Forward Static Program Slicing

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
In
Computer Science and Engineering

By

Raphnash Raphique
Roll No: 10606057
Bidyadhar Ekka

Roll No: 10606043

Under the Guidance of
Prof. D. P. Mohapatra

Department of Computer Science Engineering
National Institute of Technology
Rourkela
2010

Certificate

This is to certify that the thesis entitled Static Forward Program Slicing, submitted by Raphnash Raphique and Bidyadhar Ekka, B.TECH students in the Department of Computer Science and Engineering, National Institute of Technology, Rourkela, India, in the partial fulfillment for the award of the degree of Bachelor of Technology, is a record of an original research work carried out by them under our supervision and guidance. The thesis fulfills all requirements as per the regulations of this Institute and in our opinion has reached the standard needed for submission. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. D. P. Mohapatra
Department of Computer Science and Engineering
National Institute of Technology Rourkela
India – 769008

ACKNOWLEDGEMENT

On the submission of our Thesis report, we would like to extend our gratitude and sincere thanks to our supervisor Dr. D.P. Mohapatra, for his constant motivation and support during the course of our work in the last one year. We truly appreciate and value his esteemed guidance and encouragement from the beginning to the end of this thesis. He has been our source of inspiration throughout the thesis work and without his invaluable advice and assistance it would not have been possible for us to complete this thesis.

A special acknowledgement goes to Mr. B.Soubhagya Sankar, an M.Tech research scholar for his guidance throughout the thesis. Finally, our sincere thanks to Prof. B. Majhi for motivating us to perform better and better and providing us with all the resources required to carry out the thesis.

Raphnash Raphique

Bidyadhar Ekka

Abstract

Program slicing is a practical disintegration methodology that omits program modules that are irrelevant to a particular computation process based on a criterion known as the slicing criterion. The original program’s semantics is projected through the computation of an executable program formed by the left over modules called a slice. Using this methodology we can automatically determine the relevance of a module in a particular computation. Once such modules are ascertained amongst the program, the testing process takes considerably less effort and time because testing phase generally accounts for more than one third of time during the software development cycle. Slicing applications spread out through debugging methodologies to authentication of properties done through finite state models are appraised. Program slicing methodology reduces the effort of a software designer/coder and enthusiasts as well to directly get to the core of the problem. Forward slices contain all parts of the program that might be influenced by the variable. Static slicing may be used to identify these parts of the program that potentially contribute to the computation of the selected function for all possible programs inputs. Static slicing is helpful to gain a general understanding of these parts of the program that contribute to the computation to the selected function. In this project work, we have developed a forward static slicing algorithm. We have used file tracing to compute the forward static slices of a simple program. We have implemented our algorithm by using Java in net beans IDE on windows platform. The experimental results show that as the number of statements increase the slice computation time also increases raciprocally.

TABLE OF CONTENTS

Chapter 1

Introduction

Programming paradigms that imitate real world functioning are becoming immensely popular these days with innovative solutions and high level of easy interactivity plus the ease in the designing and maintenance of such software. However it has become a challenging research problem to find satisfactory and effective slicing algorithm [1].

During it lifecycle, an application/ software undergoes various changes. With these changes mainly carried out in order to eliminate the bugs that one might come across during the testing period, enhance functionality and to provide better stable performance compatibility as the need arises in new changing environments. Such alterations or tweaking might possibly also introduce errors or diversely affect the performance of some components or modules of previously working functionality of the software. So every time a software undergoes some modification , it undergoes testing again for checking smooth working in standard environments and provide confidence that modified code behaves as intended, and does not adversely affect the functioning of unmodified code [2]. Retesting of application code again from the beginning would take considerable amount of time so finding necessary codes relevant to the specific computation makes things very easier.

Based on the code analysis of the software several testing approaches are proposed for the selective retest problem. But all these approaches are computationally expensive to implement and various types of code relations among program elements are neither explicit in code, nor is code a compact representation. A better approach is based on intermediate representation of the program. This approach is computationally less complex than the code analysis and are more flexible so these can represent code relations among program elements that are not explicit in the code. Another advantage of the intermediate code representation is that we can have a better understanding of the program which is very useful for computing slices.

1.1 Motivation for our work

Slicing is mainly used in different s/w engg. applications such as program comprehension & testing. So, the slicing techniques need to be efficient. This requires to develop

- Efficient slicing algorithms
- Suitable intermediate representations

So, there is a pressing need to develop efficient slicing algorithms for programs.

1.2 Objective of our work

Our main objective is to develop a Forward slicing algorithm for simple programs.

1.3 Organization of the thesis

The rest of the thesis is organized as follows:

Chapter 2

This chapter mainly provides the basic concepts, definitions used in the rest of the thesis which is about introduction to program slicing and basic terminologies associated with intermediate representation for better understanding of program.

Chapter 3

Here we describe the related Work.

Chapter 4

In this section, we describe our proposed pseudo code for forward static slicing and its implementation and the results.

Chapter 5

We conclude the thesis and discuss the future work that can be done in this area.

References

Chapter 2

Fundamental concepts

In this section we discuss the basic concepts and terminologies associated to our work and that are used in later sections. And also few details which include in the intermediate representation of a program .

2.1 Program Slicing

Program slicing was originally introduced by Mark Weiser.[3]

- Finding all statements in a program that directly or indirectly affect the value of a variable occurrence is referred to as Program Slicing .

2.2 Slicing Criterion

- The pair <s,v> is known as Slicing Criterion where ‘s’ is a program point of interest and ‘v’ is a variable used or defined at s.[3]

2.3 Types of program slicing

Depending on the run-time environment, it can be

- Static slicing
- Dynamic slicing

Depending on graph traversal, it can be

- Backward slicing
- Forward slicing

2.3.1 Static Slicing

Static slicing may be used to identify these parts of the program that potentially contribute to the computation of the selected function for all possible programs inputs. Static slicing is helpful to gain a general understanding of these parts of the program that contribute to the computation to the selected function. Although static slicing has many advantages in the process of program understanding, static slices are frequently still large subprograms because of the imprecise computation of these slices. In addition, static slices cannot be used in the process of understanding of program execution. [5]

Considering an example for Static Slicing

1. IMPORT Math,In,Out;
2. VAR x,y:REAL;
3. op:ARRAY 10 OF CHAR
4. **In. Open;**
5. **In. String(op);**
6. **In. Real(x);**
7. **IF op="sin" THEN**
8. **y:=Math.sin(x);**
9. Else
10. **y:=Math.cos(x);**
11. **End**
12. **Out.REAL(y);**

Static Slicing of the above statements w.r.t. slicing criterion(12,y) are shown as bold

2.3.2 Dynamic Slicing

Dynamic slicing is used to identify these parts of the program that contribute to the computation of the selected function for a given program execution (program input). Dynamic slicing may help to narrow down this part of the program that contributes to the computation of the function of interest for particular program input. Dynamic slices are frequently much smaller than static slices. Moreover, dynamic slicing may be used to understand program execution. Programmers may still have difficulties to understand the program and its behavior. The slicing tools usually developed provide limited support during the process of understanding of large programs and their executions. Therefore, it is important to devise methods that will support the process of understanding of large software systems. One aid to understanding of large software systems is to use a intermediate representation of a program and then compute a slice from the graph. This slicing technique aims at giving a better understanding of large programs and their executions for a particular input. These concepts have been developed, static and dynamic program slicing which when combined with different methods of visualization of program slices is to guide programmers in the process of understanding of large programs and their executions.[6]

Considering an example for Dynamic Slicing

1. IMPORT Math,In,Out;
2. VAR x,y:REAL;
3. op:ARRAY 10 OF CHAR
4. **In. Open;**
5. **In. String(op);**
6. **In. Real(x);**
7. **IF op="sin" THEN**
8. **y:=Math.sin(x);**
9. Else
10. y:=Math.cos(x);
11. **End**
12. **Out.REAL(y);**

Dynamic Slicing of the above statements w.r.t. slicing criterion(12,y) are shown as bold

2.3.3 Forward Slicing

Forward slices contain all parts of the program that might be influenced by the variable.

2.3.4 Backward Slicing

Backward slices contain all parts of the program that might have influenced the variable at the statement under consideration.

Now considering few statements for forward and backward slicing

```
S1:VAR

S2:x,y,z: INTEGER;

S3:BEGIN

S4: x:=3;

S5: y:=x+4;

S6: z:=y+3;

S7:END
```

Output Forward slice w.r.t.(4,x)

```
y:=x+4;

z:=y+3;
```

Output Backward slice w.r.t(6,z)

```
x:=3;

y:=x+4;
```

2.4 Visualization of slices in large programs

Forward slices contain all parts of the program that might be influenced by the variable. Program slicing transforms a large program into a smaller one that contains only statements relevant to the computation of a given function. However, the slicing tools usually offer only limited help during the process of understanding of large programs. A program slice is represented in a textual form, i.e., a slice is displayed to programmers in the form of highlighted statements in the original program or as a subprogram by removing all statements from the original program that do not belong to the slice.[7]

2.5 Graph slicing

One aid to improve the understanding of large programs is to reduce the amount of detail a programmer sees by using slicing to represent a program and represent slice or nodes to the relevant computation.[8]

2.6 Control dependence graph

A Control dependence graph (CDG) is a graph representation of the program that represents which statements are dependent on which control condition. [9]

2.7 Data Dependence Graph

Data dependence graph (DDG) is a graph representation of the program, that represents the flow of data from statement to statement.[9]

Sample program for Dependence Graph

```
Static int
Link (int p, int q)
{
          If (rank[p] > rank[q])
           {
                    int t=p;
                    p=q;
                    q=t;
            }
          else if (rank[p] == rank[q])
                    rank[q] += 1;
          p[f] = q;
          return (q);
}
```

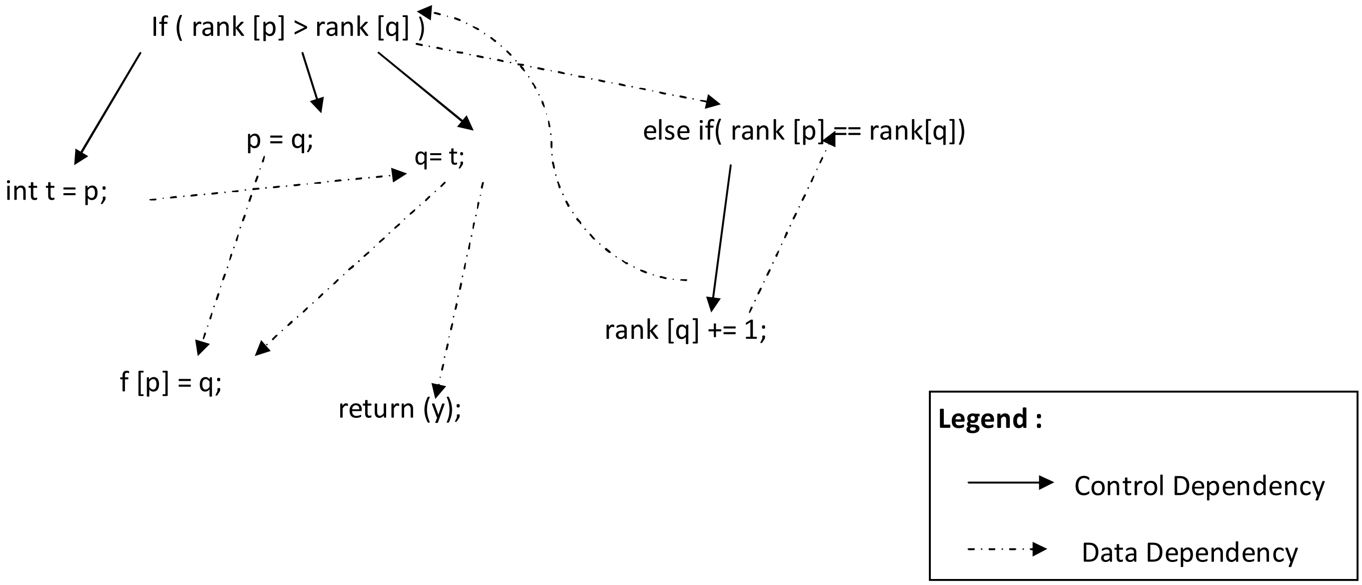


Figure 2.1: Dependence graph for the sample program

2.8 Program dependence graph

The program dependence graph (PDG) consists of nodes and direct edges. Each program's simple statement and control predicate is represented by a node. Simple statements include assignment, read, and write statements. Compound statements include conditional and loop statements and they are represented by more than one node. There are two types of edges in a PDG: data dependence edges and control dependence edges. A data dependence edge between two nodes implies that the computation performed at the node pointed by the edge directly depends on the value computed at the other node.[10]

Chapter 3

Related Work

Mark Weiser introduced analysis technique in his PHD thesis year 1979, the idea came to his mind is when he was observing experienced programmers debugging a program. Hence he found out that every experienced programmer uses slicing to debug a program. Program slices, as originally introduced by Weiser [12] are now called executable backward static slices. Weiser originally used a control-flow graph as an intermediate representation for his slicing. Horwitz [13] was the one who introduced the notion of forward slicing. Finally, Korel and Laski introduced the notion of dynamic slicing [14]. Intermediate representation is very important aspect in program slicing to have a better understanding of the program. This can be done with the help of analyzer called Lex. Lex can be used with YACC which is a parser generator . Lex, originally written by Eric Schmidt and Mike Lesk, is the standard lexical analyzer generator on many Unix systems. Flex & Bison, where Flex is a Lex implementation by Vern Paxon and Bison the GNU version of YACC. Flex and Bison are windows compatible which we have used. When properly used, these programs allow you to parse complex languages with ease. This is a great boon when you want to read a configuration file, or want to write a compiler for any language you (or anyone else) might have invented [11].

3.1 Constructing a graph

One aid to improve the understanding of large programs is to have Intermediate representation of a program which gives you a better understanding of program. Each statements are considered as a node and there dependency between each node is show. If a flow date is there between nodes then it is data dependency .And if a node or statement is having control condition then it is control dependency. The program dependence graph (PDG) consists of nodes and direct edges. There are two types of edges in a PDG: data dependence edges and control dependence edges. A data dependence edge between two nodes implies that the computation performed at the node pointed by the edge directly depends on the value computed at the other node. A ClDG captures the control and data dependence relationships that can be determined about a class without knowledge of calling environments and it represents the programs with object oriented features that include data hiding, inheritance, polymorphism, etc.

Considering a simple program

```
s1:     void NumSub(int n, int &total, double &avg, int &prod) {
s2:     int i=1;
s3:      total=0;
s4:      prod=1;
s5:      while (i<=n) {
s6:      sum=sum+i;
s7:      prod=prod*i;
s8:      i=i+1;
s9        }
s10       avg=static_cast<double>(sum)/n;
11 }
```

The program dependence graph (PDG) for the above sample program has been shown in the next page. It comprises of both the data and the system dependence graphs.

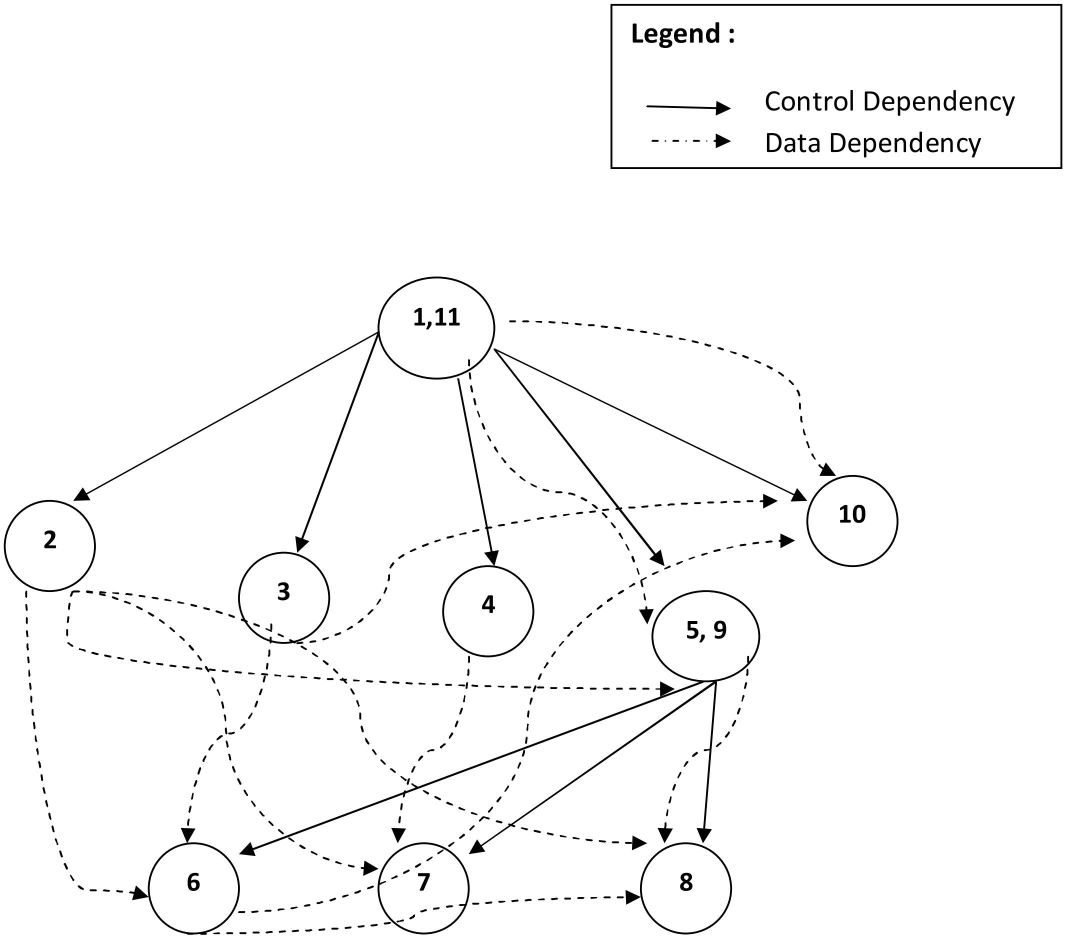


Figure 2.1: Dependence graph for the program given above

Chapter 4

Proposed Algorithm & Implementation

Forward Static Program slicing has got lot of advantages like here all general inputs are considered for the program. Hence for testing purpose it is very useful. Analyzing and reusing the code by the help of the program slices becomes easier by this technique. We see in larger programs there is redundant and repeated use of the same code. But the programmer is unaware of the problem. By the use of the technique the redundancy is minimized. And also by using forward slicing the state space is minimized to a greater extent. Automatic differentiation of variables that are used for a particular function call can be known easily.

4.1 Pseudocode for forward static slicing algorithm

Now we present our proposed algorithm for computing forward static slices for a simple program in the pseudocode form.

```
for  (each line)
while (  )
{   arr  = { temp,’var’ } ;
  off var  =  find ( arr ) ;
  if ( offset  ( = )<  offvar );
  {     print (currline );
    Skip space( before =)
   }
Temp[i] =  store string (till another space)
}
find(var)
{
     skip space( whole line);
    while ( )
             {
             check( var is between “ and “);
             if (yes )
             continue ;
             else
              {
               Check ( offvar - 1) and (off var  + 1) has only
                  { ‘-‘ , ‘ +‘ , ‘ *‘ , ‘ / ‘ %’ , ‘ ; ‘ ,’ =‘ }
             }
if (yes )
return offvar
else ( )
continue ;
}
```

4.2 Algorithm

We now will explain our algorithm in a step wise manner.

Input: A file containing c/c++ program ,a variable V and line number , n.

Output: A text file containing statements affected by the by the variable at line number n.

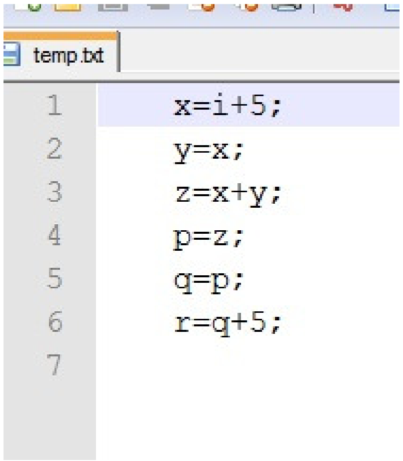
```
Step 1: Initialize the array var list to null
Step 2: Read the line no ( n ) & var ( v ). // taking input slicing criterion
 Step 3: Add v to the list of var list.
Step 4: while (current line no < n) do
Step 5: move to next line
Step 6: end while              // we reach at line n
Step 7: while (current line no < last line no) do
Step 8: current line = next line no
Step 9: for each variable in the list
Step 8:if var  is present as right side of '=' in the line then
Step 9: Add v to var array list
Step 10:  write line to the output file
Step 11: end if
Step 11 end while
```

4.3 Working of our algorithm

Considering a sample program which is c++ as input for our algorithm:

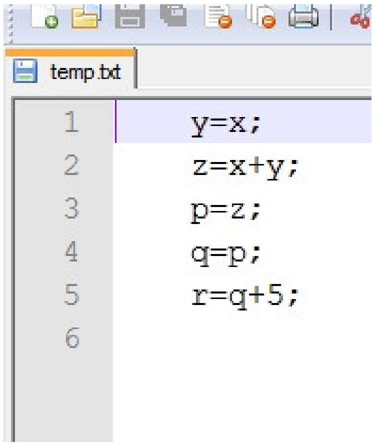
```
S1:  #include <iostream.h>
S2: int main()
S3:   {
S4:       int i,x,y,z,p,q,r;
S5:              i=3;
S6:              x=i+5;
S7:              y=x;
S8:              z=x+y;
S9:              p=z;
S10:             q=p;
S11:             r=q+5;
S12:             cout<<r;
S13:             return 0;
S14:}
```

The forward static slice w.r.t to Slicing criteria (5, i) is given below :



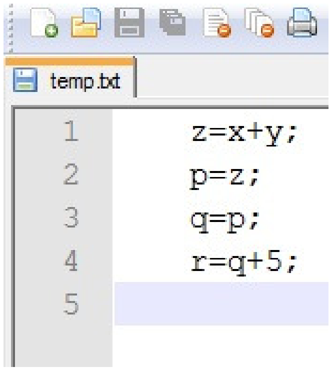
```
x=i+5;
y=x;
z=x+y;
p=z;
q=p;
r=q+5;
```

The forward static slice w.r.t to Slicing criteria (6, x) is given below :



```
y=x;
z=x+y;
p=z;
q=p;
r=q+5;
```

The forward static slice w.r.t to Slicing criteria (7,y) is given below :



```
z=x+y;
p=z;
q=p;
r=q+5;
```

4.4 Tool's Used

4.4.1 Netbeans

Netbeans is an open-source software development project with an active community of collaborating users and developers with multi-language software development environment comprising an integrated development environment and an extensible plug-in system. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages as well, including Java, JavaScript, PHP, Python, Ruby, Groovy, C, C++, Scala, Clojure and much more. For netbeans to be installed the only requirement is that JVM should be installed. We prefer the java for coding because of the support for object oriented features (which C does not have) and for dynamic memory allocation. [11]

4.4.2 Lex

The program Lex generates a so called `Lexer'. This is a function that takes a stream of characters as its input, and whenever it sees a group of characters that match a key, takes a certain action.[10]

Sample lex program :

```
%{
#include <stdio.h>
%}
%%
[0123456789]+ printf("NUMBER\n");
[a–zA–Z][a–zA–Z0–9]* printf("WORD\n");
%%
```

This Lex file describes two kinds of matches (tokens): WORDs and NUMBERs. Regular expressions can be pretty daunting but with only a little work it is easy to understand them. Let's examine the NUMBER match: [0123456789]+: This says: a sequence of one or more characters from the group 0123456789.Now, the WORD match is somewhat more involved: [a–zA–Z][a–zA–Z0–9]*

Sample inputs which we have taken :,

```
food
WORD
water
WORD
```

4.4.3 YACC

YACC can parse input streams consisting of tokens with certain values. This clearly describes the relation YACC has with Lex, YACC has no idea what 'input streams' are, it needs preprocessed tokens. While you can write your own Tokenizer, we will leave that entirely up to Lex.[10]

4.5 Results

We have tested our algorithm with 10 simple programs and we have measured the slice computation time for each of the program with different no of statements. Which are given below table:

Serial No.	Number of lines in the program	Average slicing time (nano seconds)
1	10	11751236
2	20	23866388
3	50	85045915
4	70	133942117
5	100	216877402
6	120	248737718
7	140	299392057
8	170	354438586
9	200	414309592
10	250	540031082

Figure 4.1: Table showing average slicing Vs number of lines

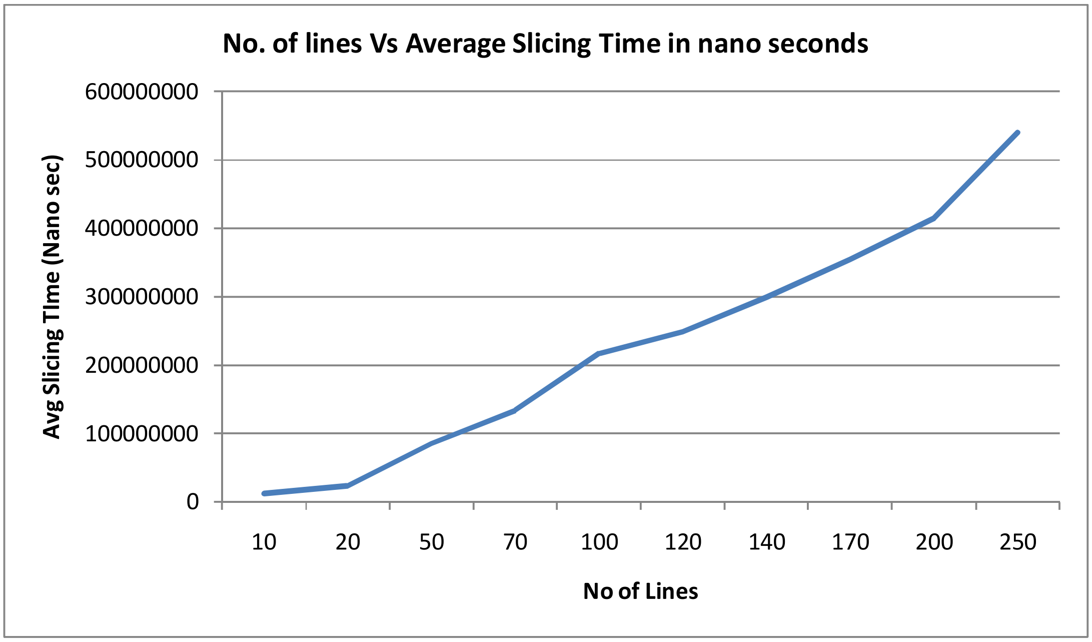


Figure 4.2: Graph of the slicing algorithm

From the graph we can conclude that as the size of the input program increases, the average slicing time also increases.

Chapter 5

Conclusion and future work

5.1 Conclusion

We developed an algorithm for computing forward static slices of simple programs which derives the slices which are affected by the variable. This slicing technique has various uses like analyzing and reusing the code which makes the task of the programmer easier. We see in larger programs there is redundant and repeated use of the same code. But the programmer is unaware of the problem. By the use of forward static slicing technique the redundancy is minimized and the state space is reduced to a greater extent. Automatic differentiation of variables that are used for a particular function call can be known easily. Apart from these there are a lot of advantages of this technique like easy debugging etc.

We have implemented our algorithm using java in netbeans IDE on windows platform and have tested our algorithm for 10 different programs and we observed that slice computation time increases when the number of statements increases.

5.2 Future work

- We have considered only the arithmetic statements in our algorithm for forward static slicing purpose. We have not considered other statements such as I/O statements, logical statements etc. so our algorithm can be extended to compute forward slices of programs containing these strings.
- We have not considered control statements & looping statements so hence that can be added to our algorithm.
- We have not considered object-oriented features such as inheritance, polymorphism etc. Our algorithm can be extended for computing forward slices of object-oriented programs.

References

[1] David Binkley and Keith Brian Gallagher, "Program slicing", Advances in Computers, Academic Press, Volume 43, pages 1–50, 1996.

[2] Mark Harman and Robert Hierons, "An overview of program slicing", Software Focus, Volume 2, Issue 3, pages 85–92, January 2001.

[3] Mark Weiser, "Program slicing", Proceedings of the 5th International Conference on Software Engineering, IEEE Computer Society Press, pages 439–449, March 1981.

[4] Mark Weiser, "Program slicing", IEEE Transactions on Software Engineering, IEEE Computer Society Press, Volume 10, Issue 4, pages 352–357, July 1984.

[5] Frank Tip. "A survey of program slicing techniques", Journal of Programming Languages, Volume 3, Issue 3, pages 121–189, September 1995.

[6] Andrea de Lucia. "Program slicing: Methods and applications", International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, pages 142-149, 2001.

[7] Jens Krinke, "Visualization of Program Dependence and Slices," IEEE International Conference on Software Maintenance, pages 168-177, 2004.

[8] B. Korel, J. Rilling, "Program Slicing in Understanding of Large Programs," 6th International Workshop on Program Comprehension, pages 145-167, 1998.

[9] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," ACM Trans Program Language System, Volume. 9, pages 319-349, July 1987.

[10] Lex & Yacc, Second Edition By Doug Brown, John Levine, Tony Mason, O'Reilly Media

[11] "Open Source Java Technology Debuts In GNU/Linux Distributions". Sun Microsystems, May, 2008.

[12] S. Horwitz and T. Reps. Efficient comparison of program slices. Technical Report 983, University of Wisconsin at Madison, 1990.

[13] B. Korel and J. Laski. Dynamic slicing of computer programs. Journal of Systems and Software, pages 198-195, 1990.