queries from S_P. The role of PARENT(P) is in getting the parent PCB of P.

7 On-the-fly Interprocedural Slicing

In our previous work [1], we sketched an inter-procedural slicing algorithm for procedures having a single out formal argument and a single return statement. This section extends this method to be applied to real procedures, which have many out formal arguments and multiple return statements or points. Furthermore, the construction of PCBs in inter-procedural programs is formalized math-

ematically, the syntax of the model language is extended to accommodate inter-procedural programs, the special transfer function for call sites is improved, and the *super interface* concept is introduced.

a: SlicePCB$(P, \mathcal{I}, N_{slc})$

```
1  // P: current PCB. I:Interfaces
2  // N_slc:sliced labels
3  while S_P ≠ ∅ do
4    < i, s, v >:= Select(S_P) ;
5    j := −1;
6    if (P is C and i ≠ final(P)) then
7      e := i + 1
8    else e = 0;
9    repeat
10     switch j do
11       case j = −1 : j := i; break;
12       case j > 0 : j := j − 1; break;
13       case j = 0 : j := final(P) ;
14     if (v ∉ kill(P[j])) then
15       Intrfc(P[j], s, v, I);
16     else
17       δ(s) := {δ(s) ∪ (P[j], v)};
18       if (P[j] ∉ N_slc) then
19         N_slc := N_slc ∪ {P[j]};
20         if (δ(P[j]) ≠ ∅) then
21           Trck(P[j], v, N_slc);
22         else
23           S_P := f^{j,e}_{(i,s,v)}(S_P);
24           foreach x ∈ gen(P[j]) do
25             Intfc(P[j], P[j], x, I);
26         Cntrl(P, N_slc, I);
27       break;// Fetch new SLV
28   until j = e;
29 return N_slc;
```

b: Intfc$(\ell, src, var, \mathcal{I})$

```
1 foreach (P'[k] ↪ ℓ ∈ I) do
2   i = P'[k] ↪ ℓ;
3   if (var ∈ W_m(i)) then
4     W_m(i) := W_m(i)\{var} ;
5     S_P' := S_P' ∪ {(k, src, var)};
6 return;
```

c: Trck(ℓ, var, N_{slc})

```
1  foreach (ℓ', var') ∈ δ(ℓ) do
2    if s_ℓ ≠ skip ∧ s_ℓ ≠ in_child then
3      N_slc := N_slc ∪ ℓ' ;
4      Trck(ℓ', var', N_slc );
5      Cntrl(M(ℓ), N_slc, I)
6    else
7      if var' = var then
8        N_slc := N_slc ∪ ℓ' ;
9        Trck(ℓ', var', N_slc );
10 return;
```

d: Cntrl$(P, N_{slc}, \mathcal{I})$

```
1  repeat
2    if (P[0] ∈ N_slc ) then return;
3    N_slc := N_slc ∪ P[0];
4    if (δ(P[0]) ≠ ∅) then
5      foreach v ∈ gen(P[0]) do
6        Trck(P[0], v, N_slc );
7    else
8      S_P := S_P ∪ {(0, P[0], v)|v ∈
         gen(P[0])}
9      foreach v ∈ gen(P[0]) do
10       Intfc(P[0], P[0], v, I);
11   P := parent(P);
12 until P ≠ ∅;
13 return;
```

e: Transfer Function

$$f^{j,e}_{(i,s,v)}(S_P) = \begin{cases} S_P \backslash \{(i,s,v)\} & \text{if } (j = e \wedge v \notin kill(P[j])) \\ & \vee\ (v \in kill(P[j]) \wedge P[j] \in N_{slc}) \\ & \vee\ (v \in kill(P[j]) \wedge \delta(P[j]) \neq \emptyset) \\ S_P & \text{if } j \neq e \wedge v \notin kill(P[j]) \\ S_P \backslash \{(i,s,v)\} \cup \{(j-1, P[j], u) \mid u \in gen(P[j])\} & \\ \quad \text{if } v \in kill(P[j]) \wedge P[j] \notin N_{slc} \wedge \delta(P[j]) = \emptyset & \end{cases}$$

Fig. 5: Two-Mode Algorithms

The algorithm is restricted to non-recursive procedures. This is a common restriction in safety-critical applications.

To be able to study inter-procedural slicing we must extend the language with procedure declarations and procedure call. Procedure declarations consist of a name, zero or more formal arguments and a body. The formal arguments are declared to be either in arguments, that pass information into the procedure, or out arguments that in addition pass information from the procedure to the caller. The global variables are considered out arguments. Procedure calls are, without loss of generality, restrained to variables. Let F range over procedure names and let $\psi_{P[j]}$ denote the bijective function that maps the actual arguments at call site $P[j]$ to the formal arguments of the procedure. For a procedure F and its output formal parameter v, $\mu_F(v)$ refers to the set of formal arguments that might influence v. The extended syntax is defined as follows.

$$
\begin{aligned}
p \quad &::= d \mid d\ p \\
arg &::= \texttt{in}\ x \mid \texttt{out}\ x \\
d \quad &::= [\texttt{proc}\ F\ \overline{\texttt{arg}}\ s]^{\ell_F} \\
es \quad &::= \ldots \mid [\texttt{call}\ F\ \overline{x}]^{\ell} \mid [\texttt{return}]^{\ell}
\end{aligned}
$$

We extend the notion of interfaces from being between two labels to being a relation on sets of labels. For clarity we write ℓ for the singleton set $\{\ell\}$. Let $calls(F)$ be the set of labels of calls to F.

To compute the PCB-graph in the presence of procedures, the algorithm found in Section 3.1 is extended as shown in Figure 6.

$$
\begin{aligned}
\lambda_{\ell_p}([\texttt{call}\ F\ \overline{x}]^{\ell}) &= [\texttt{call}\ F\ \overline{x}]^{\ell}, (\emptyset, \emptyset) \\
\lambda_{\ell_p}([\texttt{return}]^{\ell}) &= [\texttt{return}]^{\ell}, (\emptyset, \{\ell \hookrightarrow calls(F)\})
\end{aligned}
$$

On the top-level

$$
\begin{aligned}
\lambda([\texttt{proc}\ F\ \overline{\texttt{arg}}\ s]^{\ell}) &= (\phi[\ell \mapsto \{true^{\ell} : es, L\}], \epsilon \cup \{calls(F) \hookrightarrow \ell\}) \\
&\quad \textit{where } es, (\phi, \epsilon) = \lambda_{\ell}(s) \\
\lambda(d\ p) &= (\phi_1 \cup \phi_2, \epsilon_1 \cup \epsilon_2) \\
&\quad \textit{where } es, (\phi_1, \epsilon_2) = \lambda(d) \textit{ and } (\phi_2, \epsilon_2) = \lambda(p)
\end{aligned}
$$

Fig. 6: Extension of PCB graph computation

The resulting algorithm introduces two inter-procedural interfaces: one going from the call sites to the entry label of the procedure (*Many-to-One* Interface) and one going from a return label to the call sites (*One-to-Many* Interface). Each of these two interfaces could be expressed by a set of a normal interfaces (single-

ton label to singleton label). Thus, *Many-to-One* and *One-to-Many* interfaces are referred to *Super Interfaces*.

The Super Interface comprises of many thin parts linked through a joint to a single thick part. Together with the single thick part, each thin part constitutes a normal interface. Hence, the thick part is shared between all normal interfaces contained in a super interface. This design allows some inter-procedural information to be shared between the different call sites of a procedure, whereas other information is exclusively linked to individual call sites

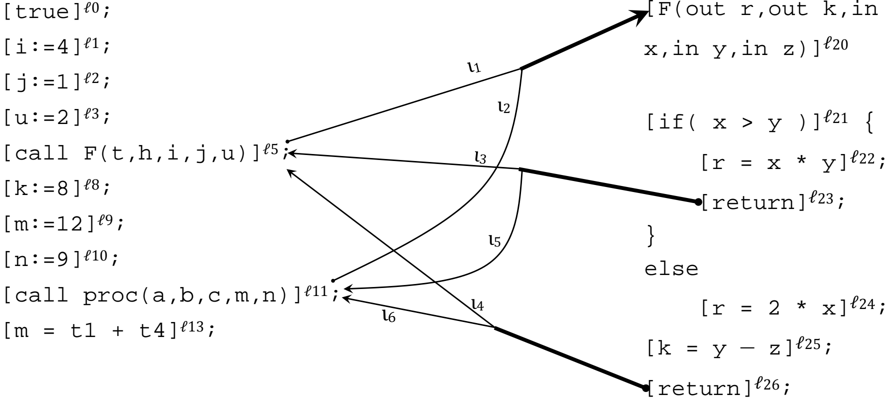

Fig. 7: Inter-procedural Example

In Fig. 7, a Many-to-One super interface is: $\ell_5, \ell_{11} \hookrightarrow \ell_{20}$. It contains two normal interfaces: $\ell_5 \hookrightarrow \ell_{20}$ and $\ell_{11} \hookrightarrow \ell_{20}$. Further, we have two One-to-Many interfaces: $\ell_{23} \hookrightarrow \ell_3, \ell_5$ and $\ell_{26} \hookrightarrow \ell_4, \ell_6$. Similarly, each could be expressed by two normal interfaces.

The shared thick part in a *Many-to-One* interface is linked with a μ_F definition for each out formal argument. The thin part holds a ψ_j^{-1} definition for each formal argument. In Fig. 7, the thin part of ι_1 associates with $\psi_{\ell_5}^{-1}(r) = t$, $\psi_{\ell_5}^{-1}(k) = h$, $\psi_{\ell_5}^{-1}(x) = i$, $\psi_{\ell_5}^{-1}(y) = j$, $\psi_{\ell_5}^{-1}(j) = u$. The thin part of ι_2 associates with $\psi_{\ell_{11}}^{-1}(r) = a$, $\psi_{\ell_{11}}^{-1}(k) = b$, $\psi_{\ell_{11}}^{-1}(x) = c$, $\psi_{\ell_{11}}^{-1}(y) = m$, $\psi_{\ell_{11}}^{-1}(j) = n$. The thick shared part of ι_1 and ι_2 contains $\mu_{\ell_{20}}(r)$ and $\mu_{\ell_{20}}(k)$.

The thick shared part in a *One-to-Many* super interface associates with a white list that contains initially all the out formal arguments. A thin part connecting a return statement to a j call site holds a ψ_j definition for each out actual argument at j. Based on that, the shared thick part in $\ell_5, \ell_{11} \hookrightarrow \ell_{20}$ associates with $R_m = \{r, k\}$. The thin part in ι_3 holds $\psi_{\ell_5}(t) = r$ and $\psi_{\ell_5}(h) = k$.

The SLV queries reproduced through super interfaces are processed in two stages, one over the thick part and the another in the thin part regardless of the order, before they reach the opposite sides. We indicate to this fact by saying

the variable is *thrown* on a thin or thick part. The *throwing* of a variable on a super interface part means that the variable is going to be processed according to the stage of this part.

There are two types of inter-procedural slicing. The first type is the *Up* slicing. Up slicing occurs when the procedure body is a source of SLV queries due to the existence of slicing criteria in it. Since any of call sites might run the procedure body, the procedure header is control dependent on all its call sites. Thus, the direction of SLV queries is from the procedure body to all its call sites.

The second type is the *Down* slicing. This type refers to the situation where a particular call site is the source of SLV queries in a procedure body. Therefore, for the statements which are sliced in the procedure body with respect of these SLV queries, the procedure header is control dependent on this call site. In other words, in down slicing, the SLV queries reach a procedure's header are not reproduced in all call sites. Instead, the transfer function shown in Sec. 7.1 is applied.

To maintain the context-sensitivity, we define two stacks, ST_{call} and ST_{var}. ST_{call} stores the last call site and ST_{var} stores the formal arguments used to compute μ_F.

7.1 The Transfer Function

In the down slicing, when the call site whose label is $P[j]$ is visited by an SLV query $< i, s, x >$, S_P has to be updated according to the following transfer function:

$$f^{j,F}_{i,e,x}(S_P) = \begin{cases} S_P & \psi_j(x) \text{ is undefined} \\ S_P \backslash \{(i,s,x)\} \cup \{(j-1, P[j], \psi_j^{-1}(u) \mid u \in \mu_F(\psi_j(x))\} & \mu_F(\psi_j(x)), \psi_j(x) \text{ defined} \end{cases} \quad (5)$$

The first case occurs when x is not an *out* actual argument at $P[j]$, so S_P is not updated. In the second case, x is an *out* actual argument at $P[j]$, $\mu_F(\psi_j(x))$ is computed. Thus, $< i, s, x >$ is killed and the actual arguments at $P[j]$ whose correspondent formal arguments belong to $\mu_F(\psi_j(x))$ are generated as SLV queries.

7.2 The Algorithm of Down Slicing and the Computation of μ_F

Suppose $P[j]$ is a call site of F and ι_r is an interface from $P[j]$ to a return statement in F and ι_h is an interface from F header to $P[j]$. The down slicing algorithm is:

1. When an SLV query $< i, s, x >$ visits a call site $P[j]$ and x is an out argument at $P[j]$, then:
   - if ι_h contains an already computed $\mu_F(\psi_{P[j]}(x))$, then we go to 9
     Note: ι_h and ι_r are known from the call site side.
   - Otherwise:
     - x is thrown on the thin part of ι_r.

  - The analysis is completely frozen on $P[j]$ side.
2. if the thin part of ι_r receives a variable x, then we get $\psi_{P[j]}(x)$, say v. Then, v is pushed in ST_{vars} and $P[j]$ is pushed in ST_{calls}. Finally, v is thrown on the thick part of ι_r.
3. At the thick part of ι_r, v is removed from $W_m(\iota_r)$ and it is reproduced in the opposite side of ι_r (a return statement in F).
4. F's body is sliced with respect to the SLV queries of its PCBs. When no more SLV query is alive in F's PCBs, we move to the thick part of the many-to-one interface.
5. In the thick part of the many-to-one super interface:
   - We read from the top of ST_{vars} the formal argument which F is sliced with respect to. In other words, we retrieve v. Then we redefine $\mu_F(v)$ held by the thick part from the dependencies from v at return statements to F's formal parameters located in F's header. The edges of PDDG can be tracked to find these dependencies.
   - We retrieve from the top of ST_{calls} the call site, which is $P[j]$. From $P[j]$ we find ι_h. Then we move to the thin part of ι_h.
6. By using the $\psi^{-1}_{P[j]}$ definitions held by the thin part of ι_h, we reverse each formal argument in $\mu_F(v)$ to its actual argument at $P[j]$.
7. ST_{var} and ST_{calls} are popped.
8. P is released from being frozen.
9. The transfer function in Eq. 5 is applied and computed from $\mu_F(v)$ and $\psi^{-1}_{P[j]}$ definitions held in ι_h.
10. The analysis goes on.

8 Results and Discussions

To measure the efficiency of the proposed approach, we have implemented four algorithms:

- **A**: implements the proposed methods in this paper; two-modes PCB-based algorithm, PDDG and filtering SLVs through shrinking sets.
- **B**: is an implementation of the original PCB-based slicing [1]. It produces single slices, filters SLVs by whitelists, but it does not implement PDDG.
- **C**: PDG-based slicing [11]
- **D**: is an implementation of the original PCB-based slicing [1]. It produces single slices, filters SLVs by blacklists and it does not implement PDDG.

These algorithms are implemented by using Microsoft Visual C++ 2013. The experiments have been run on an Intel Core i5 with a 2.66GHz processor, 8 GB RAM, and 64-bit operating system.

8.1 The Efficiency of Using PDDG and Two-Mode Algorithm

To setup the comparisons, a synthetic program is produced automatically. It is an intra-procedural program, $125K$ statements, the predicates constitute 26%

No.Slice	Slice Size	(A) ms	(B) ms	Speedup	(C) ms
1	69%	234	180	0,8	12204
2	33%	468	108	0,2	1
3	14%	1	54	54	1
4	15%	1	54	54	1
5	51%	234	144	0,6	1
6	21%	1	72	72	1
7	79%	18	216	12	1
8	58%	1	180	180	1
9	40%	1	108	108	1
10	37%	1	108	108	1
11	45%	1	126	126	1
12	10%	1	18	18	1
13	20%	1	72	72	1
14	39%	1	108	108	1
15	50%	1	144	144	1
sum		965	1692		12204

Table 1

of the program, and it has 50 variables. This program has to be sliced by (A), (B) and (C) according to 15 distinct and individual slicing criteria.

In Table 1, the first entry shows the times for computing the first slice. Then, for each subsequent slice, the additional time for computing this slice is shown. The last entry shows the total times for computing all 15 slices. The second column gives the size of each slice relative the size of the whole program.

(B) computes every slice individually from scratch. (B) shows that as slices gets bigger, more computations are performed and intuitively more execution time is consumed. (C) shows that for PDG-based slicing, the first slice is the heaviest than others with respect to the execution time. Afterwards, very little time is consumed to compute each slice.

most the PDDG is constructed while (A) computes the slices: (1,2,5). Afterwards, the speedups (B / A) shown in the fifth column vary from 4.8 to 156 for the slices from 5 to 15. Hence, the data dependencies that are accumulated in (1,2,5) are used in computing the slices from 6 to 15. The main advantage of the two-modes slicing algorithm is that it does not need a full comprehensive analysis of the program at the beginning. As well, it does not lose previous computations.

In comparing (A) and (C), we find that both of them depend on a graph form to retrieve their dependencies. By comparing their results, we find that for the slices from 6 to 15, the execution times are very close together, which is because for both algorithms the slicing mainly turns into a backwards search in a dependence graph. Finally, the last row accumulates the execution times

obtained by each implementation. The results indicate that the two-modes slicing perform significantly better than our previous PCB-based algorithm when computing many slices for the same code, and that also PDG-based slicing is outperformed for a moderate number of slices.

8.2 Whitelist vs Blacklist

The two local implementations (B and D) are used to measure the results of the change from blacklists to whitelists. To do so, six synthetics source code programs were produced to be analyzed by (B) and (D). These six programs differ in their number of variables, which varies from 25 to 800. To ensure fair comparisons, other factors which could influence execution times are fixed. Thus, the size of each synthetic program is $50K$, the number of predicates in every program is around 6500 and the slice size is 70% from the total size.

No. Var.	25	50	100	200	400	800
Blacklists - D (s)	0.10	0.35	1.85	10.85	64	424
Whitelists - B (s)	0.014	0.052	0.083	0.146	0.25	0.48
Speedup (D/B)	7.1	6.7	22.2	74.3	256	883

Table 2: Whitelists *vs* Blacklists

There are two facts can be easily read from Table. 2, the first is that using whitelists enhances significantly the performance of the analysis. The second, which is more important than the first, shows that the superiority of the whitelists increases as more variables are added to the source code. While more variables are added to program, thus generating more SLV queries, moving more SLV queries between the PCBs and performing more operations through black and white lists. Since the blacklists sizes are proportional to the number of the programs' variables and on the other side, the whitelists sizes are proportional to the PCBs' variables, the execution times of (D) shows much more higher sensitivity than (B) to the change of the number of programs' variables.

9 Related Work

Program slicing was first introduced by Weiser [13] in the context of debugging. Ottenstein et al. [9, 11] introduced the PDG, and proposed its use in program slicing. PDG-based slicing has then been the classical program slicing method. Horwitz et al. [17] extended the PDG to as System Dependence Graph to capture calling context of procedures.

Hanjal and Forgàcs [6] propose a complete demand-driven slicing algorithm for inter-procedural well-structured programs. Their method is based on the propagation of tokens over Control Flow Graph (CFG). Harrold and Ci [16] propose a demand-driven method that computes only the information which

is needed in computing a slice. Harrold's method is based in Augmented CFG (ACFG). Furthermore, there are more special works for computing interprocedural information in a demand-driven way [18, 19].

10 Conclusions and Future Work

We have shown how to extend our previous algorithm for demand-driven slicing of well-structured programs [1] into an algorithm that can efficiently compute several slices for the same program. The main mechanism for achiveing good performance is to store and reuse previously computed data dependencies across several slices. We also make some clarifications regarding the original algorithm [1], including a formal description how the underlying PCB program representation is computed from the program code, and a description of how the "filtering" of Strongly Live Variables at interfaces can be implemented efficiently.

An experimental evaluation indicates that the new two-mode algorithm, with stored and reused data dependences, performs considerably better than the previous version when taking several slices of the same code. It also performs significantly better than the standard, PDG-based algorithm in the experiment

There are a number of possible future directions. One direction is to directly apply the slicing algorithm to speed up the verification of safety-critical software. For instance, SPARK Ada programs are often filled with assertions to be checked during the verification process. Formal methods for checking assertions, like symbolic execution [20], can be very prone to path explosion: slicing with respect to different slicing criteria derived from the assertions can then help to keep the complexity under control. A second direction is to generalise the slicing approach to richer languages including procedures, pointers, and object-oriented features, and to gradually relax the requirements on well-structuredness. A final observation is that the SLV analysis performed by our algorithm provides a pattern to perform other dataflow analyses, like Reaching Definitions and Very Busy Expressions [10], efficiently on well-structured code.

11 Acknowledgments

The authors thank Daniel Hedin, Daniel Kade, Iain Bate and Irfan Sljivo for their helpful comments and suggestions. This work was funded by The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC), and the Swedish Foundation for Strategic Research under the SYNOPSIS project.

References

1. Khanfar, H., Lisper, B., Abu Naser, M.: Static Backward Slicing for Safety Critical Systems. ADA-Europe 2015, The 20th International Conference on Reliable Software Technologies, , 9111 (50-65), June 2015

2. Lisper, B., Masud, A.N., Khanfar, H.: Static backward demand-driven slicing. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 115–126. PEPM '15, ACM, New York, NY, USA (2015)
3. Yunja Choi, Mingyu Park, Taejoon Byun, Dongwoo Kim: Efficient safety checking for automotive operating systems using property-based slicing and constraint-based environment generation. Sci. Comput. Program. 103: 51-70 (2015)
4. Kraft, J.: Enabling Timing Analysis of Complex Embedded Software Systems. Ph.D. thesis, Mälardalen University Press (August 2010)
5. Ákos Hajnal and István Forgács: A precise demand-driven definition-use chaining algorithm, Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on, Budapest, 2002, pp. 77-86. doi: 10.1109/CSMR.2002.995792
6. Ákos Hajnal and István Forgács: A demand-driven approach to slicing legacy COBOL systems",Journal of Software Maintenance,volume 24, no. 1, pages67–82, 2012
7. Agrawal, G.: Simultaneous demand-driven data-flow and call graph analysis, in Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on Software Maintenance, ICSM 1999,pages 453–462.
8. Agrawal, H.: On slicing programs with jump statements. SIGPLAN Not. 29(6), 302–312 (Jun 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 (Jul 1987)
10. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2nd edition. Springer (2005), iSBN 3-540-65410-0
11. Ottenstein, K.J., Ottenstein, L.M., The program dependence graph in a software development environment. SIGSOFT Softw. Eng. Notes 9(3), 177–184 (Apr 1984)
12. Tip, F., A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
13. Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering, 352–357 (Jul 1984)
14. Dimov Aleksandar and Chandran, Senthil Kumar and Punnekkat, Sasikumar: How Do We Collect Data for Software Reliability Estimation, 11th International Conference on Computer Systems and Technologies,CompSysTech '10,2010, ACM.
15. Eric Larson: Assessing Work for Static Software Bug Detection, 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies, WEASELTech '07, 2007, ACM NewYork
16. M. J. Harrold and N. Ci, Reuse-driven interprocedural slicing. In The 20th International Conference on Software Engineering, pages 7483, Apr. 1998.
17. S. Horwitz, T. Reps, and D. Binkley: Interprocedural slicing using dependence graphs". ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 12 Issue 1, Jan. 1990, 26-60. ACM New York, NY, USA
18. E. Duesterwald and M. L. Sofifa: Demand-driven computation of interprocedural data flow. In Proceedings of 22nd ACM Symposium on Principles of Programmzng Languages, pages 37-48, January 1995.
19. S. Horwitz, T. Reps, and M. Sagiv: Demand interprocedural dataflow analysis. SIGSOFT '95 Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, pages 104-115, 1995. ACM New York.
20. J.C. King: Symbolic Execution and Program Testing, Communications of the ACM, Vol: 19, No. 7, July 1976, pp. 385-394. ACM New York