World Academy of Science, Engineering and Technology
International Journal of Mathematical and Computational Sciences
Vol:2, No:12, 2008

Efficient Program Slicing Algorithms for Measuring Functional Cohesion and Parallelism

Jehad Al Dallal

***Abstract*—**Program slicing is the task of finding all statements in a program that directly or indirectly influence the value of a variable occurrence. The set of statements that can affect the value of a variable at some point in a program is called a program slice. In several software engineering applications, such as program debugging and measuring program cohesion and parallelism, several slices are computed at different program points. In this paper, algorithms are introduced to compute all backward and forward static slices of a computer program by traversing the program representation graph once. The program representation graph used in this paper is called Program Dependence Graph (PDG). We have conducted an experimental comparison study using 25 software modules to show the effectiveness of the introduced algorithm for computing all backward static slices over single-point slicing approaches in computing the parallelism and functional cohesion of program modules. The effectiveness of the algorithm is measured in terms of time execution and number of traversed PDG edges. The comparison study results indicate that using the introduced algorithm considerably saves the slicing time and effort required to measure module parallelism and functional cohesion.

***Keywords*—**Backward slicing, cohesion measure, forward slicing, parallelism measure, program dependence graph, program slicing, static slicing.

I. Introduction

AT a program point p and a variable x, the slice of a program consists of all statements and predicates of the program that might affect the value of x at point p. Program slicing can be static or dynamic. In the static program slicing (e.g., [1]), it is required to find a program slice that involves all statements that may affect the value of a variable at a program point for any input set. In the dynamic program slicing (e.g., [2]), the slice is found with respect to a given input set. Many algorithms have been introduced to find static and dynamic slices. These algorithms compute the slices automatically by analyzing the program data flow and control flow. The process of computing the slices of a given procedure is called intra-procedural slicing [1]. The process of computing the slices of a multi-procedure program is called inter-procedural slicing [3]. This paper focuses on computing intra-procedural static slices.

Manuscript received April 23, 2007. The author would like to acknowledge the support of this work by Kuwait University Research Grant WI04/04.

Jehad Al Dallal is with Department of Information Sciences, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait (e-mail: jehad@cfw.kuniv.edu).

The basic algorithms for computing static intra-procedural slices follow three main approaches. The first approach uses data flow equations (e.g., [1,4]), the second approach uses information-flow relations (e.g., [5]), and the third approach uses Program Dependence Graphs (PDG) (e.g., [6]). Dependency graph-based slicing algorithms are in general more efficient than the algorithms that use data flow equations or information-flow relations [7].

Depending on the slicing purpose, slicing can be backward or forward [3]. In backward slicing, it is required to find the set of statements that may affect the value of a variable at some point in a program. This can be obtained by walking backwards over the PDG to find all the nodes that have an effect on the value of a variable at the point of interest. In forward slicing, it is required necessary to find the set of statements that may be affected by the value of a variable at some point in a program. This can be obtained by walking forward over the PDG to find all the nodes that might be affected by the value of the variable. In this paper, we are interested in both backward and forward slicing.

Program slicing is used in several software engineering applications, including program debugging [8], regression testing [9], maintenance [10], integration [11], and measuring program cohesion and parallelization [12]. Some of these applications, such as program debugging, regression testing, and measuring program cohesion and parallelization, require computing slices at different program points.

In program debugging, when an error is detected, it is required to slice the statements that can affect the program point at which the error is detected. In a typical programming, several errors are detected in each module in the system. Therefore, several slices at different points have to be calculated.

In regression testing, it is required to check that the modifications performed on the system have not caused unintended effects. Each modification might require changes at different program points and it is required to test the slices computed at each of these program points.

Different algorithms that use program slicing are introduced to measure the cohesion of a module in a program. Weiser [1] suggests computing slices for each variable at all program output statements. Longworth [13] suggests computing a slice for each variable in the module. Ott and Thuss [12] suggest computing a slice for each output variable in the module. The computed slices are used to find different metrics, including cohesion and parallelism. As a result, in order to compute the cohesion and parallelism of a module, it is necessary to compute several slices of the module.

The above program slicing applications are considered important in the software development process, and therefore, they need efficient slicing algorithms to speed them up. Unfortunately, no special algorithm has been introduced in the literature to serve the above program slicing applications. In this case, the same single-point-based slicing algorithms have to be applied several times, and as a result, the dependency graph has to be traversed several times. This introduces the need for slicing algorithms that compute all the required slices in a more efficient way.

In this paper, two algorithms are introduced to compute all possible static intra-procedural slices of a program. The first one computes the backward slices and the other one computes the forward slices. Each of the algorithms requires walking over the PDG only once. In addition, we compare experimentally the effectiveness of the introduced backward slicing algorithm over the single-point slicing approach (e.g., [1]) in computing the slices required to measure the parallelism and functional cohesion of a program function. The comparison is performed using 25 software modules selected from five software applications and shows that the introduced algorithm is more effective in terms of PDG traversal and execution time.

The paper is organized as follows. Section II overviews the problems of computing program slices and measuring functional cohesion and parallelism. The efficient algorithms for computing all static backward and forward slices are introduced in Sections III and IV, respectively. Section V explains a comparison study settings and reports the results. Finally, Section VI provides a conclusion and discussion of future work.

II. Background

This section overviews the problem of program slicing and the problem of measuring the functional cohesion and parallelism as follows.

A. Program Slicing

The 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. This means that the pointed node has the definition of the variable used in the other node. A control dependence edge between two nodes implies that the result of the predicate expression at the node pointed by the edge decides whether to execute the other node or not. Fig. 1 shows a C function example. The function computes the sum, average, and product of numbers from 1 to n where n is an integer value greater than or equal to 1. Fig. 2 shows the PDG of the C function example given in Fig. 1. The number associated with each PDG node is called a node identifier. For simplicity, in this paper, the node identifier indicates the line numbers of the statements that are represented by the node. Solid and dotted direct edges represent the control and data dependency edges, respectively.

```
1   void NumberAttributes(int n, int &sum,
            double &avg, int &product) {
2       int i=1;
3       sum=0;
4       product=1;
5       while (i<=n) {
6          sum=sum+i;
7          product=product*i;
8          i=i+1;
9       }
10      avg=static_cast<double>(sum)/n;
11  }
```

Fig. 1 C function example

Using the PDG shown in Fig. 2, we can obtain the backward and forward slices. For example, using a single-point slicing approach (e.g., [1]) to obtain the backward slice of variable *i* at line 5 of the C function given in Fig. 1, we first add the node that represents line 5 to the slice. This implies adding lines 5 and 9 to the slice. Then, we traverse the incoming edges to node 5 backwards and add lines represented by the nodes attached to the incoming edges to the slice. This results in adding lines 1, 2, 8, and 11 to the slice. The same process is performed for the nodes that represent the added lines of code until we reach nodes with no incoming edges. As a result, the backward slice calculated for variable *i* at line 5 contains the C function lines of code numbered 1, 2, 5, 8, 9, and 11.

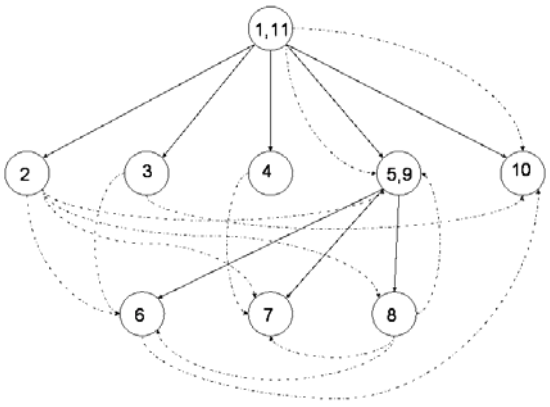

Fig. 2 PDG of the C function example given in Fig. 1

To obtain the forward slice of variable i at line 5 of the C function given in Fig 1, we first add the node that represents line 5 to the slice. This implies adding lines 5 and 9 to the slice. Then, we traverse the outgoing edges from node 5 forward and add lines represented by the nodes attached to the outgoing edges to the slice. This results in adding lines 6, 7, and 8 to the slice. The same process is performed for the nodes that represent the added lines of code until we reach

nodes with no outgoing edges. As a result, the forward slice calculated for variable i at line 5 contains the C function lines of code numbered 5, 6, 7, 8, 9, and 10.

To calculate all the backward or forward slices using a single-point slicing approach, the same slicing process applied at line 5 have to be applied at each line in the C function example given in Fig. 1. This implies applying the slicing process at each node given in Fig. 2, which results in traversing the PDG partially several times.

B. Measuring Functional Cohesion and Parallelism

Functional cohesion is defined as the relatedness of the module parts that contribute to different outputs. An output can be a single value output to a file or a device, an assignment to a global variable, or an output parameter. For example, the C function given in Fig. 1 has three outputs: *sum* at line 6, *product* at line 7, and *avg* at line 10. Bieman et al. [14] argue that using data tokens (i.e., variables and constant definitions and references) as the basis for slicing ensures that all changes of interest will cause a change of interest in at least one slice of the module. Changing an operator to a different one is an example of a change that is not of interest. To measure the functional cohesion and parallelism three steps are required including (1) computing the backward slices at each output variable lines of code, (2) computing the data slices for each output variable, and (3) applying metrics on the data slices to measure the functional cohesion and parallelism. To perform the first step for the function example given in Fig. 1, backward slices have to be computed at lines 6, 7, and 10. This results in having code statements at lines 1, 2, 3, 5, 6, 8, 9, and 11 in the slice compute at line 6, code statements at lines 1, 2, 4, 5, 7, 8, 9, and 11 in the slice computed at line 7, and code statements at lines 1, 2, 3, 5, 6, 8, 9, 10, and 11 in the slice compute at line 10. In the second step, the data slices are computed for each of the three output variables at lines 6, 7, and 10. Data slicing is performed by mapping a slice to the data tokens included in the slice. A data slice is a sequence of data tokens included in a slice. For example, the first column of Table I lists all the data tokens in the program given in Fig. 1, where T_i indicates the *i*'th occurrence of data token *T* in the function. The last three columns of Table I show the data slices for the three output variables. A cell in these columns is ticked if the data token is included in the slice computed for the output variable in Step 1. For example, the data token n_1 appears in the first line of the code given in Fig. 1. This line of code is included in the slices computed for the three output variables. Therefore, the cells in the row of the first data token in Table I are all ticked. The data slice for the variable *sum* at line 6 of the function given in Fig. 1 is a sequence of the data tokens: n_1,sum_1, avg_1, $product_1$, i_1, 1_1, sum_2, 0_1, i_2, n_2, sum_3, sum_4, i_3, i_5, i_6, 1_3.

Bieman et al. [14] introduce the concept of data slicing and used it as abstraction for measuring module functional cohesion. Bieman et al. define three terms, including *glue token*, *super-glue token*, and *glue stickiness*. The glue token is the data token that exists in more than one data slice. The super-glue token is the data token that exists in all data slices. The stickiness or adhesiveness of a glue token is the number of data slices that it binds. Three measures are introduced for a module, including *strong functional cohesion* (SFC), *weak functional cohesion* (WFC), and *adhesiveness* (A). The SFC is the ratio of the number of super-glue tokens to the total number of data tokens in the module. The WFC is the ratio of the number of glue tokens to the total number of data tokens in the module. Finally, the A of the module is the ratio of the total adhesiveness of all glue tokens to the total possible adhesiveness.

Ott and Thuss [12] introduce a technique to measure module parallelism. The technique requires computing a slice for each module output. Module parallelism is defined as the number of slices that are totally independent of all the other slices in the module.

TABLE I
DATA SLICE ABSTRACTION FOR THE SLICES COMPUTED AT LINES 6, 7, AND 10 OF THE C FUNCTION GIVEN IN FIG. 1

Data tokens	Data slices		
	sum	avg	product
n_1	x	x	x
sum_1	x	x	x
avg_1	x	x	x
$product_1$	x	x	x
i_1	x	x	x
1_1	x	x	x
sum_2	x	x	
0_1	x	x	
$product_2$			x
1_2			x
i_2	x	x	x
n_2	x	x	x
sum_3	x	x	
sum_4	x	x	
i_3	x	x	
$product_3$			x
$product_4$			x
i_4			x
i_5	x	x	x
i_6	x	x	x
1_3	x	x	x
avg_2		x	
sum_5		x	
n_3		x	

III. COMPUTING-ALL-BACKWARD-SLICES ALGORITHM

The algorithm for computing all intra-procedural static backward slices of a module is given in Fig. 3 and named *Compute-All-Backward-Slices* algorithm. Each node in the PDG is associated with an empty set before applying the algorithm. After the algorithm is applied, the set associated with a node n consists of the lines of code included in the slice computed at node *n*. The algorithm builds the set associated with each node in the PDG incrementally as the function called *ComputeABSlice* is applied recursively. The *ComputeABSlice* function takes a node n as an argument. If the node is not visited yet, the node is marked visited, the node identifier is added to the set associated with node *n*, and

all incoming edges to node *n* are traversed backwards. If an incoming edge is attached to a visited node *v*, the node identifiers included in the set associated with node *v* are added to the set associated with node *n*. Otherwise, if the incoming edge is attached to a node *m* not yet visited, node *m* is passed as an argument to the *ComputeABSlice* function. The function finds the set of nodes included in the backward slice computed at node *m*. After that, the node identifiers included in the set associated with node *m* are added to the set associated with node *n*.

Input: A PDG that has a single exit node, an empty set of node identifiers associated with each node, and all nodes contained in a cycle are combined in one node.
Output: The PDG that each of its nodes is associated with a set of identifiers of certain nodes. These certain nodes represent the lines of code contained in the computed backward slice.
Algorithm:

1. Mark all PDG nodes as "not visited"
2. *ComputeABSlice*(*exit* node)

```
ComputeABSlice(node n) {
    if node n is not visited
        Mark node n as visited
        Add the identifier of node n to the set associated
          with node n
        for each node m in which node n directly
          depends do
            ComputeABSlice(m)
            Add the contents of the set associated with
            node m to the set associated with node n
}
```

Fig. 3 The *Compute-All-Backward-Slices* algorithm

The algorithm requires performing four necessary preparations before applying the *ComputeABSlice* function as follows.

1. Adding an exit node to the PDG. The exit node is a node that has no outgoing edges. Since a node in a PDG represents a program code, an exit node represents a program code on which no other code depends. When the *Compute-All-Backward-Slices* algorithm is applied, the exit node is passed as an argument to the *ComputeABSlice* function as shown in the second step of the algorithm given in Fig. 3. Since a PDG can have several exit nodes, we are in need for a unique exit node to start with. As a result, a special exit node is added to represent the end of the program, or module, and a control flow edge is added from each of the exit nodes in the PDG to the added exit node. In the PDG given in Fig. 2, nodes 7 and 10 have no outgoing edges and, therefore, they are exit nodes. As shown in Fig. 4, a new node, node 12, is added to be the special exit node and two control dependence edges are added from nodes 7 and 10 to node 12. In this case, node 12 is first passed as an argument to the *ComputeABSlice* function to compute all static forward slices of the C function given in Fig. 1.
2. Combining all nodes contained in each cycle in the PDG in one node. Having a cycle between two or more nodes in the PDG implies that each of the nodes depends directly or indirectly on the other nodes in the cycle. This results in having same slice contents for each of the nodes in the cycle. Therefore, combining the nodes in a graph cycle in one node does not change the slicing results. However, having cycles in the graph leads to an infinite recursion when *ComputeABSlice* function is applied. Combining nodes in a cycle is performed by replacing the nodes by a new node. All incoming edges to each of the combined nodes are redirected to be incoming edges to the new node. Similarly, all outgoing edges from each of the combined nodes are redirected to be outgoing edges from the new node. Finally, any resulting self-loop edge is removed because such an edge is not considered when computing program slices. In the PDG given in Fig. 2, the two nodes that represent lines 5, 8, and 9 are contained in a cycle. Therefore, as shown in Fig. 4, the two nodes are replaced by the node labeled 5,8,9. All incoming edges to the nodes that represent lines 5, 8, and 9 are redirected to be incoming edges to the new node. All outgoing edges from the nodes that represent lines 5, 8, and 9 are redirected to be outgoing edges from the new node. This results in having two self-loop edges linked to the new node, and these edges are removed.
3. Associating an empty set with each node in the PDG. When the algorithm is applied, the set associated with each node contains the identifiers of the nodes that represent the program backward slice at the program point represented by the node.
4. Marking all nodes in the PDG as not visited. After applying the *Compute-All-Backward-Slices* algorithm and computing all backward slices, all nodes are marked visited.

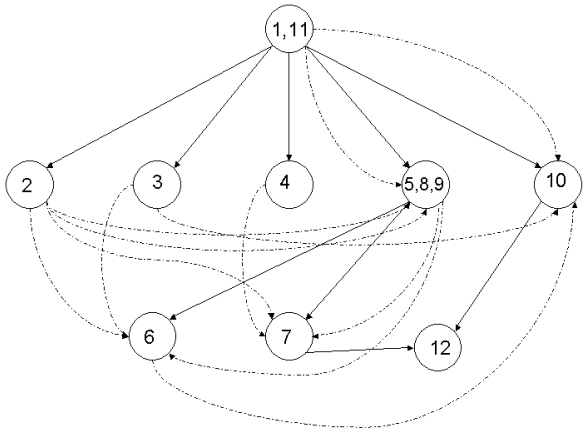

Fig. 4 The PDG prepared for applying the *Compute-All-Backward-Slices* algorithm. The PDG is derived from the PDG given in Fig. 2

Compute-All-Backward-Slices algorithm ensures that each edge is not traversed more than once by marking a traversed node as visited. Nodes are initially marked as *not visited*. Whenever a node is passed as an argument to *ComputeABSlice* function, it is checked whether it is marked previously as visited. If the node is not previously marked as

visited, *ComputeABSlice* function marks the node as visited and traverses all the incoming edges to the node. If the node is previously marked as visited, the *ComputeABSlice* function terminates without traversing the incoming edges. As a result, the incoming edges to any node are traversed once when the node is first passed as an argument to the *ComputeABSlice* function. Therefore, when the *Compute-All-Backward-Slices* algorithm is applied, no edges are traversed more than once.

For example, a backward slice is to be computed at each line in the C function given in Fig. 1. Fig. 4 shows the updated PDG as discussed formerly in this section. To compute the backward static slices, *Compute-All-Backward-Slices* algorithm is applied and Node 12 is passed as an argument to *CompueABSlice* function. Since Node 12 is initially marked as not visited, it is marked now as visited and the node identifier "12" is added to the slice set of node 12. Nodes 7 and 10 are linked by direct edges to Node 12, and therefore, *ComputeABSlice* function is applied to both of them. After computing their backward slices by recursively applying the *ComputeABSlice* function, the set of identifiers associated with each of the two nodes is added to the set of identifiers associated with Node 12. The resulting contents of sets of identifiers associated with each of the PDG nodes are listed in Table II. These contents are computed using *Compute-All-Backward-Slices* algorithm.

TABLE II
THE SLICE CONTENTS COMPUTED FOR EACH LINE OF CODE OF THE FUNCTION GIVEN IN FIG. 1. THE CONTENTS OF THE SLICES ARE COMPUTED USING *COMPUTING-ALL-BACKWARD-SLICES* ALGORITHM

Line of code	Slice contents
1	1,11
2	1,2,11
3	1,3,11
4	1,4,11
5	1,2,5,8,9,11
6	1,2,3,5,6,8,9,11
7	1,2,4,5,7,8,9,11
8	1,2,5,8,9,11
9	1,2,5,8,9,11
10	1,2,3,5,6,8,9,10,11
11	1,11
12	1,2,3,4,5,6,7,8,9,10,11,12

IV. COMPUTING-ALL-FORWARD-SLICES ALGORITHM

The algorithm for computing all intra-procedural static forward slices of a module is given in Fig. 5 and named *Compute-All-Forward-Slices* algorithm. Each node in the PDG is associated with an empty set before applying the algorithm. After the algorithm is applied, the set associated with a node n consists of the lines of code included in the slice computed at node *n*. The algorithm builds the set associated with each node in the PDG incrementally as the function called *ComputeAFSlice* is applied recursively. The *ComputeAFSlice* function takes a node *n* as an argument. If the node is not visited yet, the node is marked visited, the node identifier is added to the set associated with node *n*, and all outgoing edges form node *n* are traversed forwards. If an outgoing edge is attached to a visited node *v*, the node identifiers included in the set associated with node *v* are added to the set associated with node *n*. Otherwise, if the outgoing edge is attached to a node *m* not yet visited, node *m* is passed as an argument to the *ComputeAFSlice* function. The function finds the set of nodes included in the forward slice computed at node *m*. After that, the node identifiers included in the set associated with node m are added to the set associated with node *n*.

Input: A PDG that has a single entry node, an empty set of node identifiers associated with each node, and all nodes contained in a cycle are combined in one node.
Output: The PDG that each of its nodes is associated with a set of identifiers of certain nodes. These certain nodes represent the lines of code contained in the computed forward slice.
Algorithm:

1. Mark all PDG nodes as not visited
2. *ComputeAFSlice*(*entry* node)

```
ComputeAFSlice(node n) {
  if node n is not visited
      Mark node n as visited
      Add the identifier of node n to the set associated
        with node n
      for each node m that depends directly
      on node n do
          ComputeAFSlice(m)
          Add the contents of the set associated with
          node m to the set associated with node n
}
```

Fig. 5 The *Compute-All-Forward-Slices* algorithm

The algorithm requires performing three necessary preparations before applying the *ComputeAFSlice* function as follows.

1. Combining all nodes contained in each cycle in the PDG in one node as illustrated in Section III.
2. Associating an empty set with each node in the PDG. When the algorithm is applied, the set associated with each node contains the identifiers of the nodes that represent the program forward slice at the program point represented by the node.
3. Marking all nodes in the PDG as not visited. After applying the *Compute-All-Forward-Slices* algorithm and computing all forward slices, all nodes are marked visited.

Typically, each module has an entry point, and therefore, it is not required to add an entry node to the PDG. For the sample example considered in this paper, the resulting PDG prepared for applying *Compute-All-Forward-Slices* algorithm is similar to the PDG given in Fig. 5 with the exception of not having Node 12 and its incoming edges. Node 12 is not required here because it is an exit node added specifically to apply the *Compute-All-Backward-Slices* algorithm.

To compute the forward static slices for the PDG given in Fig. 4, *Compute-All-Forward-Slices* algorithm is applied and Node 1 is passed as an argument to *CompueAFSlice* function. Since Node 1 is initially marked as not visited, its marked now as visited and the node identifier "1,11" is added to the slice set of node 1. Nodes 2, 3, 4, 10, and the node that has an identifier "5,8,9" are linked by direct edges to Node 1, and therefore, *ComputeAFSlice* function is applied to each of these five nodes. After computing their forward slices by recursively applying *ComputeAFSlice* function, the set of identifiers associated with each of the five nodes is added to the set of identifiers associated with Node 1. The resulting contents of sets of identifiers associated with each of the PDG nodes are listed in Table III. These contents are computed using *Compute-All-Forward-Slices* algorithm.

TABLE III
THE SLICE CONTENTS COMPUTED FOR EACH LINE OF CODE OF THE FUNCTION GIVEN IN FIG. 1. THE CONTENTS OF THE SLICES ARE COMPUTED USING *COMPUTE-ALL-FORWARD-SLICES* ALGORITHM

Line of code	Slice contents
1	1,2,3,4,5,6,7,8,9,10,11
2	2,5,6,7,8,9,10
3	3,6,10
4	4,7
5	5,6,7,8,9,10
6	6,10
7	7
8	5,6,7,8,9,10
9	5,6,7,8,9,10
10	10
11	1,2,3,4,5,6,7,8,9,10,11

V. COMPARISON STUDY

The following comparison study shows the effectiveness of the Computing-All-Backward-Slices algorithm in computing the slices required for measuring the functional cohesion and parallelism. Twenty five functions selected from five software applications were considered in this comparison study. This section illustrates the comparison study settings and reports the results.

A. Comparison Study Settings

The comparison study uses 25 functions selected from five software applications developed by groups of senior undergraduate students for a project in a networking course. The project required building a simple client-server communicating program. All the applications were developed using C programming language, and they all have the same specifications. The study was initially conducted to compare the applications in terms of their functional cohesion and parallelism. Measuring the functional cohesion and parallelism is a computation- and labor-intensive task. Therefore, we developed a supporting tool that fully automates the functional cohesion and parallelism-measuring tasks for modules written in C programming language. The tool has four inputs, including the C program source code that includes the Module Under Consideration (MUC), MUC name, the PDG table file, and the symbol table file. The last two inputs are generated automatically using Aristotle Analysis System [15]. The PDG table file includes the data and control dependency information for all program functions. The symbol table file includes the global identifiers, function names, formal parameters, and non-global identifiers. The tool computes the required program and data slices, and measures the functional cohesion and parallelism of each function in each software application. For the purpose of this comparison study, two versions of the tool were developed. The first version computes only the required program slices using the single-point slicing approach, and the second version computes all program slices, including the required ones, using the *Computing-All-Backward-Slices* algorithm. Each version of the tool computes the slices and reports the slicing execution time and the number of PDG edges that were traversed to compute the slices.

The tool was executed on a Pentium 4 2.6GHz processor. To increase the collected slicing time accuracy, all unnecessary software applications running on the PC were switched off. When the tool was executed, it was noticed that the tool consumed micro-seconds to perform the slicing task. Since such little time is very sensitive and can change from one run to another, we have set the tool to start the timing clock, perform the required slicing task a million times, stop the timing clock, and report the average slicing time.

The purpose of this comparison study is to show the effectiveness of the *Computing-All-Backward-Slices* algorithm for applications that require computing more than one slice. Therefore, we have considered only the functions for which more than one slice is computed. This restriction results in considering 25 functions out of a pole of 50 functions included in the five considered software applications.

B. Comparison Study Results

Table IV shows the characteristics of the 25 functions selected for the comparison study and reports the comparison results. The first column of the table shows the function identifier in the form *application_number.file_name .function_name*. The second and third columns show the number of lines of code (not including comments and blank lines) and the number of computed slices for each function, respectively. The fourth column reports the number of nodes in the PDGs of the functions. The fifth and sixth columns report the number of PDG edges traversed using the single-point slicing approach and the *Computing-All-Backward-Slices* algorithm, respectively, during the functional cohesion and parallelism measuring processes. The sixth column also reports a percentage called *PDG Edge Percentage* (PDGEP), calculated as the percentage of the number of PDG edges traversed during the slicing process of the function using the *Computing-All-Backward-Slices* algorithm to the number of PDG edges traversed to perform the same task using the single-point slicing approach. Finally, the last two columns report the slicing execution time spent using the single-point slicing approach and the *Computing-All-Backward-Slices* algorithm, respectively. The last column also reports a

percentage called *Time Percentage* (TP), calculated as the percentage of the time spent during the slicing process of the function using the *Computing-All-Backward-Slices* algorithm to the time spent to perform the same task using the single-point slicing approach. The functions are ordered according to the number of slices required to measure the functional cohesion and parallelism of the functions.

TABLE IV
THE PDG TRAVERSED EDGES AND SLICING EXECUTION TIME RESULTS OF THE FUNCTIONS IN THE CLIENT-SERVER APPLICATIONS

Function identifier	Number of lines of code	Number of slices	Number of PDG nodes	Number of traversed PDG edges		Slicing time in micro-seconds	
				Using single-point slicing approach	Using *Computing-All-Backward-Slices* algorithm	Using single-point slicing approach	Using *Computing-All-Backward-Slices* algorithm
app1.client.CRC	19	2	29	76	49 (64.47%)	1.01	0.80 (79.63%)
app1.client.tc	5	2	7	14	8 (57.14%)	0.47	0.35 (75.02%)
app1.server.CRC	16	2	27	76	43 (56.58%)	1.08	0.67 (62.02%)
app1.server.CreateFrame	27	2	28	32	29 (90.63%)	1.06	1.21 (114.22%)
app3.hub.GenerateCRC	33	2	63	232	129 (55.60%)	2.65	1.81 (68.39%)
app4.channel_end.generate_CRC	34	2	50	162	92 (56.79%)	2.09	1.36 (65.19%)
app3.clientA.createFrame	58	3	75	185	132 (71.35%)	3.20	2.88 (90.03%)
app4.hub.create_data	15	5	29	39	34 (87.18%)	0.87	0.94 (107.39%)
app4.channel_end.create_i_frame	37	6	44	89	76 (85.39%)	1.73	1.86 (107.45%)
app4.hub.create_nak	10	6	12	18	13 (72.22%)	0.65	0.56 (86.53%)
app1.client.Input	62	7	116	192	189 (98.44%)	3.84	4.61 (120.16%)
app4.channel_end.create_ack	12	7	15	22	17 (77.27%)	0.72	0.64 (88.82%)
app5.centralhub.Server	24	7	37	44	38 (86.36%)	1.29	1.45 (112.81%)
app4.hub.print_frame	18	9	36	50	46 (92.00%)	1.08	1.34 (124.36%)
app3.hub.main	106	13	237	649	293 (45.15%)	10.17	8.22 (80.85%)
app2.cp_a.main	49	19	91	228	102 (44.74%)	4.20	3.15 (74.99%)
app2.cp_b.main	50	19	91	280	105 (37.50%)	5.07	3.15 (62.03%)
app3.clientA.main	150	19	304	2298	387 (16.84%)	33.53	9.68 (28.87%)
app1.client.main	153	24	264	1389	354 (25.49%)	21.21	9.44 (44.49%)
app4.channel_end.main	170	27	255	578	296 (51.21%)	11.79	9.22(78.18%)
app2.sp.main	71	33	139	211	145 (68.72%)	5.66	5.03(88.88%)
app5.endworkstation.main	168	33	266	2604	281 (10.79%)	35.60	8.75 (24.58%)
app4.hub.main	246	35	358	573	385 (67.19%)	14.13	12.73 (90.13%)
app5.centralhub.Client	176	42	290	2355	291 (12.36%)	36.97	10.00 (27.05%)
app1.server.main	412	53	735	15440	945 (6.12%)	230.63	26.56 (11.52%)
Average	85	15	144	1113	179 **(16.09%)**	17.23	5.06 **(29.35%)**

The results of the comparison study show that the percentage of the total number of traversed edges during the slicing process of the selected functions using the *Computing-All-Backward-Slices* algorithm is about 16% of the total number of edges traversed to perform the same task using the single-point slicing approach. In other words, the percentage of the reduction of the total number of traversed edges using the *Computing-All-Backward-Slices* algorithm is about 84%. Reducing the number of traversed PDG edges implies a reduction of the efforts involved in measuring the module parallelism and functional cohesion. Similarly, the results show that the percentage of the total time spent in the slicing process of the selected functions using the *Computing-All-Backward-Slices* algorithm is about 29% of the total time spent performing the same task using the single-point slicing approach. In other words, the percentage of the reduction of the time spent for the slicing process using the *Computing-All-Backward-Slices* algorithm is about 71%. Generally, the results of the comparison study indicate that it is more effective in terms of time and effort to apply the *Computing-All-Backward-Slices* algorithm to compute the program slices during the functional cohesion and parallelism measuring processes.

C. Observations and Discussion

During the analysis performed in the comparison study, we made the following observations:

1. For any of the analyzed functions, the PDGEP is always lower than the TP. This is due to the fact that the time required to traverse an edge using *Computing-All-Backward-Slices* algorithm is more than the time required to traverse an edge using a single-point slicing algorithm. For the analyzed functions, it was found the average time required during the traversal process of an edge using *Computing-All-Backward-*

Slices algorithm was 0.029 microseconds, whereas the time required for the same process using the single-point slicing algorithm is 0.021 microseconds. The *Computing-All-Backward-Slices* algorithm requires more time during the traversal process of an edge because when it traverses an edge it adds the contents of the set associated with the destination node of the traversed edge to the set associated with the source node of the traversed edge. This task is not performed when traversing an edge using the single-point slicing algorithm.

2. During the slicing process required for measuring the functional cohesion and parallelism of the analyzed functions, the number of traversed edges using the *Computing-All-Backward-Slices* algorithm is always less than the number of traversed edges using the single-point slicing algorithm. This is due to the fact that the slices required to measure the functional cohesion and parallelism are computed at the function outputs. Typically, all the statements in a function contribute to finding the function outputs. This means that all the edges of the PDG are traversed to compute the required slices. In some cases, the slices overlap because of the relatedness between the slices that contribute to the function outputs. This means that some or all of the PDG edges are traversed more than once using the single-point slicing algorithm. Since each PDG edge is traversed only once during the slicing process when using *Computing-All-Backward-Slices* algorithm, the number of traversed edges using the *Computing-All-Backward-Slices* algorithm is always less than the number of traversed edges using the single-point slicing algorithm.

3. There is no direct relationship between the *Computing-All-Backward-Slices* algorithm (in terms of saving the slicing time or effort) and the four characteristics of the analyzed functions: the number of lines of code, the number of outputs (equal to the number of required slices), the number of PDG nodes, and the number of PDG edges.

VI. Conclusions and Future Work

In this paper, two algorithms are introduced to compute all static backward and forward slices of a program by traversing the PDG that represents the program once. The algorithms use recursive functions to incrementally compute the slices as the PDG is traversed. The algorithms are useful for software engineering applications that require computing slices at different program points. In this case, the PDG is traversed once to find all slices instead of traversing the graph several times using other algorithms.

The paper addresses the effectiveness of using the *Computing-All-Backward-Slices* algorithm to compute the slices required in the functional cohesion and parallelism measuring processes of software modules. An experimental comparison study was conducted to determine whether it would be worthwhile to apply the *Computing-All-Backward-Slices* algorithm in the functional cohesion and parallelism measuring processes of software modules. The comparison study compared the results of using a single-point slicing approach and the *Computing-All-Backward-Slices* algorithm. The total number of traversed PDG edges and the slicing execution time were used as criteria in the comparison study. The comparison study results indicate that generally, using the *Computing-All-Backward-Slices* algorithm saves the effort applied and the time spent on the functional cohesion and parallelism measuring processes of software modules.

The introduced algorithms are limited to compute intra-procedural slices only. In future, we plan to extend the algorithms to consider inter-procedural slicing. In addition, we plan to extend the algorithms to compute all slices for object-oriented programs.

Acknowledgment

The author would like to acknowledge the support of this work by Kuwait University Research Grant WI04/04.

References

[1] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352-357, 1984.

[2] B. Korel and J. Laski, Dynamic slicing of computer programs, The *Journal of Systems and Software* , vol. 13, no. 3, pp. 187-195, 1990.

[3] S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, 1990.

[4] P. Hausler, Denotational program slicing, *In Proceedings of the 22nd Hawaii International Conference on System Sciences*, Hawaii, pp. 486-494, 1989.

[5] J. Bergstar and B. Carre, Information-flow and data flow analysis of while-programs, *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 37-61, 1985.

[6] K. Ottenstein and L. Ottenstein, The program dependence graph in software development environment, *In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, vol 19, no. 6, pp. 177-184, 1984.

[7] F. Tip, A survey of program slicing techniques, *Technical Report: CS-R9438, CWI (Centre for Mathematics and Computer Science)*, Amsterdam, The Netherlands, 1994.

[8] M. Weiser, Programmers use slices when debugging, *Communications of the ACM*, vol. 25, pp. 446-452, 1982.

[9] R. Gupta, M. Harrold, and M. Soffa, An approach to regression testing using slicing, *Proceedings of the International Conference on Software Maintenance*, pp. 299-308, 1992.

[10] K. Gallagher and J. Lyle, Using program slicing in software maintenance, *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751 – 761, 1991.

[11] S. Horwitz, J. Prins, and T. Reps, Integrating non-interfering versions of programs, *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, pp. 345-387, 1989.

[12] L. Ott and J. Thuss, Slice based metrics for estimating cohesion, *Proceedings of the IEEE-CS International Metrics Symposium*, pp. 78-81, 1993.

[13] H. Longworth, Slice based program metrics, *Master's thesis*, Michigan Technological University, 1985.

[14] J. Bieman and L. Ott, Measuring functional cohesion, *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 644-657, 1994.

[15] Aristotle Research Group, http://www-static.cc.gatech.edu/aristotle/Tools/, July 2006.

Jehad Al Dallal received his B.Sc. and M.Sc. in degrees in Computer Engineering from Kuwait University in Kuwait in 1995 and 1997, respectively. He received his PhD degree in Computer Science from University of Alberta in Canada in 2003.

He is currently working at Kuwait University, Department of Information Sciences as an Assistant Professor. His research interests include software testing and software analysis.