International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Optimizing Dynamic Dependence Graph

Toshi Sharma[1], Madhuri Sharma[2]

[1, 2]Computer Science and Engineering Department, Bharat Institute of Technology, Meerut, India

Abstract: ***A dynamic dependence graph is one of many techniques to extract a dynamic slice. Dynamic program slicing is very useful in debugging. This paper discusses about brief comparison of static and dynamic slicing, the dynamic dependence graph and its optimization algorithm and conclusion.***

Keywords: graph, dynamic slicing, program dependence graph, dynamic dependence graph, dependency matrix.

1. Introduction

The original concept of a program slice was introduced by Weiser [1, 2, 3, 4]. Program slicing is a technique to extract only those statements from the program, which affect a chosen set of variable also known as variable of interest 'VOI'. Slicing is used to reduce the size of a program by eliminating the statements that cannot affect the value of variable of interest. The reduced program is known as slice. We can also say that program slicing is program understanding or analysis technique. With the help of slicing the focus can be made only on a specific sub-component of a very large program.

Slicing is broadly classified into two categories i.e. static slicing and dynamic slicing. A static program slice S consists of all statements in program P that may affect the value of variable v at some point p [4, 5].The slice is defined for a slicing criterion C=(x,V), where x is a statement in program P and V is a subset of variables in P. A static slice, preserves the program's behavior (value of variable v) for all possible program executions.The exact terminology "dynamic program slicing" was first introduced by Korel and Laski [7, 8]. Dynamic slicing may very well be regarded as a non-interactive variation of Balzer'snotion of flowback analysis [8].A dynamic slice preserves the program's behavior for a specific program input, rather than for all program inputs where as a static slice preserves the program's behavior for all the program inputs. This paper presents the difference between static and dynamic slicing on the basis of statement coverage, further it explains the program dependence graph and dependency matrix and dynamic dependence graph along with its optimized approach.

2. Comparison of Static Slicing and Dynamic Slicing

```
begin:
S1: read(X)
S2: if(X<0)
 then
S3: Y:=f1(X);
S4: Z:=g1(X);
 else
S5: if(X=0)
 then
S6: Y:=f2(X);
S7: Z:=g2(X);
 else
S8: Y:=f3(X);
S9: Z:=g3(X);
 end_if;
 end_if;
S10: write(Y);
S11: write(Z);
 End
```

Figure 1: Example Program 1

In case of static slicing the slice for the slicing criterion (Y,10) in figure 3 would consists of the statements {1,2,3,5,6,8}. The total number of statements in the slice is 6. In case of dynamic slicing the slice would be computed on the basis of input value [3,4,5]. So the slice for the slicing criterion

(-1,Y,10) would consists of only three statements i.e. {1,2,3}.

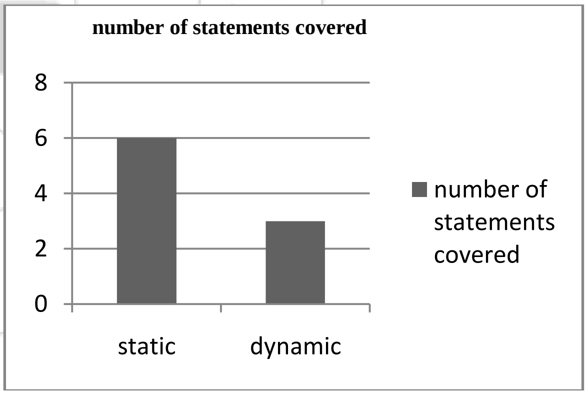

Figure 2: Comparison between static slicing and dynamic slicing

3. Program Representation

A. Program Dependence Graph

A graph G = (V, E) where V is the set of vertices and E is the set of edges formed by joining two vertices. A PDG represents the relationship between various statements of a program [1,4,6]. The nodes or a vertex in a PDG represents statements and the edges represent the dependency between the statements. There are two kinds of dependencies.

- Data Dependency
- Control Dependency

Data Dependency – An edge from node x to node y means that the computation at node y depends on the value computed at node x.

Control Dependency – An edge from node x to node y means that the computation of node y depends on the Boolean outcome at node x [6].

In order to plot the graph we need to have knowledge of the dependency matrices. PDG is the graphical representation of dependency matrices.

B. Dependency Matrices

The representation of a program in the form of matrix refers to as dependency matrix. There are separate matrix for control dependency and data dependency. The values or numbers in the matrix correspond to the statement number. With the help of these matrices we can make out a statement dependency on other statements. For eg. The dependency matrices for the program in Figure 1 is given below.

Data Dependency Matrix	*Control Dependency Matrix*
data_dependency =	control_dependency =
2 1	3 2
3 1	4 2
4 1	5 2
5 1	6 5
6 1	7 5
7 1	8 5
8 1	9 5
9 1	
10 3	
10 6	
10 8	
11 4	
11 7	
11 9	

Figure 3: Dependency Matrix

The first entry in the data dependency column is 2 1 it means that statement 2 is data dependent on statement 1. It means that statement 1 has some value which is used by statement 2. When we plot a graph the data dependency edge is constructed from 1 to 2.

Similarly by checking out this way we can easily make out the dependencies. By the help of these dependencies we can construct a graph. These matrices turn out to be very helpful when the graph grows huge. A bigger graph becomes very messy with the large number of edges, thereby making it really difficult to read the graph.

C. Dynamic Dependence graph

Dynamic dependence graph is modified and optimized version of the previously created dynamic slicing approaches. It solves the problem of multiple reaching definitions of the same variable used by the statement. The program dependence graph represents only the dependency of the statements. So in case if we have a program having multiple reaching definitions of the same variable and we use program dependence graph to extract a dynamic slice then the resulting dynamic slice would include those statements also which have not been executed [1,9]. The drawback of this approach was that the size of the graph becomes equivalent to that of the program. For e.g.,

```
begin
S1: read(N);
S2: I:=1;
S3: while(I<=N)
 do
S4: read(X);
S5: if(X<0)
 then
S6: Y:=f1(X);
 else
S7: Y:=f2(X);
 end_if;
S8: Z:=f3(Y);
S9: write(Z);
S10: I:=I+1;
 end_while;
end
```

Figure 4: Example Program 3

The program in Fig. 4 is checked for the test case N =3, X=-4,3,-2. First the execution trace is constructed for the given test case [9]. The data dependency and control dependency matrix are constructed for different values of X in different iterations.

For I=1 is $1,2,3^1,4^1,5^1,6^1,8^1,9^1,10^1$
For I=2 is 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2
For I=3 is 3^3, 4^3, 5^3, 6^2, 8^3, 9^3, 10^3 and 3^4

X=-4	X=3	X=-2
3 1	3 1	3 1
3 2	3 10	3 10
5 4	7 4	5 4
6 4	8 7	6 4
8 6	9 8	8 6
9 8	10 10	9 8
10 2		10 10

Figure 5: Data Dependency Matrix for the program in figure 4 for the test case N=3 and X=-4,3,-2

X=-4	X=3	X=-2
4 3	7 5	5 3
5 3	8 3	6 5
6 5	9 3	8 3
8 3	10 3	9 3
9 3		10 3
10 3		

Figure 6: Control Dependency Matrix for the program Fig.4 in for the test case N=3 and X=-4,3,-2

4. Proposed Work

Software is often modified to reflect new functionality, with the changes of its specification. In the modification, several bugs are usually injected and so debugging is an important task in software evolution. Program slicing and specifically dynamic program slicing is highly efficient in debugging. Dynamic Dependence Graph is a approach to find the dynamic slice. But the issue with dynamic dependence graph is that it gets very complex in case of bigger program . So in order to make the task of finding the faults easier, my thesis work focuses on lowering the number of nodes in dynamic dependence graph for program slicing i.e. optimizing the dynamic dependence graph.

The algorithm for an optimized dependence graph is as follows;

1. Taking the program as an input.
2. Input the slicing criterion i.e. <input (t), occurrence of statement (l), variable of interest (v)>.
3. Executing the input program against the given test case.
4. Find the trace of the program according to the test case.
5. Draw the dependency matrices of each iteration of the loop.
   - Construct the matrix for the first iteration including all the data dependency and control dependency.
   - Construct the matrix for the second iteration If the statement in inside if else condition has been included in the matrix for the previous iteration, then do not include that statement again in the matrix of other iteration
6. Repeat the above steps till the program terminates
7. Now construct the graph of the matrix
8. Construct a node for each statement labelling them with their respective statement numbers.
9. Draw the edges between the nodes with the help of matrices.
10. Once the graph is constructed we can find out the dynamic slice with respect to a variable, var by first finding out the last definition of variable 'v' and finding all the reachable statements

A. Implementation of the proposed work

For the program in figure 5 we are going to find an optimized version of the dynamic dependence graph using the proposed approach

X=-4	X=3	X=-2
3 1	3 1	3 1
3 2	3 10	3 10
5 4	7 4	9 8
6 4	8 7	10 10
8 6	9 8	
9 8	10 10	
10 2		

Figure 7: Data Dependency Matrix for the program in Fig. 4 for the test case N=3 and X=-4,3,-2

X=-4	X=3	X=-2
4 3	7 5	9 3
5 3	8 3	10 3
6 5	9 3	
8 3	10 3	
9 3		
10 3		

Figure 8: Control Dependency Matrix for the program in Fig. 4 for the test case N=3 and X=-4,3,-2

Now, compare the column X=-2 in figure 5 and 7 we can see that the matrix in figure 7 is smaller than in figure 5. This is because we have omitted those dependencies inside the loop which have already been included in previous iterations. Similarly the control dependency has been constructed.

5. Conclusions

The dynamic dependence graph has been optimized and this method can be used to develop a tool for slicing. Dynamic slicing is an important concept and it finds its application in areas like debugging [15, 24], testing [10,11,12,13,14], reverse engineering [15,16], software maintenance [16,17,18,19], program integration [20, 21]and software metrics [34,35].

The concept explained in proposed work can be used to develop a tool. Tool for dynamic slicing of java programs can be developed as most of the programming for software is done using java. The tool would provide a lot of help in debugging and testing.

References

[1] Frank Tip,A Survey of Program Slicing Techniques
[2] M. Weiser. Programmers use slices when debugging. Communications of the ACM, 25(7):446–452, 1982.
[3] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, 10(4):352–357, 1984.
[4] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. InProceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 177–184, 1984. SIGPLAN Notices 19(5).
[5] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations.In Conference Record of the Eighth ACMSymposium on Principles of Programming Languages, pages 207–218,1981.
[6] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319–349, 1987.
[7] B. Korel and J. Laski. Dynamic slicing of computer programs. Journal of Systems and Software, 13:187–195,1990.
[8] B. Korel and J. Laski. Dynamic program slicing. Information Processing Letters, 29(3):155–163, 1988
[9] Hiralal Agarwal and Joseph R.Horgan , "Dynamic Program Slicing, " ACM SIGPALN'90 Conference on Program Slicing Design and Implementation, White Plains, New York, June 20- 22 ,1990.
[10] D.W. Binkley, The application of program slicing to regression testing, in: M. Harman, K. Gallagher (Eds.), Information and SoftwareTechnology Special Issue on Program Slicing, Vol. 40, Elsevier, Amsterdam, 1998, pp. 583–594.
[11] R. Gupta, M.J. Harrold, M.L. Soffa, An approach to regression testing using slicing, in: Proc. IEEE Conf. on Software Maintenance, IEEE Computer Society Press, Los

[12] R.M. Hierons, M. Harman, C. Fox, L. Ouarbya, M. Daoudi, Conditioned slicing supports partition testing, Software Testing, Verification Reliability 12 (2002) 23–28. Alamitos, CA, USA, Orlando, FL, USA, 1992, pp. 299–308.
[13] M. Harman, S. Danicic, Using program slicing to simplify testing, Software Testing, Verification Reliability 5 (3) (1995) 143–162.
[14] R.M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification Reliability 9 (4) (1999) 233–262.
[15] H. Agrawal, R.A. DeMillo, E.H. Spafford, Debugging with dynamic slicing and backtracking, Software Practice Experience 23 (6) (1993) 589–616.
[16] G. Canfora, A. Cimitile, M. Munro, RE2: reverse engineering and reuse re-engineering, J. Software Maintenance: Res. Practice 6 (2) (1994) 53–72.
[17] D. Simpson, S.H. Valentine, R. Mitchell, L. Liu, R. Ellis, Recoup—maintaining Fortran, ACM Fortran Forum 12 (3) (1993) 26–32.
[18] G. Canfora, A. Cimitile, A. De Lucia, G.A.D. Lucca, Software salvaging based on conditions, in: Internat. Conf. on Software Maintenance(ICSM'96), IEEE Computer Society Press, Los Alamitos, CA, USA, Victoria, Canada, 1994, pp. 424–433.
[19] A. Cimitile, A. De Lucia, M. Munro, A specification driven slicing process for identifying reusable functions, Software Maintenance: Res.Practice 8 (1996) 145–178.
[20] S. Horwitz, J. Prins, T. Reps, Integrating non-interfering versions of programs, ACM Trans. Programming Languages Systems 11 (3) (1989)345–387.
[21] D.W. Binkley, S. Horwitz, T. Reps, Program integration for languages with procedure calls, ACM Trans. Software Eng. Methodology 4 (1)(1995) 3–35.
[22] J.M. Bieman, L.M. Ott, Measuring functional cohesion, IEEE Trans. Software Eng. 20 (8) (1994) 644–657.
[23] A. Lakhotia, Rule-based approach to computing module cohesion, in: Proc. 15th Conf. on Software Engineering (ICSE-15), 1993, pp. 34–44.
[24] J.R. Lyle, M. Weiser, Automatic program bug location by program slicing, in: Second Internat. Conf. on Computers and Applications, IEEEComputer Society Press, Los Alamitos, CA, USA, Peking, 1987, pp. 877–882