Mälardalen University Licentiate Thesis 284

DEMAND-DRIVEN STATIC BACKWARD PROGRAM SLICING BASED ON PREDICATED CODE BLOCK GRAPHS

Husni Khanfar

MÄLARDALEN UNIVERSITY
SWEDEN

Mälardalen University Press Licentiate Theses
No. 284

DEMAND-DRIVEN STATIC BACKWARD PROGRAM SLICING BASED ON PREDICATED CODE BLOCK GRAPHS

Husni Khanfar

2019

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISBN 978-91-7485-440-4
ISSN 1651-9256
Printed by E-Print AB, Stockholm, Sweden

Abstract

Static backward program slicing is a technique to compute the set of program statements, predicates and inputs that might affect the value of a particular variable at a program location. The importance of this technique comes from being an essential part of many critical areas such as program maintenance, testing, verification, debugging, among others. The state-of-art slicing approach collects all the data- and control-flow information in the source code before the slicing, but not all the collected information are used for computing the slice. Thus, this approach causes a significant amount of unnecessary computations, particularly for slicing large industrial systems, where unnecessary computations lead to wastage of a considerable amount of processing time and memory. Moreover, this approach often suffers from scalability issues.

The demand-driven slicing approaches aim at solving this problem by avoiding unnecessary computations. However, some of these approaches trade precision for performance, whereas others are not entirely demand-driven, particularly for addressing unstructured programs, pointer analysis, or inter-procedural cases.

This thesis presents a new demand-driven slicing approach that addresses well-structured, unstructured, and inter-procedural programs. This approach has four distinct features, each of which prevents a special type of unnecessary computations. The effectiveness and correctness of the proposed approach are verified using experimental evaluation. In addition, the thesis proposes an approach that can compute on the fly the control dependencies in unstructured programs.

Sammanfattning

Static backwards program slicing är en teknik för att hitta alla de programsatser, predikat och indata som möjligen kan påverka värdet av en viss variabel i någon given programpunkt. Teknikens betydelse kommer av att den är en viktig del inom många kritiska tillämpningar som till exempel mjukvaruunderhåll, testning, verifiering, och debuggning. Ett problem med de nuvarande metoderna för slicing är att de beräknar alla data- och kontrollberoenden, för hela programmet, innan slicingen äger rum. Därför utför dessa metoder ofta onödigt arbete, speciellt när man slicar stora industriella system där detta då kan leda till ett slöseri med processortid och minne. Dessutom lider dessa metoder ofta av skalbarhetsproblem.

Demand-driven slicing syftar till att lösa detta problem genom att undvika onödiga beräkningar. En del av dessa metoder betalar dock den ökande prestandan med minskad precision, medan andra inte är helt "demand-drivna" speciellt vad beträffar ostrukturerade program, pekaranalys, eller interprocedurell analys.

Denna avhandling presenterar en ny metod för demand-driven slicing som hanterar både strukturerade och ostrukturerade program med procedurer. Denna metod har fyra utmärkande egenskaper, där var och en förhindrar en viss typ av onödig beräkning. Effektiviteten och korrektheten hos den föreslagna tekniken har verifierats genom experiment. Dessutom presenterar avhandlingen en metod som är demand-driven för att beräkna kontrollberoendena i ostrukturerade program.

To My Family

Acknowledgements

First and foremost, I wish to express my appreciation to my supervisors Björn Lisper, Hans Hansson, Saad Mubeen and Daniel Hedin for guiding me in the past years. Without your patient, feedback and continuous support, there is no possible for this thesis to appear. During my work in MDH, I met many great people who deserve to be thanked, Sasikumar Punnekkat, Iain Bate, Jan Calrsson, Jan Gustafsson, Radu Dobrin, Damir Isovic, and Christer Norström.

It is pretty beautiful to be surrounded by many people who do not forget me in their daily praying wherever I am. From the bottom of my heart, I thank my affectionate mother and generous father for their unlimited support at every moment in my life. I also thank my brothers and sisters Maysoon, Aws, Tasneem, Rowaa, Besher and Baraa for encouraging me to continue my high studies in Sweden.

I must express my gratitude to Esraa, my wife. Thanks to accompanying me away from your family to support me in the continuation of my high studies. Ooh, I can not leave out mentioning the most important people, my kids Aws, Maryam, Raghad, Zayd and Tamim, sorry guys for being busy. My office mates in MDH are considered as part of my family members, all the thanks to Omar Jaradat, Irfan Sljivo, Gabriel Companeu and Filip Markovik.

The work in this thesis has been supported by the Swedish Foundation for Strategic Research (SSF), The Knowledge Foundation and VINNOVA via projects SYNOPSIS, SPACES and ASSUME, respectively.

Husni Khanfar
Västerås, June 13, 2019

List of Papers Included in the Thesis

This thesis is based on the following papers:

Paper A Static backward program slicing for safety-critical systems: Husni Khanfar, Björn Lisper, and Abu Naser Masud. *Ada-Europe International Conference on Reliable Software Technologies*, pages 50–65, Springer, 2015.

Paper B Enhanced PCB-based slicing: Husni Khanfar and Björn Lisper. *Fifth International Valentin Turchin Workshop on Metacomputation*, pages 71–91, 2016

Paper C Demand-driven static backward slicing for unstructured programs: Husni Khanfar, Björn Lisper, and Saad Mubeen, Technical Report MDH-MRTC-324/2019-1-SE, May 2019.

List of Papers Not Included in the Thesis

- Static backward demand-driven slicing: Björn Lisper, Abu Naser Masud, and Husni Khanfar. *InProceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, ACM (pages 115–126), Jan 2015.

Reprints were made with permission from the publishers.

Contents

List of Tables

List of Figures

Part I:

Thesis

1. Introduction

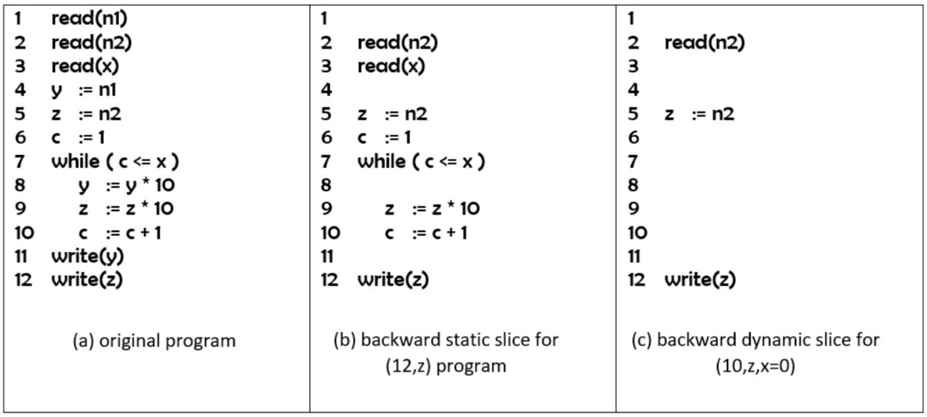
Program slicing [1] is a static program analysis technique that decomposes a program based on a program's property. Slicing identifies the program parts that affect that property and produces from those parts a smaller program that still produces that property. The reduced program is called "slice". Program slicing techniques aims to find the minimal form of the slice. Program slicing is an essential part of many areas such as the program understanding, maintenance, debugging, verification, testing and much more. This technique is classified to backward and forward.

Backward program slicing is a technique that identifies the set of program statements, predicates[1], and inputs that might influence a slicing criterion, which is a pair of $<$*loc,var*$>$, wherein (*var*) is the value of a variable at a program location (*loc*). *Forward program slicing* [2] identifies the program statements that are influenced by a slicing criterion. This feature makes it suitable for removing *dead code* and maintaining source codes [3]. A statement can be dependent on another statement due to data or control dependence. For example, a statement s_2 is data dependent on a statement s_1 if a variable updated or assigned by the execution of s_1 might be read by the execution of s_2. A statement s_2 is control dependent on the predicate s_1 if the execution of s_1 decides the possible execution of s_2.

Weiser [1], who was the first to define the program slicing, used the term *executable* slice to define a slice that can be compiled and run. Such slices give results on the program points identical to those in the original program in the case of using the same input values. The contrast is *non-executable* slice. A slice is said to be *static* if it does not consider any predefined value of any input variable or any particular execution. On the other hand, *dynamic slicing* takes into account a specific execution, where input variables are predefined. Since the value of the predicates in the dynamic slicing are determined, the size of dynamically computed slice is often smaller than the corresponding statically computed slice for the same slicing criteria [3]. In the perspective of the number of procedures in the program, *intra-procedural slicing* analyses programs that have strictly one procedure. On the other hand, *inter-procedural slicing* handles multiple-procedure programs.

Figure 1.1(a) demonstrates an example of a program that computes the values of y and z, where y is the product of an integer $n1$ and 10 to the power of variable x. Similarly, z is the product of an integer $n2$ and 10 to the power

[1]Predicates appear in conditional statements.



```
1    read(n1)
2    read(n2)
3    read(x)
4    y  := n1
5    z  := n2
6    c  := 1
7    while ( c <= x )
8        y  := y * 10
9        z  := z * 10
10       c  := c + 1
11   write(y)
12   write(z)
```

(a) original program

```
1
2    read(n2)
3    read(x)
4
5    z  := n2
6    c  := 1
7    while ( c <= x )
8
9        z  := z * 10
10       c  := c + 1
11
12   write(z)
```

(b) backward static slice for (12,z) program

```
1
2    read(n2)
3
4
5    z  := n2
6
7
8
9
10
11
12   write(z)
```

(c) backward dynamic slice for (10,z,x=0)

Figure 1.1: Slicing example

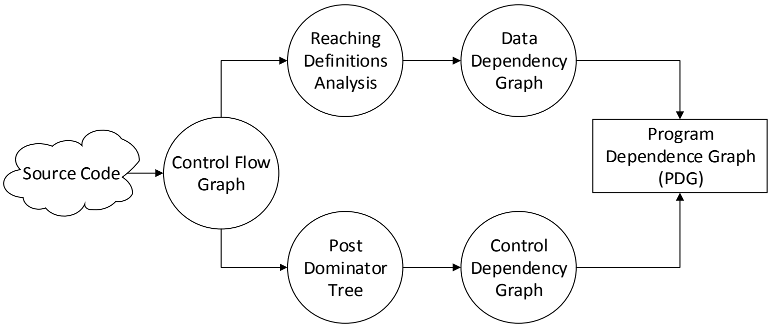


Figure 1.2: The Stages of constructing a Program Dependence Graph

of variable x. Hence, this example implements the equations $y := n1 * 10^x$ and $z := n2 * 10^x$. The backward static slice of this program concerning the slicing criterion $< 12, z >$ is shown in Figure 1.1(b). The backward dynamic slice for the same slicing criterion, when the value of the program input x is 0, is shown in Figure 1.1(c).

1.1 Program Dependence Graph Based Slicing

The first essential step that is required to analyse a program is in decomposing its source code lines to extract some basic information. Afterwards, that information is organised in a graph format. Next, such forms make the base for other analyses to build on these algorithms. These graphical forms are called program representations.

The primary three known representations used by the static program analysis[2] methods are Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). AST is used in abstracting the hierarchical syntactic structure of the software. It represents every program statement as a sub-tree, where the node is the operator, procedure name, or condition, while the leaves represent operands, argument lists, an expression of a condition (predicate), or inner statements inside the body of the condition. The sub-trees are ordered from left to right with respect to their locations in the source code. The AST does not represent explicitly the control flows in the well-structured source codes, and it does not represent at all the control flows in unstructured source codes.

The CFG represents program statements as nodes and the transfer of controls as edges. Representing the control flows in the CFGs enabled it to be the best choice for the early scientists who developed the basis of the state-of-the-art slicing techniques [4, 5, 6, 7, 8, 9]. The CFG for an intra-procedural program P is a 4-tuple $(N, E, Entry, End)$:

1. N is a set of nodes, where each node represents an elementary program statement in P.

2. E is a set of edges, where each edge represents a possible program flow from one node to another. $E \subseteq (N \times N)$.

3. $Entry$: is a unique start node. $Entry \in N$.

4. End: is a unique exit node. $End \in N$.

The programs that are represented by a CFG must have the following features:

- There is a path from $Entry(P)$ to every $n \in N$.

- There is a path from every $n \in N$ to End.

Figure 1.3 shows the CFG of the program in Figure 1.1(a).

The analyses in this thesis are achieved using a simple programming language called While [10]. This language comprises statements of assignments to local variables, if conditional statements, while loops, and simple integer and boolean expressions. In the syntax of the While language, some types of program pieces are indicated by metavariables as shown below, where the letter on the left is a metavariable, and the kind of its program piece is described on the right:

[2]Static program analysis methods are the approaches that are used to understand the behavior of a software program without running it. Static program slicing is one of these methods.

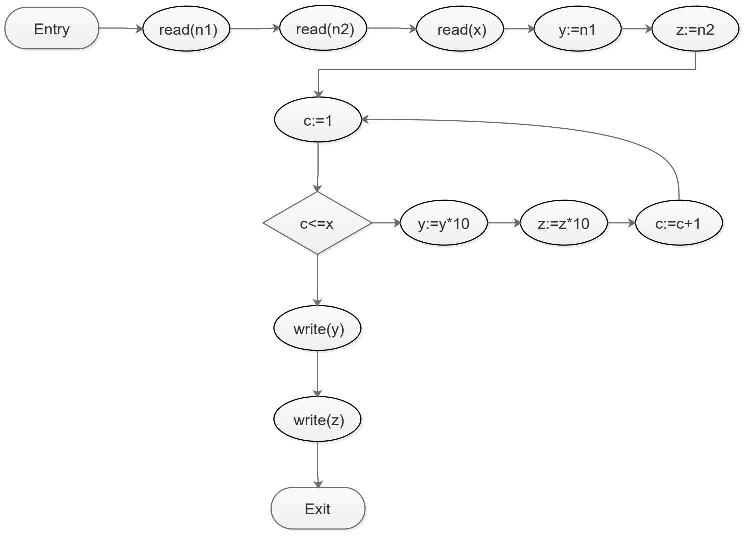


Figure 1.3: The Control Flow Graph (CFG) of the program shown in Figure 1.1(a)

s	statement
a	arithmetic expression
x	program variable
n	numerical constant
b	boolean predicate

The syntax of the While language is as follows:

$$\begin{array}{ll} s & := [x := a]^\ell \mid [\text{skip}]^\ell \mid s_1;s_2 \mid \text{If } b^\ell \text{ then } s_1 \text{ else } s_2 \mid \text{while } b^\ell \text{ do } s \\ a & := x \mid n \mid a_1\ op_a\ a_1 \\ b & := \text{true} \mid \text{false} \mid \text{not b} \mid b_1\ op_b\ b_2 \mid a_1\ op_a\ a_2 \\ op_b & := < \mid <= \mid >= \mid > \mid \text{and} \mid \text{or} \\ op_a & := + \mid - \mid * \mid \div \end{array}$$

In this syntax, each elementary statement (e.g. $x := a$) or a predicate of a conditional statement gets a unique label (ℓ). These labels correspond to the nodes in the CFG that is constructed for the While program.

In the state-of-the-art slicing technique, the CFG is subjected to many algorithms to obtain all the data and control dependencies in the program. Those dependencies are added as edges to the CFG. This addition of dependence edges upgrades the CFG to a PDG. Figure 1.2 shows how the PDG is constructed from a CFG. First, the source code is converted into a CFG. Second, the Reaching Definition (RD) dataflow analysis[3] is applied to the CFG to determine for every program the definitions that reach this point. These analyses construct use-def chains[4], which are essential to finding out the data dependencies. Third, another algorithm is applied to the same CFG to build a post-dominator tree[5], which is used later to take out the control dependencies. Since each data or control dependence reflects a direct relationship between two nodes in the CFG, such dependencies are translated to edges, encoded in the CFG to produce a PDG.

The PDG is used in the state-of-the-art slicing approach to slice the program for particular slicing criteria. To do so, the reachability analysis slices each node in the PDG that reaches the slicing criterion node by a sequence of program dependence edges [4]. As an example, Figure 1.4 shows the PDG of the source code shown in Figure 1.1(a). In Figure 1.4, all the nodes that could reach the slicing criterion $< 10, p >$ are shaded, and their corresponding statements form the slice shown in Figure 1.1(b).

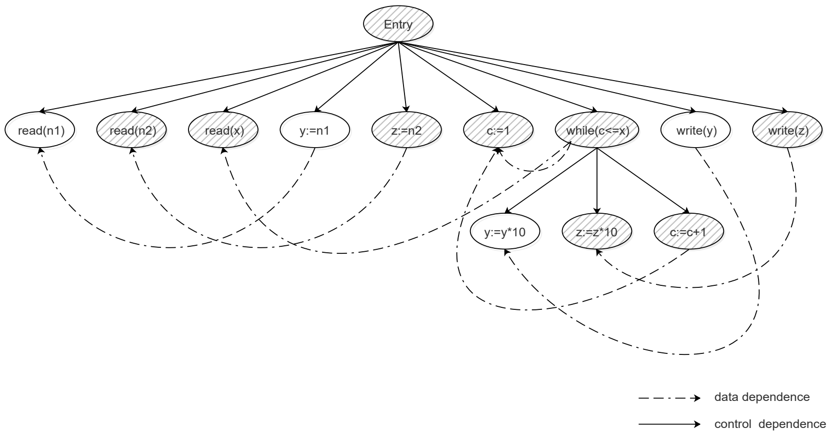


Figure 1.4: The PDG of the program in Figure 1.1

1.2 Reaching Definitions and Data Dependencies

Dataflow analysis techniques rely on generating, propagating and killing data queries to explore the source code. Each of these techniques works to find out a special type of facts by propagating a particular type of data queries. Dataflow analyses are designed to use the nodes and edges in the CFG. *Forward dataflow analyses* propagate the data queries with the direction of the

[3]Sec. 1.2 provides a discussion on the RD analysis

[4]use-def chain concerning variable x used in s_1 refers to the definitions defining x and reaching s_1.

[5]The post-domination concept is introduced in Sec. 1.3.

edges, while *backward dataflow analyses* propagate them against the direction of the edges. Each node n in the CFG has two program points; *entry(n)* and *exit(n)*. In the forward dataflow analyses, *entry(n)* is immediately before n, and *exit(n)* is immediately after it [10].

Dataflow analyses track the movement of their queries by keeping a copy of each query at every program point that it reaches. Thus, a dataset is associated with every point. The denotations of $S_{entry}(n)$ and $S_{exit}(n)$ refer to the sets of *entry(n)* and *exit(n)*, respectively. In the forward dataflow analyses, $S_{entry}(n)$ is computed from the exit point sets of the predecessors of n, while the value of $S_{exit}(n)$ is computed from n as well as $S_{entry}(n)$. Such a way in tracking the data flow facts adds many operations in the level of each data set (e.g. checking, removing and copying) and in the level between the data sets (e.g.intersecting and unifying). Further, this method requires extra memory space to be available, which becomes considerable with slicing large industrial systems. However, some techniques can reduce the consumption of resources. One of them is based on using bit vectors[6].

Since the values of the program sets are strongly connected, changing the value of any set might cause a chain of changes to other sets. Thus, the dataflow analysis usually requires many iterations until reaching the fixed-point status[7], where all the sets become saturated. At this moment, it is not possible to add more queries to any data set.

The dataflow analyses can be classified into MAY and MUST analyses. MAY dataflow analyses compute the entry point set as a *union* of the predecessor exit point sets, whereas it is an *intersection* from the same predecessor sets in MUST analyses. The equations of MAY dataflow analyses that compute the $S_{entry}(n)$ and $S_{exit}(n)$ for a node n in a CFG are formed as [11, 10, 12]:

Reaching Definition (RD) dataflow analysis is classified as forward and MAY. It generates RD data queries from the assignments (definitions) and propagates them forward to recognise every program point that each assignment might reach [10]. The RD data query is a pair $< v,n >$, where v is a variable defined in a program node n. Based on that, if $< v,n >$ is stored in $S_{entry}(n')$, then this means that v is defined at n and there is a path from n to n'

[6]In the bit-vector data flow analysis, the data set is represented by a bit-vector, where one bit is allocated for each data query. Thus, the size of the bit-vector equals the number of data queries in the program. The simple OR and AND bit operations are used to compute the union and intersection operations of bit vectors.

[7]The iterated function is a function that is applied a certain number of times $f^0, f^1, f^3..f^n$, where the output of each function f^n is an input to the next one f^{n+1}. So, the function itself is composed with itself and produces a sequence of values where $x_{n+1} = f(x)$. The fixed-point status in iterated functions occurs when $x_{n+1} = x$. In this case, the output of f becomes steady, and there is no use from applying f more than $n+1$ from the same initial status. Since dataflow analyses apply their equations many times until they become satisfied, we consider these equations as iterated functions. Also, since such data sets are subsets of a finite number of variables and locations, dataflow analyses are indeed fixed-point iterative techniques.

that does not overwrite v. In this case, providing n' uses v, meaning that there is a flow of data from n to n', and n' is data dependent on n.

In dataflow analyses, the nodes generate and kill data queries by a set of functions that are used in dataflow equations such as what is found in Equation 1.1. RD functions are formed in (1.2):

$$\begin{array}{lcl} S_{entry}(Entry) & = & S_{init} \\ S_{exit}(Entry) & = & S_{entry}(Entry) \\ S_{exit}(End) & = & S_{entry}(End) \\ S_{exit}(n) & = & (S_{entry}(n) \setminus kill(n)) \cup gen(n), \\ & & \text{ where } n \notin Entry, n \notin End \\ S_{entry}(n) & = & \bigcup_{n' \in pred(n)} S_{exit}(n'), \\ & & \text{ where } n \notin \{Entry\} \\ & & \text{ and } pred(n) \text{ is the set of } n \text{ predecessors} \end{array} \tag{1.1}$$

The time complexity of the RD analysis is $O(t \cdot h \cdot |N|)$ [12], where h is the size of the largest possible set at a program point and $|N|$ is the number of nodes in the CFG. $h \cdot |N|$ is the maximum possible number of fixed-point iterations. The expression $O(t \cdot h \cdot |N|)$ is the worst execution time, where t is the maximum time needed to perform one fixed-point iteration.

$$\begin{array}{rcll} S_{init} & = & \{(x,?) | x \text{ is a program variable}\} & \\ & & & \textit{where (x, ?): the ? refers to the Enty node.} \\ gen([x := a]^{\ell}) & = & \{(x,\ell)\} & \\ kill([x := a]^{\ell}) & = & \{(x,\ell') | \ell' \in N\} \cup \{(x,?)\} & \\ & & & \textit{where N is the set of nodes in the CFG} \\ kill([b]^{\ell}) & = & \emptyset & \textit{where b is a predicate} \\ gen([b]^{\ell}) & = & \emptyset & \textit{where b is a predicate} \end{array} \tag{1.2}$$

1.3 Post-domination and Control Dependency

In the CFG, a node n post-dominates another node n' if all the paths from n' to the End node contain n. The importance of the post-domination information is in being the basis of defining the control dependence relationship as follows: a node n is control dependent on a node b iff b is a predicate, there is a path in the CFG from b to n, wherein n post-dominates all the nodes in it except b, and there is a path from b to the End node, which does not contain n.

In the state-of-art slicing approach [4], a post-dominator tree is built to conclude from it all the control dependence facts in the program. The time complexity of all the algorithms constructing post-dominator trees is strongly related to the number of nodes in the corresponding CFG. Cooper et al. [13] described the algorithm of Lengauer and Tarjan [14] as the best-known algorithm that builds a post-dominator tree, and it is almost linear. Based on that, the time complexity of building a post-dominator tree by any algorithm is directly proportional to the number of the nodes in the CFG or the program size.

1.4 System Dependence Graph (SDG)

The PDG represents program statements, inputs, predicates and dependencies in a one-procedure program. It cannot work with multiple-procedures programs. To address this problem, Horwitz, Reps, and Binkley (HRB) extended the PDG to a so-called System Dependence Graph (SDG) [6]. The SDG represents each procedure by a PDG, and it also adds to those PDGs new vertices and edges to represent the call sites, procedure headers, formal parameters, actual parameters, the parameters passing between the call sites and their procedures. So, the leading role of the SDG is to link the different PDGs properly.

The SDG introduces new types of vertices and edges. In the SDG, the call site is represented by a *call vertex*. The actual parameter is represented by an *actual-in* vertex, and if its value might be updated due to the calling of the procedure, then this parameter is also represented by an *actual-out* vertex. An *entry vertex* represents a procedure header. A *formal-in* vertex represents the formal parameter, and if this parameter corresponds to an actual-out parameter, then it is also represented by a *formal-out vertex*. Regarding the edges, each actual vertex is control dependent on its call vertex. Similarly, each parameter vertex is control dependent on its entry vertex. Control edges represent these control dependencies.

The SDG is formed by PDGs that are connected by *call edges*, *formal-in edges*, and *formal-out edges*. The call edge connects the call site vertex with its corresponding entry vertex. The parameter-in and parameter-out edges represent parameter passing. Parameter-in edges run from actual-in vertices to their corresponding formal-in vertices. Parameter-out edges run from formal-out edges to their corresponding actual-out vertices.

Transitive dependence appears in a call site between two of its actual parameters, suppose x and y, if the value of y is updated due to the calling and it might be affected by x. In this case, we say that y is a transitive dependence on x. This dependency is represented by a summary edge from the actual-in vertex of x to the actual-out vertex of y. The summary edges help the slicing approach not to dig into the called procedure whenever one of its call sites is encountered.

Similar to the PDG-based slicing, the SDG-based slicing approach uses the reachability analysis, which tracks backward the dependence edges from the vertex of the slicing criterion. This leads to the *Calling Context Problem*. This problem occurs with the procedures that are called by many calling sites. When the analysis encounters a call site *cs* vertex of a procedure P, the analysis descends into P, slices it, and then ascends to *cs* again. Since P's formal-out parameters are connected with all the actual-out parameters in all the call sites of P, the calling context problem occurs when the analysis might ascend to another call site or to all the call sites. Therefore, preserving the calling context is essential to produce precise slices.

The SDG-based backward slicing applies the reachability analysis in two phases. The first phase uses the data, control, summary and parameter-in edges to track backwards the dependence edges, but not along parameter-out edges. The second phase uses all the types of edges except parameter-in edges. Based on that, providing the slicing criterion is located in a procedure p, the first phase starts from p for this criterion, then it ascends to every procedure calling p, but it does not descend into the procedures called by p. The call sites that are encountered in the first phase are marked. The second phase starts its analysis from the marked procedures, and it descends into every called procedure, but the choice of edges in the second phase limits the traversal from ascending into calling procedures.

The two-phase interprocedural slicing is designed to avoid the context-insensitive problem. This problem occurs when the procedure p is reached from an actual-out y vertex for a call site and exit to an actual-in vertex x' for another call site. The essence of this problem comes from creating the false fact that y is transitive dependent on x'. The context-insensitive problem happens due to the existence of invalid dependence paths, which are formed due to connecting the formal parameter vertices with all corresponding actual parameters in the different call sites. Two-phase SDG-based slicing prevents this problem, because the first phase ascends to all the calling procedures. The second phase slices the called procedures without going up again to their call sites. Consequently, the procedure which is sliced due to reaching one of its parameter-out edges of one of the call sites could not use a parameter-in edge for another call site. As a result, no invalid paths are formed, and the context-sensitivity is preserved.

1.5 Motivation

The PDG-based slicing approach makes a comprehensive analysis for getting all the data and control dependencies in the program. In this approach, all the dependencies are obtained at the beginning, although not all of them are included in the final slice. In using this approach, slicing large systems causes a high overhead. Several works identify this problem. Atkisson and

Griswold [15] noted that the time and space performance is of major concern for dataflow analyses which are employed by the PDG-based slicing approach. In particular, this becomes evident when the data and control dependencies are computed a priori in large systems where significant amounts of time and space are required [16]. Hajnal and Forgács [17] had a concern in slicing legacy Cobol industrial systems because the construction of the PDG of such systems is expensive. They said that none of the existing slicing tools are suitable for such large systems. Duesterwald et al. [18] explained why the traditional dataflow analyses require a considerable amount of space and time.

The demand-driven slicing approach is an attractive option to overcome the potential overhead of the PDG-based slicing. This approach aims at computing only the necessary information (dependencies). Some previous works introduced this approach. Duesterwald et al. [18] described the advantages of demand-driven slicing analysis techniques, and they introduced a work that avoids the collection of unnecessary information (dependencies).

Demand-driven slicing approaches are divided into two categories. The main concern of the first category [12, 19, 17] is in computing the required data and control dependencies. The slicing approach presented in this thesis falls into this category. The second category parses and builds a PDG of the procedure if one of its call sites is visited during the analysis. This trend was found in a series of Atkinson and Griswold works [16, 20, 21, 15]. However, as we will see in Section 6.4, the current demand-driven slicing approaches suffer from either providing imprecise results or not being entirely demand-driven.

In brief, the motivation of this work comes from the fact that the current slicing approaches either use prohibitive time and space, compute on-demand imprecise results, or are not fully demand-driven.

1.6 Research Objective

This thesis aims to develop an entirely demand-driven slicing approach that computes only the required program dependencies, avoids the unnecessary computations, and provides precise results, for intra-procedural, inter-procedural, well-structured[8] and unstructured programs.

1.7 Research Problem and Questions

The state-of-the-art slicing approach holds unnecessary computations. As what is shown in Section 1.2, the computations of the data dependencies

[8]The constructs of well-structured programming languages do not allow for overlapping or intersecting the control flows. This makes the reading, understanding, and debugging such programs easier than the unstructured programs.

entails the first of such computations, because to find the definitions that reach a particular statement, then all the definitions in the program must be propagated. This comprehensive solution is expensive.

The algorithm that obtains the control dependencies in the state-of-the-art slicing approach builds on the post-dominator tree. As shown in Section 1.3, constructing such a tree requires a comprehensive iterative technique and this type of techniques does not distinguish between the necessary and unnecessary information. As a result, obtaining the control dependencies is another source of unnecessary computation.

The straightforward implementation of the dataflow analyses requires a set of data flow facts at each program point. Each of those sets stores the data queries that visit its program point. Using the data sets to store the dataflow queries helps to track the flow of the queries, but it adds many operations in the level of each data set (e.g. checking, removing and copying) and in the level between the data sets (e.g. intersecting and unifying). Further, this method requires extra memory space to be available. The amount of memory space becomes considerable in slicing large industrial systems, which causes scalability difficulties. However, there are some techniques such as “copy-on-write" and “bit-vector" to reduce the memory needs for traditional dataflow analyses. The question is asked here whether storing the data queries for every program point is really necessary for achieving the primary goal of the slicing (producing a slice).

Traditional dataflow analyses generate and propagate dataflow queries in all the forward or backward paths. Often, the query is propagated in a big tree of irrelevant paths. These propagations waste time and memory space.

Demand-driven slicing approaches try to avoid as much as possible the computation of entire program dependencies before the slicing. These approaches compute the needed dependencies while the program is being sliced. This is great for avoiding getting unneeded relations, but on the other side, when the same program is sliced many times for different slicing criteria, the same program dependence might be obtained many times. Hence, generating many slices in a demand-driven way entails undoubtedly significant wasteful efforts.

In inter-procedural programs, the procedure has a set of formal parameters that are classified into input parameter and output parameter. Computing the transitive dependencies on the fly for one of the parameters is the challenge that we should solve for building a demand-driven inter-procedural slicing approach.
From the above problems, the following research questions are formulated:

RQ1 How to compute on-the fly the data and control dependencies?

RQ2 How to eliminate the computations that are wasted due to propagating the data queries into unrelated paths, tracing the data queries, and slicing the same program many times?

RQ3 How to compute properly and on-the-fly the calling-context[9] of calling sites?

1.8 Thesis Overview

The first part is an overall summary of the thesis, organized as follows. Chapter 2 introduces the concept of the Predicated Code Block graph and the slicing approach based on this graph. Chapter 3 summarizes the contributions of the thesis. Chapter 4 explains the research methods used in developing the new slicing technique. Section 5 presents some experimental evaluations that show the effectiveness of the new slicing technique. Section 6 discusses related work. Finally, Section 7 states the conclusion. The second part is a collection of publications included in this thesis, listed as follows:

Paper A
Static Backward Program Slicing for Safety Critical Systems
Husni Khanfar, Björn Lisper, Masud Abu Naser
Presented in ADA-Europe conference, 2015 [22]

Paper B
Enhanced PCB Based Slicing
Husni Khanfar, Björn Lisper
Presented in META 2016 workshop [23]

Paper C
Demand-Driven Static Backward Slicing for Unstructured Programs
Husni Khanfar, Björn Lisper, Saad Mubeen
Technical Report, Mälardalen University, Sweden 2019

[9]In the calling site, the actual parameters are classified into two categories; *Actual-Out* and *Acutal In*. The actual-out parameter holds a value that the calling side sends it to the procedure header, while the actual-in parameter receives a value sent from a return statement in the procedure body to the calling site. Usually, each Actual-in parameter is data dependent on all or some actual-out parameters. This data dependence is named a transitive data dependence in accordance to Horwitz et al. [6].

2. Introduction to PCB-based Slicing

This thesis proposes a new program slicing approach. The new slicing approach builds on a new program representation that is referred to Predicated Code Block (PCB) graph. The main purpose of this graph is to preserve the syntactic structure of the source code. The main unit in the PCB-graph is the PCB, which represents a conditional statement (e.g. if-then, while) and the main body of the function. The PCBs are connected together by *interfaces*[1] to construct a PCB graph. The original place of every PCB is represented by a placeholder. The PCBs and the interfaces form a PCB graph representation. Figure 2.1 shows the PCB graph of the program in Figure 1.1.

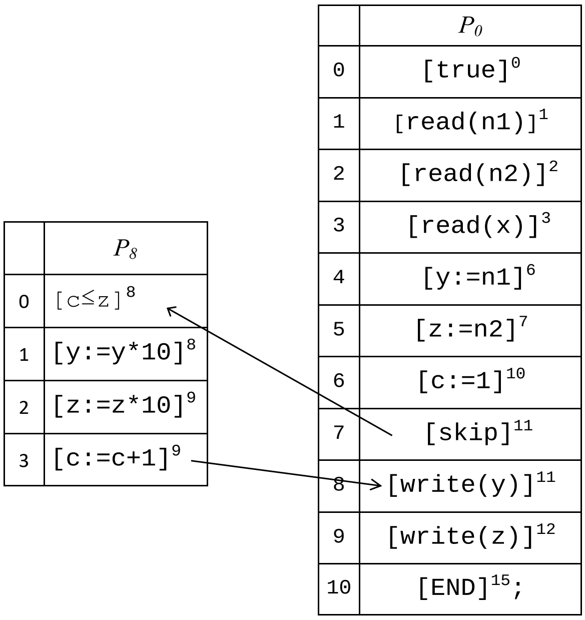
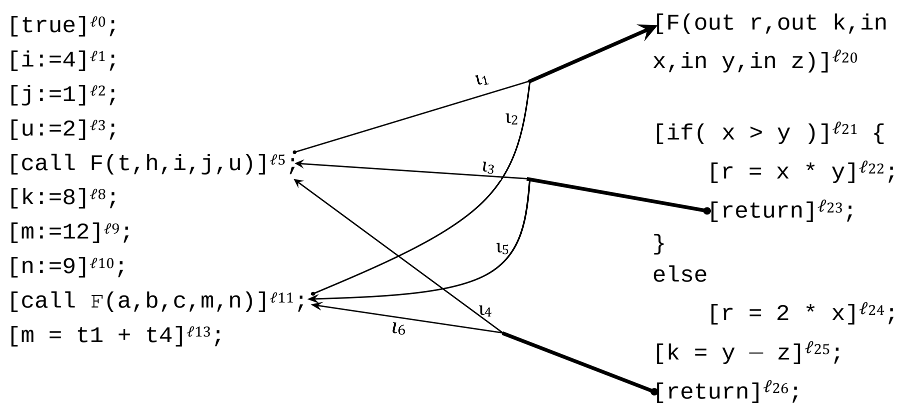


Figure 2.1: The PCB graph of the source code in Figure 1.1

[1]The interface corresponds to the uni-directional edges in the CFGs



[true]$^{\ell 0}$;
[i:=4]$^{\ell 1}$;
[j:=1]$^{\ell 2}$;
[u:=2]$^{\ell 3}$;
[call F(t,h,i,j,u)]$^{\ell 5}$;
[k:=8]$^{\ell 8}$;
[m:=12]$^{\ell 9}$;
[n:=9]$^{\ell 10}$;
[call F(a,b,c,m,n)]$^{\ell 11}$;
[m = t1 + t4]$^{\ell 13}$;

[F(out r,out k,in x,in y,in z)]$^{\ell 20}$

[if(x > y)]$^{\ell 21}$ {
    [r = x * y]$^{\ell 22}$;
    [return]$^{\ell 23}$;
}
else
    [r = 2 * x]$^{\ell 24}$;
[k = y – z]$^{\ell 25}$;
[return]$^{\ell 26}$;

Figure 2.2: Example of Connecting the Call Sites with Procedure Headers in PCB Graphs

In Figure 2.1, there are two PCBs, the first PCB represents the main body of the program, while the second PCB represents the While conditional statement.

In the PCB-based slicing, the slicing criterion, e.g. $< z, 12 >$, is converted to a dataflow query $< z, 12 >$. Afterward, each PCB processes its dataflow queries. As instance, the dataflow query $< z, 12 >$ propagates backward from P_0-9 to P_0-0. In this path, the query $< z, 12 >$ is killed when it visits the first statement that assigns a value to z. In our case it is P_0-5. Killing the dataflow query at any statement slices this statement. When the dataflow query $< z, 12 >$ arrives the outgoing side of an interface, it is reproduced inside the ingoing side of the PCB. In our case, it becomes $< z, 3 >$ in P_8, and P_8 processes it in the same manner. The prorogations obtain the data dependencies between the statements in the source code. In structured source codes, as soon as one of the statements is sliced, the first statement in the PCB, whose index is 0, is sliced. This is the method whereby the control dependencies are captured immediately from the PCB graph.

In slicing inter-procedural programs, all the sites are connected through a super interface with the procedure header and its return statements. The super interface consists of many thin parts linked through a joint to a single thick part. The thick part holds some general information that are required from all the call sites, whereas the thin parts hold other information that are exclusively linked to individual call sites.

Fig. 2.3 shows an example of an unstructured program. Three phases compute the control dependencies in such programs. In our example, if the requirement is to find the predicates that control the execution of the statements whose label is d, then the following phases are applied: The first phase checks whether d exists in a conditional statement that does not comprise a jump pro-

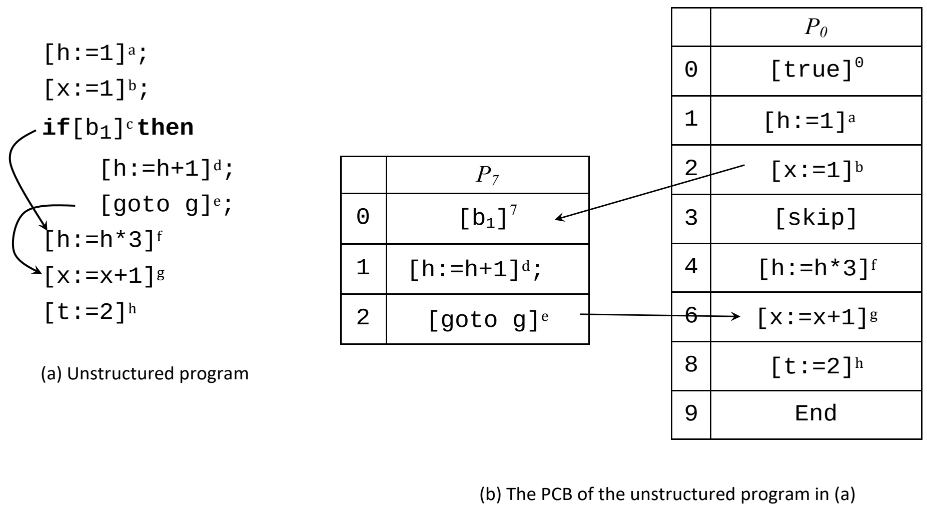


Figure 2.3: Unstructured program and its PCB-graph

gram flow. If this is the case, then d is control dependent only on the predicate of this conditional statement. Otherwise, other predicates might control d. In the case, d exists in the conditional statement *cs* whose predicate is c and its boundaries are from c to e. Since *cs* comprises a jump program flow, then d is not control dependent only on c. Thus, the analysis should move to the next phase. The second phase finds the sequence of overlapping flows *Seq* that bypasses d. The predicates which might control d are those in the outgoing side of the flows in *Seq*. Herein, the sequence of overlapping flows that bypasses d is [c$\rightarrow$ f,e$\rightarrow$ g], and the potential predicates to control d is {c}. The third phase checks carefully whether d is control dependent on c. To do so, the paths from c are explored, provided no explorations go beyond d. We get two paths from c. The first path is [c,g,f,h] and the second path is [d]. Since one of the paths reaches the last statement and the second path stops at d, we find that d is control dependent on c.

3. Technical Contributions

This thesis presents two main contributions. The first is a novel slicing approach that builds on a new program representation (PCB graph). The main feature of this approach is in avoiding many types of unnecessary computations performed in the state-of-the-art slicing approach. The second contribution is a novel approach that can compute on the fly the control dependencies in unstructured programs. The later is designed to be integrated into the PCB-based slicing, but it is presented as a separate contribution because it could be used with other slicing approaches or for other purposes than program slicing.

3.1 Contributions

The complete list of contributions is as follows. Each contribution is denoted by C.

C1: PCB-based Static Backward Slicing Approach

The PCB-based slicing is a new slicing approach that is designed especially to target the least possible computation required for forming slices from their original programs. This approach provides many distinct features. First, it computes only the required program dependencies to form the slice. Second, it implements the dataflow analysis in a particular way to avoid using data sets that are growing along the analysis. Instead, it stops when all the data sets become empty. Third, an incremental slicing feature can be integrated in the PCB-based slicing to save the data dependencies. Finally, the propagation of data queries is controlled to prevent unnecessary propagations into irrelevant paths.
Empirical experiments have been conducted to test the correctness of the PCB-based slicing approach, measure its performance, test the effect of controlling the propagation of data sets, and measure the performance gained from using the incremental slicing feature.
The proposed slicing approach works with well-structured programs with procedures, and unstructured programs without procedures. This contribution addresses the research questions RQ1 and RQ2. The major parts of this approach are as follows:

1. A new program representation is introduced. This representation translates each conditional statement to a Predicated Code Block (PCB). Then, The PCBs are connected to form the *PCB graph*. This graph is an alternative to the well-known CFG representation that also keeps some structure, which is useful when analyzing the program dependencies.

2. The Strongly Live Variable SLV dataflow analysis is employed to find the data dependencies. Since the direction of both the SLV analysis and the PCB-based slicing is backward, the employment of the SLV analysis allows computing the data dependencies in a demand-driven fashion.

3. The dataflow analysis is implemented in a particular way. It uses data sets that are shrinking during the analysis and reaches the fixed-point when these sets become empty.

4. The propagation of the dataflow queries towards irrelevant PCBs are prevented in order to decrease the amount of unnecessary computation.

5. For the inter-procedural case, the transitive dependence d at the call site of a procedure is computed on the fly when this dependence is demanded. Afterwards, the correspondent transitive dependencies to d in other call sites are concluded directly from d.

6. The control dependencies in well-structured programs are captured immediately from the PCB-graph. For unstructured programs, a new approach is developed to compute on the fly the control dependencies[1].

7. When slicing the same code several times with respect to different slicing criteria, the incremental slicing feature can be integrated in the PCB-based slicing to avoid computing the same data dependence for every slice that requires it.

C2: Computing on the fly the Control Dependencies

The second contribution introduces the first approach that can compute in a demand-driven fashion the control dependencies in unstructured programs. The approach is based on three theorems:

1. The first theorem can be used to quickly decide whether a particular predicate is the only predicate on which the statement of interest is control dependent.

2. The second theorem is used to determine a set of statements that are certainly not control dependent on a particular predicate.

3. The third theorem can be used to check whether a given statement is control dependent on a particular predicate.

[1]This approach is presented in C2

This approach builds on three phases, each of which is based on one of the theorems. The first phase is exact and fast, but it cannot be applied to all the cases. It reads the control dependencies directly from the PCB-graph. The second phase makes another layer of a fast filtration, which narrows the set of predicates that might control the execution of a statement of interest. The last phase makes in-depth path explorations from a predicate to check whether it controls the execution of a given statement. It gives accurate results without using fixed-point iterations. This contribution addresses the research question RQ1.

3.2 Included Publications

This thesis is based on a collection of three publications of which Husni Khanfar is the main author. The main approach and methods are suggested by Khanfar, but some contributions are suggested by others, as shown below.

Paper A

Static Backward Program Slicing for Safety Critical Systems
Husni Khanfar, Björn Lisper, Masud Abu Naser
Presented in ADA-Europe conference, 2015 [22]

This paper addresses the contribution C1.

In this paper, there are some contributions provided from others:

- Masud Abu Naser suggested representing the procedure body as a PCB, predicate of which is always true.
- The transfer function of slicing the PCB was implemented and described first by Husni Khanfar, then it was written, suggested and formalized by Masud Abu Naser.
- Masud Abu Naser suggested using the placeholder *in-child* to preserve the original place of the if-else conditional statement in the PCB graph.
- Husni Khanfar and Masud Abu Naser worked together to develop an approach for constructing PCBs and interfaces from intraprocedural programs.

Paper B

Enhanced PCB Based Slicing
Husni Khanfar, Björn Lisper

Presented in META 2016 workshop [23]

This paper addresses the contribution C1.

Some contributions from others:

- Husni Khanfar implements the conversion of the source code to a PCB graph. Daniel Hedin contributed a formal and recursive definition describing how to derive a PCB graph from a While program.
- Daniel Hedin contributed by using the placeholder skip to preserve the original location of if-then and while conditional statements.
- Masud Abu Naser suggested using a transfer function to formalize Husni's method in interprocedural slicing.

Paper C

Demand-Driven Static Backward Slicing for Unstructured Programs
Husni Khanfar, Björn Lisper, Saad Mubeen
Technical Report. Mälardalen University, Sweden 2019

This paper addresses the contributions C1 and C2.

3.3 Map of Contributions

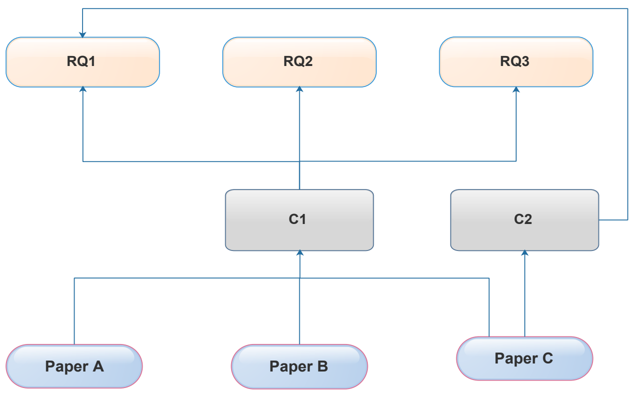


Figure 3.1: The contributions map: publications → contributions → research questions

Figure 3.1 Mapping among research questions, research contributions and publications.

4. Research Methodology

The selection of an appropriate research methodology is critical to ensure that the work achieves the research objective and makes an adequate contribution. The research methodology used in this paper consists of the following iterative steps:

1. Formulating the problem:
   We conduct a complete literature review of the program slicing. This review assists us in formulating the research motivation, research questions, and research objective.

2. Proposing a solution:
   A solution is proposed to achieve the research objective. We suppose that every solution should bridge a gap in the current knowledge, and can be formulated theoretically with a promise of significant practical results.

3. Implementing and evaluating the solution:
   The practical implementation and the mathematical or logical deduction from the current knowledge assists in proving both the correctness and effectiveness of the new proposed solution.

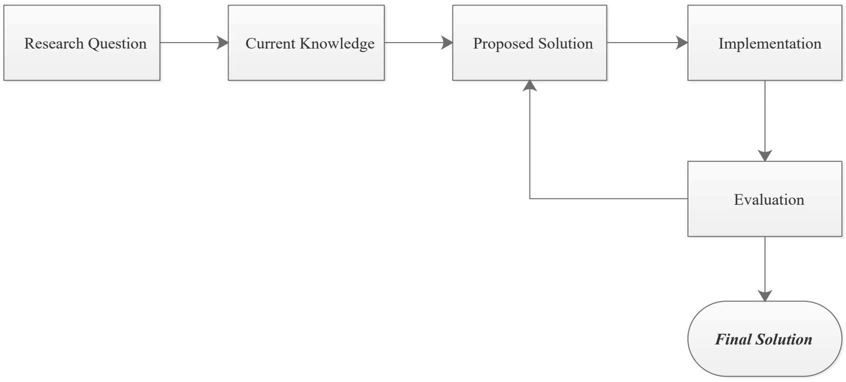


Figure 4.1: The Research Method

Since the proposed solution might be adjusted after evaluating the solution to keep it more and more consistent with the expectations, many iterations might take place between (1) to (3). The contributions were developed using the workflow depicted in Figure 4.1. Every research question is addressed separately from other questions. For solving a research question, some source code examples that create a challenge to this question are formed. The collection of such examples is used to induce a solution. The new proposed solution is proven in constructing an implementation to apply experimental evaluations, scaling the correctness and measuring the performance. These three steps (proposing or improving a solution, implementing it, and evaluating it) constitute a single iteration, which is repeated many times until the proposed solution becomes stable and its outcome agrees well with the expectations. At this moment, we select the proposed solution to be the final solution.

5. Experimental Evaluations

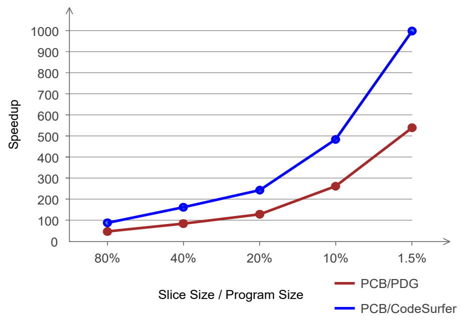
The contribution C1 is evaluated by means of experimental evaluation that is presented in Papers A and B. This evaluation is obtained for structured programs. The aim of the evaluation is to test the correctness and estimate the performance gain. The evaluation contributed in finding the bottlenecks, faults and imprecise results. Therefore, the evaluation plays an essential role in developing the algorithms.

The practical implementations are designed based on the syntax of the simple imperative While language that is used for academic purposes. This language is not used to implement real programs. Thus, a code generator is developed to generate synthetic programs. These programs are produced to have deterministic specifications, which consist of a set of individual factors such as the number of variables and statements. This control enables to study the effect of each factor on the performance of the PCB-based slicing.

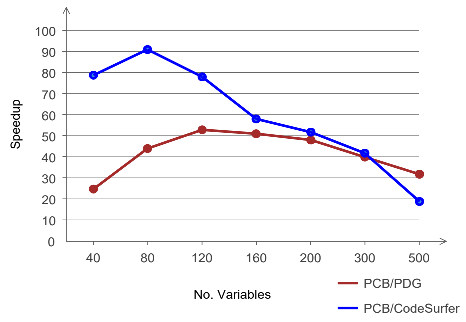
In Paper A, for evaluating the new proposed slicing approach, a local implementation for the PCB-based slicing and another local implementation for the PDG-based slicing were developed. Further, the slicer of CodeSurfer from GrammaTech[1] was used to participate in the evaluation. The evaluation obtains many observations. First of all, the evaluation shows that the PCB based slicing produces equal slices to those produced from the PDG-based slicing. So, the correctness of the new approach is proved. Second, the PCB-based slicing performs higher speedups[2] with smaller slices relative to the size of the original programs. This fact is stated in Figure 5.1a. Third, in studying the relationship between the number of variables in the program and the speedup, it is observed that there is a steady rise in the speed up until peaked at a certain point. After that, the speedup decreases. This fact is depicted in Figure 5.1b. Fourth, Figure 5.1c plots the speedup relative to the CodeSurfer which increases as long as the number of predicates increases, but this is not the case with the local implementation of the PDG-based slicing, where the speedup is peaked at the middle, then it is decreased. Finally, the evaluation studies the relationship between the speed up and the number of statements in the program. Figure 5.1d shows that the speedup over the local implementation of the PDG-based slicing remains steady with varying the number of program

[1] www.grammatech.com

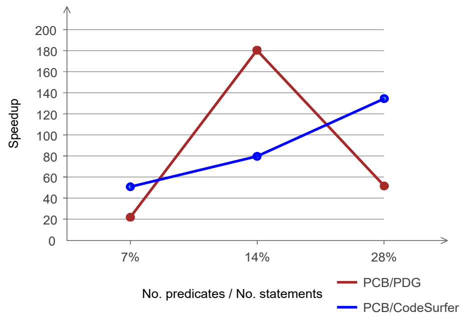
[2] The Speedup = the time consumed to slice the program by the PCB-based slicing $\div$ the time consumed to slice the same program by the PDG-based slicing.



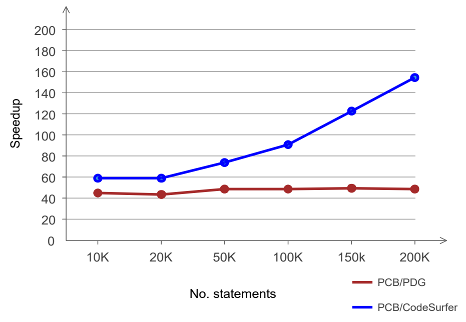
(a) varying number of slice percentages



(b) varying number of variables



(c) varying number of control predicates



(d) varying number of program sizes

Figure 5.1: PCB-based slicing vs. PDG-based slicing

statements, but it increases as the number of program statements increases with the Code Surfer implementation.

Paper B presents two sets of practical evaluations. The first set of evaluations aims at measuring the performance gained due to using whitelists[3] over the conventional growing dataflow sets. This evaluation is achieved by producing six different files, with a number of variables varying from 25 to 800, and they share other factors such as the program size and the number of predicates. Figure 5.2 indicates that the PCB-based slicing works much better with whitelists than the growing sets. The figure shows that higher speedups can be achieved for a program having a higher number of variables. The tremendous improvement shown in Figure 5.2 is due to getting rid of the unnecessary computations caused by propagating the dataflow queries toward irrelevant paths.

The second set of experiments in Paper B assesses the incremental feature. For doing this, a comparison is made between three applications (A), (B) and (C), where both (A) and (B) implement the PCB-based slicing technique.

[3]It is essential to mention that the PCB-based slicing implementations shown in Fig. 5.1 use whitelists

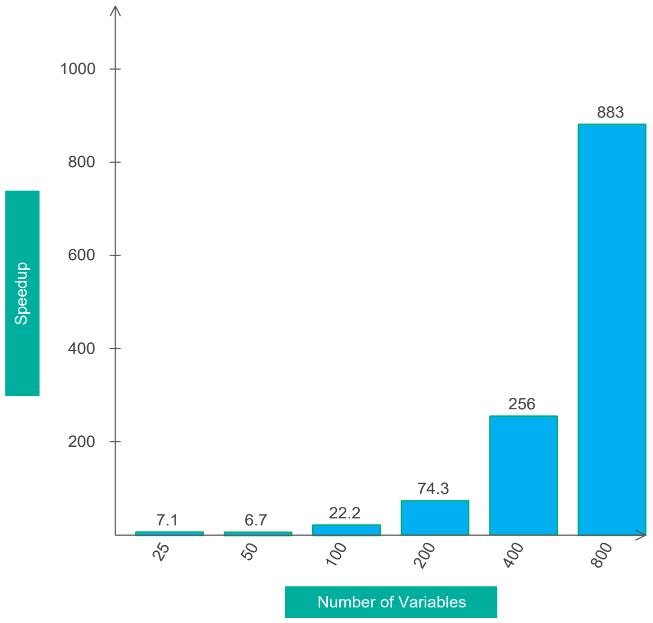


Figure 5.2: Relation between the Speedup (PCB-slicing-using-whitelists/PCB-slicing-using-blacklists) and the number of variables in intraprocedural programs.

The difference is that (A) uses the incremental feature, while (B) does not. The application (C) is an implementation of the PDG-based slicing. Table 5.1 presents the results for slicing a particular program 15 times for 15 slicing criteria. The table shows the time for each slice as well as the sum of the times from the first slice to the current slice.

The first column displays the number of the slice. The second column measures the slice size relative to the original program size. The times, which are required to produce every slice by the implementations (A), (B) and (C), are presented in the third, fifth and seventh columns, respectively. For every slice *N*, the corresponding cells in the fourth, sixth and eighth columns accumulate the times from the slice no.1 to the slice no.*N* in the third, fifth and seventh columns, respectively. What is interesting in this data is that after 5-6 slices, (A) behaves similarly to (C). It analyses the program in linear time due to using reachability analysis on the PDDG. It is worthwhile to notice that SumA for *N* is always much less than SumC. In Figure 5.3, it is observed that SumB ever increases with new slices and SumA is almost constant after 5 to 6 slices. These results state the fact that the incremental slicing feature eliminates unnecessary computations resulting from slicing the same program many times in a demand-driven fashion.

no.	Slice Size	A (m.s.)	SumA	B(m.s.)	SumB	C(m.s.)	SumC
1	63%	234	234	180	180	12204	12204
2	33%	468	702	108	288	1	12205
3	14%	1	703	54	342	1	12206
4	15%	1	704	54	396	1	12207
5	51%	234	938	144	540	1	12208
6	21%	1	939	72	612	1	12209
7	79%	18	957	216	828	1	12210
8	58%	1	958	180	1008	1	12211
9	40%	1	959	108	1116	1	12212
10	37%	1	960	108	1224	1	12213
11	45%	1	961	126	1350	1	12214
12	10%	1	962	18	1368	1	12215
13	20%	1	963	72	1440	1	12216
14	39%	1	964	108	1548	1	12217
15	50%	1	965	144	1692	1	12218

Table 5.1: *Adding incremental slicing feature to the PCB-based slicing approach*

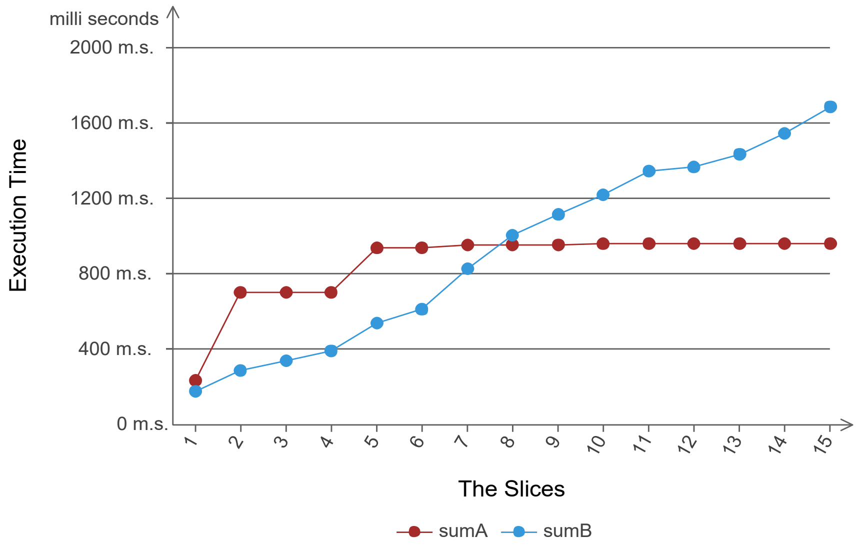


Figure 5.3: This chart illustrates sumA and sumB shown in Table 5.1

6. Related Work

A considerable amount of literature has been published on program slicing. In this section, we only consider related work in static backwards slicing. There are four groups of work that are directly related to the thesis: *Basic Program Slicing*, *Inter-procedural Program Slicing*, *Computing Control Dependencies in Unstructured Programs* and *Demand-Driven Slicing Approaches*. The first group contains the works establishing the fundamentals of the state-of-the-art slicing method. The second focuses on computing the control dependencies in unstructured programs, whereas the last group collects the previous demand-driven slicing techniques.

6.1 Basic Program Slicing Techniques

Mark Weiser [1] was the first to define the program slicing concept in the context of debugging. The major criticism of Weiser's work is that his approach does not identify the minimal slice. Some techniques were developed later based on the data flow equations [5]. Ottenstein and Ottenstein [4] introduced the PDG as a basis for the slicing. Ottenstein's approach does not take into its account inter-procedural and unstructured cases. Horwitz et al. [6] extended the PDG to the System Dependence Graph (SDG) to represent the dependencies in programs having many procedures. Furthermore, they presented a new inter-procedural slicing technique.

Danicic et al. [25] presented a new algorithm that computes the static slices, backward and in parallel. They relied on dividing the CFG into a set of interconnected concurrent processes. Chung et al. [26, 27] presented a method to slice the programs with respect to the pre and post conditions in order to obtain more precise slices.

The main works in program slicing either compute over-approximated slices or make a comprehensive analysis that depends on obtaining all the program dependencies. The slicing approach presented in this thesis suggests a new method that computes the minimal slice in a demand-driven fashion.

6.2 Inter-procedural Program Slicing

Weiser [1] was the first in introducing an interprocedural-slicing algorithm that provides executable slices. Weiser's algorithm suffers from including large portions of the source code that do not affect the slicing criterion. This failure occurs because it includes all the call sites if one of them is sliced, as well as, this algorithm treats the call sites as indivisible components, where all the parameters are included in the slice if one of them is sliced.

Most of the inter-procedural slicing works study concerns related to extending the HRB[1] approach such as improving its performance [28, 29, 30, 30, 31, 32, 33], producing executable slices [34], working with aliasing parameters [35], treating recursive cases [30, 36, 37], working with pointers [37, 37], studying the trade-off between context-sensitivity and accuracy [38, 39, 40, 41], preserving the syntax [42], and finding the arbitrary inter-procedural control dependencies [43]. From our perspective, using the SDG itself holds many unnecessary computations, which we are trying to avoid by using the PCB-graph as an alternative.

6.3 Computing the Control Dependencies in Unstructured Programs

The main challenge in working with unstructured programs is the presence of arbitrary control flows, which makes the source code difficult to be understood. Such flows make finding the control dependencies harder than such dependencies in well-structured programs. The essential step in finding the control dependencies in the state-of-the-art slicing approach is in computing the post-domination information. There are many algorithms that find such information. They differ in the internal algorithm, but they share one thing, they need to make an iterative and global analysis in the level of the entire source code, where all the nodes or program statements have to be included in the analysis. Lowry and Medlock [44] designed an algorithm that finds all the paths from the entry node to each node. Afterwards, the algorithm removes every node in order to check which nodes become unreachable. By this, it determines the dominator[2] for each node. Allen [24] in 1970 used the dataflow equations to find the dominators, and she suggested equations, which can be solved with time complexity of $O(N^2)$. In 1975, Hecht and Ullman [45] suggested dataflow equations that work in linear time. Aho and Ullman [46] developed an algorithm that works in quadratic time, in the number of edges. Purdom et al. [47] suggested an algorithm whose time complexity

[1]HRB denotes to the work of Horwitz, Reps, and Binkley in [6]

[2]We say that ℓ *dominates* ℓ' if all the paths from the entry node to ℓ' go through ℓ. Usually, the algorithms used to compute the post-dominators can be applied for computing the dominators, and vice versa.

is quadratic. Lengauer and Tarjan [14] provide the best algorithm for finding the dominators that works in linear time. Cooper et al. [13] presented an algorithm for finding the post-domination information. Their algorithm is theoretically worse than Lengauer algorithm, but its implementation works better than Lengauer's algorithm.

6.4 Demand-Driven Slicing Approaches

This subsection compares the PCB-based slicing method with five different demand-driven slicing methods. The comparisons try to realise the differences in the mechanisms used for computing the data dependencies, control dependencies, inter-procedural slicing and the pointer analysis.

Kraft [19] presented *Katana*, a tool that uses a new program slicing approach to extract simulation models from source codes. Kraft invented his tool because the existing tools for program slicing were not scalable enough, and they could not work with large industrial systems, where the number of source code lines might reach to millions. The computation of data dependencies is flow-insensitive, which makes the results over-approximated and leads to produce larger slices. Katana obtains the control dependencies of well-structured programs directly from the source code similar to the PCB-based slicing approach, but it neglects unstructured control flows. Similar to the way of computing the data dependencies, Katana also suffers from being over-approximated in analysing inter-procedural cases and pointer analysis.

Sandberg et al. [48] proposed another demand-driven slicing approach called SimpleSlice. This approach aims to accelerate Worst-Case Execution Time analysis by slicing the statements affecting the variables in the control-flow conditions. These variables with their locations constitute the slicing criteria. Every statement that assigns a value to a variable of a slicing criterion is sliced, and the variables used in this statement are considered relevant variables. The iterations go on until no more relevant variables are available. SimpleSlice is similar to Katana in using flow-insensitive information to compute the data dependencies. SimpleSlice does not need to provide any analysis to find control dependencies because all the conditions are sliced at the beginning. SimpleSlice uses Steensgaard's algorithm [19], which is not demand-driven, for computing the points-to sets[3]. There is no explanation about the inter-procedural slicing.

Lisper et al. [12] presented a slicing approach called static backward demand-driven slicing. This approach resembles the PCB approach in using SLV data flow analysis for obtaining the data dependencies, but it differs in using conventional data flow equations, which depend on storing the data flow queries at the program points. Lisper's method presents a solution for

[3]The points-to set for a specific pointer consists of all the variables which are possibly dereferenced by the pointer

slicing the inter-procedural cases on-the-fly. However, this work did not propose a pointer analysis technique and its control dependence analysis is not entirely demand-driven.

	Data Dependencies	Well-Structured	Unstructured Programs	Inter-procedural Programs	Pointers
PCB	✓	✓	✓	✓	▷
Lisper	✓	✓	×	✓	×
Katana	×	✓	×	×	×
SimpleSlice	×	×	×	×	×
Sprite	✓	×	×	✓	×
Hajnal	✓	✓	×	✓	×

Table 6.1: *Comparison Between Six Demand-Driven Slicing Approaches*

Atkinson and Griswold have published a series of works [15, 21, 49, 16] for presenting a method that works in a demand-driven fashion. These publications made the theoretical basis for developing a new slicing tool called Sprite. The primary concern of Sprite is to address the scalability issues that arise in large industrial systems. Sprite constructs the CFG of the procedure in isolation, from other procedures, for the first time the data flow analysis visits one of its call sites. After that, the intraprocedural algorithms are applied to this CFG to compute the entire data and control dependencies in this procedure. Sprite is similar to Lisper's approach in translating the slicing criteria to live variables and then applying conventional Live Variable dataflow analysis to get the use-definitions chains. The Sprite tool employs the Steensgaard's algorithm [19] for computing the points-to set.

Hajnal and Forgács [17] have designed a demand-driven slicing approach to maintain already ageing COBOL legacy systems. Their method relies on propagating queries to compute the data and control dependencies. It works well for direct data dependencies and well-structured code, but it neglects indirect data dependencies, generated due to the existence of pointers, unstructured codes and inter-procedural cases.

Table 6.1 summarizes the above information. The symbol ✓ denotes the support to compute accurately and on-demand. The symbol ×denotes either over-approximated or not entirely demand-driven approach. $\triangleright$ denotes ongoing activities.

7. Conclusion and Future Work

In this thesis, a new static backward slicing technique is introduced. This technique makes a significant improvement in performance and creates accurate minimal slices.

In response to the first research question, this technique employs the SLV dataflow analysis to compute on-the-fly the data dependencies. This technique builds on a new program representation called PCB-graph. This representation facilitates getting the control dependencies immediately in well-structured programs. The control dependencies for unstructured programs are obtained on-the-fly because of using a more efficient technique that looks to the source code from two sides; location-based information and flow-based information. As a solution to the second research question, the new technique is designed to avoid many types of unnecessarily computations. It avoids repeating unnecessarily the same computations when a program is sliced many times for different slicing criteria. This is implemented by saving every data dependence relation found in the source code. Further, it avoids saving unnecessarily a copy of each data query at every program point that it visits. Finally, it works to curb the propagation of dataflows as much as possible. The third research question is solved by enabling the PCB-based slicing approach from working with inter-procedural cases. The key in this solution is in enabling the procedure to share its global results with all its call sites and in exchanging dedicated special data with each of its call sites.

Many future works could expand the findings in this thesis to make them applicable to modern programming languages. One of the chief issues is to produce an executable slice that behaves like the original program for a particular slicing criterion. Producing such slices requires to slice both the statements as well as the dependencies which affect the criterion. For doing so with unstructured programs, the goto statements that are relevant to building these dependencies have to be included in executable slices. This inclusion deserves to be studied carefully in the future. Besides the relevant goto statements, making on-the-fly pointer analysis enables us to apply the PCB-based slicing approach in many languages such as C. Finally, this thesis implemented the SLV dataflow analysis in a new way to improve performance. It is attractive to study whether other types of dataflow analyses can be implemented in a similar way.

This thesis focuses on computing one type of control dependencies (standard control dependencies). Some efforts are needed to compute on-demand other types of control dependencies such as the strong control dependencies [50]. Finally, the computation of the inter-procedural control dependencies[1] is another suggested potential future work.

[1]The inter-procedural control dependencies refer to dependencies that appear because the called procedure halts the operation of the program and does not return back to the calling procedure.

8. Bibliography

[1] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[2] Jean-Francois Bergeretti and Bernard A Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):37–61, 1985.

[3] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys (CSUR)*, 44(3):12, 2012.

[4] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.

[5] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[6] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[7] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *International Workshop on Automated and Algorithmic Debugging*, pages 206–222. Springer, 1993.

[8] Hiralal Agrawal. On slicing programs with jump statements. In *ACM Sigplan Notices*, volume 29, pages 302–312. ACM, 1994.

[9] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1097–1113, 1994.

[10] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 2015.

[11] Björn Lisper and Husni Khanfar. Fast and precise slicing of low-level code. *Mälardalen University - Technical Report*, 2013.

[12] Björn Lisper, Abu Naser Masud, and Husni Khanfar. Static backward demand-driven slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 115–126. ACM, 2015.

[13] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[14] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[15] Darren C Atkinson and William G Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 52. IEEE Computer Society, 2001.

[16] Darren C Atkinson and William G Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th international Conference on Software engineering*, pages 16–27. IEEE Computer Society, 1996.

[17] Ákos Hajnal and István Forgács. A demand-driven approach to slicing legacy cobol systems. *Journal of Software: Evolution and Process*, 24(1):67–82, 2012.

[18] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48. ACM, 1995.

[19] Johan Kraft. *Enabling timing analysis of complex embedded software systems*. PhD thesis, Mälardalen University, 2010.

[20] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance: Research and Practice*, 10(6):415–441, 1998.

[21] Leeann Bent, D Atkinson, and W Griswold. A qualitative study of two whole-program slicers for c. *Technical Report*, 2000.

[22] Husni Khanfar, Björn Lisper, and Abu Naser Masud. Static backward program slicing for safety-critical systems. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 50–65. Springer, 2015.

[23] Husni Khanfar and Björn Lisper. Enhanced PCB-based slicing. In *Fifth International Valentin Turchin Workshop on Metacomputation*, page 71, 2016.

[24] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[25] Sebastian Danicic, Mark Harman, and Yoga Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313, 1995.

[26] In Sang Chung, Wan Kwon Lee, Gwang Sik Yoon, and Yong Rae Kwon. Program slicing based on specification. In *Proceedings of the 2001 ACM Symposium on Applied computing*, pages 605–609. ACM, 2001.

[27] Wan Kwon Lee, In Sang Chung, Gwang Sik Yoon, and Yong Rae Kwon. Specification-based program slicing and its applications. *Journal of Systems Architecture*, 47(5):427–443, 2001.

[28] Arun Lakhotia. Improved interprocedural slicing algorithm. *Report CACS TR-92-5-8, University of Southwestern Louisiana*, 1992.

[29] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 158. IEEE Computer Society, 2001.

[30] Panos E Livadas and Stephen Croll. System dependence graph construction for recursive programs. In *Computer Software and Applications Conference, 1993. COMPSAC 93. Proceedings., Seventeenth Annual International*, pages 414–420. IEEE, 1993.

[31] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19(5):11–20, 1994.

[32] Istvan Forgács and Tibor Gyimóthy. An efficient interprocedural slicing method for large programs. 1996.

[33] Panos E Livadas and Stephen Croll. System dependence graphs based on parse trees and their use in software maintenance. *Information Sciences*, 76(3-4):197–232, 1994.

[34] David Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):31–45, 1993.

[35] David Binkley. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum*, pages 261–268, 1993.

[36] Panos E Livadas and Theodore Johnson. An optimal algorithm for the construction of the system dependence graph. *Information Sciences*, 125(1-4):99–131, 2000.

[37] Donglin Liang and Mary Jean Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursions. In *International Conference on Software Maintenance*, page 421. IEEE, 1999.

[38] Gagan Agrawal and Liang Guo. Evaluating explicitly context-sensitive program slicing. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program analysis for Software Tools and Engineering*, pages 6–12. ACM, 2001.

[39] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 44–53. IEEE, 2003.

[40] Jens Krinke. Evaluating context-sensitive slicing and chopping. In *International Conference on Software Maintenance*, page 0022. IEEE, 2002.

[41] Jens Krinke. Context-sensitivity matters, but context does not. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 29–35. IEEE, 2004.

[42] Mark Harman, Lin Hu, Malcolm Munro, Xingyuan Zhang, Sebastian Danicic, Mohammed Daoudi, and Lahcen Ouarbya. An interprocedural amorphous slicer for wsl. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 105–114. IEEE, 2002.

[43] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 432–441. IEEE, 1999.

[44] Edward S Lowry and Cleburne W Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969.

[45] Matthew S Hecht. *Flow analysis of computer programs*. Elsevier Science Inc., 1977.

[46] Alfred V Aho and Jeffrey D Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., 1977.

[47] Paul W Purdom Jr and Edward F Moore. Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, 1972.

[48] Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. In *ACM SIGPLAN Notices*, volume 41, pages 103–112. ACM, 2006.

[49] Darren C Atkinson and William G Griswold. Effective whole-program analysis in the presence of pointers. *ACM SIGSOFT Software Engineering Notes*, 23(6):46–55, 1998.

[50] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.