Detection of Parallelism in Sequential Programs Based on Functional Partitioning

1Banshidhar Majhi, 2Hasan Shalabi, 3Mowafak Fathi and 4Abbas M. Ali
1,3,4Department of Computer Science and Information Technology,
Al-Hussein Bin Talal University, Ma’an, Jordan
1On Leave from National Institute of Technology, Rourkela-8, Orissa, India
2Department of Computer Engineering, Al-Hussein Bin Talal University, Ma’an, Jordan

Abstract: Partitioning programs into smaller slices facilitates easy maintenance and testing of any software. A method for partitioning programs into segments performing single specific functions is required for simplifying the things. The notion of program slicing is observed to restrict the focus of a task to specific sub-components of a program. The slicing method utilizes information on data and control dependence and based on the concept of program slicing in order to carry out the task of partitioning programs. The tightly coupled program is formalized and it is shown that the tightly coupled statements performing single specific function can be extracted from a given program. After unraveling a sequential program into different specific functions executed in parallel environment and tested whether it is providing same output same as parent program.

Key words: Software engineering, program dependence graph, program slicing, program partitioning

INTRODUCTION

Program slicing is a program analysis and reverse engineering technique that reduces a program to those statements that are relevant for a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the value of a variable (v) at statement (s)?" Programmers have some abstractions about the program in mind during development. During the development following dependences from the statement s back to the influencing parts of the program. These statements may influence s either because they decide whether s is executed at all (control dependence) or because they define a variable that is used by s (data dependence). A program slicer can be used to automatically compute and visualize the slice of the program with regard to the statement s and the variables used or defined at s. It allows the programmer to focus his attention on the statements that are part of the slice and that might therefore contribute to the fault. However, in some cases the programmer may overlook the statements pertaining to a particular slice. Program slicing is an approach to assist the programmer during debugging, program integration, software maintenance, testing and software quality assurance. Several variants of program slicing have been proposed for these purposes, including static slicing, dynamic slicing, backward slicing, forward slicing, chopping, interface slicing, etc. A survey of existing program slicing tools shows that most of them are written for the programming language C, some for COBOL and FORTRAN[2]. Since program slicing is an interactive method intended to assist the programmer, the results should ideally be presented immediately. It is unsatisfactory to wait for several minutes before the slice can be viewed.

Slicing algorithms together with their data structures can be viewed as a data flow as well as graph-reachability problem[5]. Initially, Weiser[3] used a control flow graph as an intermediate representation for slicing algorithm and computed slices by solving the data flow problem of relevant nodes. Subsequently, he suggested algorithms for inprocedural and interprocedural slicing. However, since the interprocedural version did not account for the calling context, it produced imprecise slices. Ottenstein et al.[6] recognized that inprocedural backward slices could be efficiently computed using dependence graphs as intermediate representations by traversing the dependence edges backwards (i.e. from target to source). Horwitz et al.[7] introduced system dependence graphs for interprocedural slicing. They also suggested a two-phase algorithm that computes precise interprocedural slices.

The program slices introduced by Weiser[3] are called static program slices since it produces the slices irrespective of the input values. Therefore, it often results in producing relatively larger slices. To overcome these shortcomings, Korel and Lasky[8] introduced the concept

Corresponding Author: Dr. Hasan Shalabi, Department of Computer Engineering,
Al-Hussein Bin Talal University, P.O. Box 20, Ma’an, Jordan
Tel: 0096232179000 E-mail: shalabi@ahu.edu.jo
of dynamic slicing, which contains only those statements that actually affect the value of a variable at a program point for a given execution. Therefore dynamic slices are usually smaller than static slices and have been found to be useful in debugging, program understanding, maintenance, testing etc.\cite{10,13}.

In retrospect, the intra-procedural static program slicing based in interactively solving data flow equations representing inter-statement influences. Weiser\cite{16} presented a two-phase algorithm for computing inter-procedural slices. Ottenstein and Ottenstein\cite{18} presented a linear time solution to intra procedural static slicing in terms of graph reachability in the Program Dependence Graph (PDG). Horwitz et al.\cite{19} extended the PDG representation to System Dependence Graph (SDG) for interprocedural static slicing. Korel and Laski\cite{20} extended Weiser's static slicing algorithm to the dynamic case. They computed dynamic slices by solving the associated data flow equations. Their method needs $O(n)$ space to store the execution history and $O(n^2)$ space to store the dynamic flow data, where $n$ is the number of statements executed during the run of the program.

**PARALLELISM IN PROGRAMS**

Conventional uni-processor computers are programmed in a sequential environment in which instructions are executed one after another in a sequential manner. In fact the UNIX operating systems kernel was designed to respond to one system call from the user process at a time. Successive system calls are serialized through kernel. Most existing computers are designed to generate sequential object codes to run on a sequential computer. When using parallel computer, it requires a parallel environment where parallelism is automatically exploited. Language extensions or new constructs must be developed to specify parallelism or to facilitate easy detection of parallelism at various granularity levels by more intelligent compilers. OS also plays an important role to support parallel activities along with parallel languages and compilers. The OS must be able to manage the resources behind parallelism.

**Implicit parallelism:** This approach uses a conventional language such as C, FORTRAN and Lisp etc. to write the source program. The sequentially coded source program is translated into parallel object code by a parallelizing compiler as in Fig. 1. This compiler approach uses shared-memory multiprocessors. With parallelism being implicit, success relies heavily on the intelligence of a parallelizing compiler.

**Explicit parallelism:** This approach uses a parallel language version of C, Lisp and Pascal. Parallelism is explicitly specified in the user programs as depicted in Fig. 2. This will significantly reduce the burden on the compiler to detect parallelism. Instead, the compiler needs to preserve parallelism and where ever possible assigns target machine resources. This approach is adopted in a multiprocessor environment\cite{9,16}. Special software tools are needed to make an environment friendlier to user groups. Some of the tools are parallel extensions of conventional high level languages. Others are integrated environments which include tools providing different levels of program abstraction, validation, testing, debugging and tuning.
PROGRAM-DEPENDENCE GRAPHS AND PROGRAM SLICES

Different definitions of program dependence representations have been given, depending on the intended application; they are all variations on a theme introduced by Kuch et al. and share the common feature of having an explicit representation of data dependences. The program dependence graphs defined the additional feature of an explicit representation for control dependences. The definition of program dependence graph given below differs in two ways. First, our definition covers only a restricted language with scalar variables, assignment statements, conditional statements, while loops and a restricted kind of output statement called an end statement and hence is less general than the one. Therefore, we refer to our graphs as program dependence graphs, borrowing the term from. An end statement, which can only appear at the end of a program, names one or more of the variables used in the program; when execution terminates, only those variables will have values in the final state; the variables named by the end statement are those whose final values are of interest to the programmer.

The program dependence graph: The program dependence graph for program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges. The vertices of G_P represent the assignment statements and control predicates that occur in program P. In addition, G_P includes three other categories of vertices:

1. There is a distinguished vertex called the entry vertex.
2. For each variable x for which there is a path in the standard control-flow graph for P on which x is used before being defined, there is a vertex called the initial definition of x. This vertex represents an assignment to x from the initial state.
3. For each variable x named in P’s end statement, there is a vertex called the final use of x. It represents an access to the final value of x computed by P.

The edges of G_P represent dependences among program components. An edge represents either control dependence or data dependence. Control dependence edges are labeled either true or false and the source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex v_1 to vertex v_2, denoted by v_1 → v_2, means that, during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will eventually be executed if the program terminates. A method for determining control dependence edges for arbitrary programs is given by Thomas. However, because we are assuming that programs include only assignment, conditional and while statements, the control dependence edges of G_P can be determined in a much simpler fashion. For the language under construction here, the control dependences reflect a program’s nesting structure; program dependence graph G_P contains a control dependence edge from vertex v_i to vertex v_j of G_P iff one of the following holds:

1. v_i is the entry vertex and v_j represents a component of P that is not nested within any loop or conditional, these edges are labeled true.
2. v_i represents a control predicate and v_j represents a component of P immediately nested within the loop or conditional whose predicate is represented by v_i. If v_i is the predicate of a while-loop, the edge v_i → v_j is labeled true, if v_i is the predicate of a conditional statement, the edge v_i → v_j is labeled true or false according to whether u occurs in the then branch or the else branch, respectively.

A data dependence edge from vertex v_i to vertex v_j means that the program’s computation might be changed if the relative order of the components represented by v_i and v_j were reversed. Program dependence graphs contain two kinds of data dependence edges, representing flow dependences and def-order dependences. The data dependence edges of a program dependence graph are computed using data-flow analysis.

A program dependence graph contains a flow dependence edge from vertex v_i to vertex v_j iff all of the followings hold:

1. v_i is a vertex that defines variable X.
2. v_j is a vertex that uses X.
3. Control can reach v_j after v_i via an execution path along which there is no intervening definition of X. That is, there is a path in the standard control flow graph for the program by which the definition of X at v_i reaches the use of X at v_j. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph; final uses of variables are considered to occur at the end of the control-flow graph).

A program dependence graph contains a def-order dependence edge from vertex v_i to vertex v_j iff all of the followings hold:

120
1. \( v_1 \) and \( v_2 \) both define the same variable.
2. \( v_1 \) and \( v_2 \) are in the same branch of any conditional statement that encloses both of them.
3. There exists a program component \( v_3 \) such that \( v_1 \rightarrow v_3 \) and \( v_2 \rightarrow v_3 \).

A def-order dependence from \( v_1 \) to \( v_2 \) with “witness” \( v_3 \) is denoted by \( v_1 \rightarrow_{\text{def}} v_2 \).

Program Dependence Graph has different uses like detection of Parallelism, node splitting, code motion, loop fusion, program slicing etc. The dependence graph for the example program 1 is shown in Fig. 3.

**Example Program 1**

```c
1  #include<stdio.h>
2  void main()
3  {
4    int n, prod, x, sum;
5    scanf("%d", &n);
6    prod = 1;
7    sum = 0;
8    x = 1;
9    while(x<=n+1)
10    {
11      prod = prod*x;
12      sum = sum+x;
13      x = x+1;
14    }
15    printf("%d", prod);
16    printf("%d", prod);
17    printf("%d", sum);
18  }
```

**Program Slice:** A program slice with respect to a vertex \( v \) of a program dependence graph \( G_p \) is defined as:

\[ G_p/v = \{ V(G_p/v), A(G_p/v) \} \]

where, \( V(G_p/v) \) is a set of vertices on which \( v \) has a transitive flow or control dependence (i.e. all vertices that can reach \( v \) via flow or control dependence edges) i.e.

\[ V(G_p/v) = \{ [w \in V(G_p) \land w \rightarrow c, f] \}

\[ \emptyset, \text{ for any } v \not\in G_p \]

and

\[ A(G_p/v) = \{ w \rightarrow v \mid w \rightarrow c \in A(G_p) \land w, v \in V(G_p/v) \} \cup
\]

\[ \{ w \rightarrow v \mid w \rightarrow c \in A(G_p) \land w, v \in V(G_p/v) \} \cup
\]

\[ \{ w \rightarrow_{\text{def}} v \mid w \rightarrow_{\text{def}} c \in A(G_p) \land w, v \in V(G_p/v) \}
\]

This definition can be used to compute a slice \( S \) as:

\[ S = \bigcup_{i} s_{i} \land G_p/S = G_p \bigcup_{i} s_{i} = \bigcup_{i} G_p/s_{i} \]

**SLICING BASED ON FUNCTIONAL PARTITIONING**

Much of the literature on program slicing is concerned with improving the algorithms for slicing both in terms of reducing the size of the slice and improving the efficiency of slice computation. These works address computation of precise dependence information and accuracy of the computed slices. When a program performs multiple different functions, users or maintainers are often faced with the need of partitioning the program into programs performing single specific functions. For example, consider a program performs set of relatively unrelated functions, one of which is explicitly selected according to the interpretation of some parameter of the calling program. In order to utilize this kind of program in a given environment appropriately, users must understand the roles of the parameters of the module, which is bound to increase the complexity of the program. In contrast, programs that perform single specific functions are easy to understand and manage because changes are easy to understand and manage because changes or errors can easily be isolated. Therefore, it is desirable to restructure programs performing multiple different functions in order to perform single specific functions.

The need of partitioning a program into programs performing single specific functions also arises while the program is being verified. For verification of a program,
requirements must be stated in a formal language and each program statement must be examined in a step of inductive proof to ensure that the program will produce the correct outcome for all possible input sequence. However, such proofs are very expensive and have been applied only to small programs. One way to ease the task of correctness proving is to partition a program into meaningful smaller units in such a way that the different aspects of the program can be verified independently by different persons. Further more, partitioning the programs performing single specific functions may improve the performance when the program is executed on a parallel machine. In a parallel machine, all processors must be used at all times to keep utilization at maximum level. If a program can be partitioned into independent pieces (in a sense of performing single specific functions), then we can assign each partitioned program to one processor to result in speed up than executing the program as a whole on one processor.

We have considered the case of intra-procedural static slicing with functional partitioning approach. Functional partitioning approach utilizes the PDG to partition a multifunction program into programs performing single specific functions, if exists. Side by side sufficient care is taken to keep the semantic equivalence of the original program. The proposed method is based on the assumption that program P can be considered as a function \( F(X, Y) \), Where, \( X = \langle x_1, x_2, \ldots, x_p \rangle \) input vector and \( Y = \langle y_1, y_2, \ldots, y_m \rangle \) output vector. Execution of the statements in P may cause the values of \( x_1, x_2, \ldots \), \( x_p \), \( p: m \) to be changed and propagated to the final value of some output variables \( y_i \) (i = 1,2,3,...,m). If all the input variables and output variables and output variables other than \( y_i \) are relevant to the computation of the final value of \( y_i \), then P is regarded as a program performing a single specific function \( F(X, y_i) \). Here the output variables other than \( y_i \), as well as auxiliary program variables participate in the computation of the final value of \( y_i \) and all statements in P possibly influence the final value of \( y_i \). Otherwise, P is considered to be a program performing multiple different functions. Before actually going to functional partitioning, let us discuss program slicing in more detailed and mathematically. This is because program slicing is used in functional partitioning.

**Definition 1:** For a program P the statements of P are tightly coupled iff the following condition is satisfied: There exists an output statement \( O_i \) such that

\[ O = \{ W | W \rightarrow^* O_i \} \cup \{ O_k \} \]

where, \( O \) is the set of all output statements of P.

---

**Fig. 4:** Functional dependency diagram of program 1

First we check whether the program is tightly coupled or not. If the program is not tightly coupled, there exist functionally independent output statements \( O_1, O_2, \ldots, O_m \) and then the slices of that program can be computed.

**Definition 2:** Functional Partitioning: For vertices denoting output \( O_i \) and \( O_j \) (\( i != j \)) of a PDG \( G_n \), \( O_i \) is directly functionally dependent (d) on \( O_j \), denoted as \( O_j \rightarrow_d O_i \), if there exist vertices \( v \in V(G_{i/O}) \) and \( v \in V(G_{j/O}) \) such that \( v \rightarrow \gamma \) and \( \gamma \) is a statement that assigns a value to a variable in \( O_i \). Otherwise, \( O_i \) is said to be directly functionally independent of \( O_j \). The functional dependence is transitive that is if \( O_i \) is directly functionally dependent on \( O_k \) and \( O_k \) is directly functionally dependent on \( O_j \), Figure 4 depicts the functional dependence relations of program 1.

The above program performs \( F \) (n, x, prod, sum). Statement 16 is functionally dependent on 15 as \( 8 \in \nu \) \( (G_{y/15}) \), \( 11 \in \nu \) \( (G_{x/16}) \) and \( 8 \in \gamma \). Similarly, 17 and 15 are functionally dependent. But statement 16 and 17 are functionally independent. Thus the program can be divided into two sub-partitions, one performing \( F \) (n, prod) and other one \( F \) (n, sum). The two partitions can be computed by the program slicing approach i.e. \( V \) \( (G_{x/16}) \) and \( V \) \( (G_{y/17}) \). The two partitions are given below.

**Partition 1:**
```c
#include<stdio.h>
2 void main()
3 {
4 int n,prod,x;
5 scanf("%d",&n);
6 prod = 1;
7 x = 1;
8 while(x<n+1)
```

**Partition 2:**
```c
#include<stdio.h>
2 void main()
3 {
4 int n,sum;
5 scanf("%d",&n);
6 sum = 0;
7 x = 1;
8 while(x<n+1)
```
The control structures do not support complex expressions in its predicate i.e. while(x! = 5 & y! = 6) is not allowed.

**Assignment statements:** The assignment statement is of the form \(x = y + w/z\). The number of right variables should be less than or equal to 5.

**CONCLUSIONS**

In this study, a simple but useful method of partitioning programs on functional basis is presented. The concepts of data, control and functional dependencies are discussed thoroughly. The construction of PDG and its importance is discussed in detail. Exhaustive simulation has been carried out to test different programming features of C language and partitions are generated wherever possible. Even though, partitions of a program are basically used here to execute in parallel, it can be used in software complexity reduction and testing. Further, to incorporate the completeness all other features of the programming language can be included.

**REFERENCES**