Technical Report of Osaka University, Department of Information and Computer Scineces, IIP Lab, IIP-06-05-00, June 5 2000.

Lightweight Semi-Dynamic Methods for Efficient and Effective Program Slicing*

Katsuro Inoue†, Fumiaki Ohata†, Yoshiyuki Ashida†
† Osaka University

June 5, 2000

Abstract

When we try to debug a large program effectively, it is very important to separate a suspicious program portion from the overall source program. Program slicing is a promising technique to extract a program portion; however, it remains difficult issues. Static slicing sometimes produces a large portion of the source program, especially for a program with arrays and pointers. Dynamic slicing requires unacceptably huge run-time overhead. In this paper, we discuss intermediate semi-dynamic methods between static and dynamic slicing. We propose two slicing methods named call-mark slicing and dependence-cache slicing. These algorithms have been implemented in our experimental slicing system, and execution data for several sample programs have been collected. The result shows that call-mark slicing reduces the slice size by about 10–20% from the static slice size, with very little overhead increase. Also, dependence-cache slice reduces by about 30–90%, even for programs using arrays, with affordable run-time overhead increase. These slicing methods will be important features for effective debugging environments.

Author address: Katsuro Inoue
Department of Informatics, Graduate School of Engineering Science,
Osaka University, 1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
Ph. +81-6-6850-6570 Fax +81-6-6850-6574 E-mail inoue@ics.es.osaka-u.ac.jp

*This work is partly supported by Ministry of Education, Science, Sports, and Culture, grant-in-aid for priority areas "Principles for Constructing Evolutionary Software".

1 Introduction

Finding faults in source programs is a very time-consuming activity in software testing and maintenance phases. Looking over all of the source programs to find a fault is inefficient. We would like to focus our attention to a specific portion of the source programs to improve efficiency.

As a candidate of the focusing aids, program slicing techniques[27, 28] have been studied. A program slice is intuitively a collection of program statements which affect the value of a variable in a statement we are interested in. We can concentrate our attention only on the statements in the slice so that we would be able to effectively debug the source program.

We have investigated the effectiveness of program slicing for program debugging and maintenance processes using a controlled method[22]. The bug-finding time was measured and compared between two independent groups of programmers, where one group of subjects used an ordinary debugging tool and the other used the debugging tool with static slicing features. Each subject was given a fault-injected program and an associated test data set that effectively detected the faults. The average bug-finding times were 165 minutes without the slicing, and 122 minutes with the slicing. The effectiveness of the slicing was confirmed statistically.

A lot of researches and applications for program slicing have emerged[5, 6, 7, 10, 14] from the original work of Mark Weiser[27, 28]. These slicing techniques are roughly categorized into two classes, static slicing and dynamic slicing.

Static slicing was first proposed by Weiser[27]. A static slice is a collection of program statements possibly affecting a variable's value at a particular program point. The variable of interest and the program point of interest are called the slicing criterion. Static slicing extracts portions from an original program; however, the resulting portions are still large in many cases. In extreme cases, there is no reduction after taking a static slice. This is due to its analysis nature such that it must consider all possible input data and all possible control flows.

Also, they remain many difficult issues on the analysis, i.e., aliasing of variable names, separation of array and structure data elements, and tracking of pointer variables. In addition, object oriented programs, recently prevalent widely, cause the static analysis much harder, since those programs tend to contain a lot of small methods (procedures or functions) with dynamically

bind names.

Dynamic slicing was proposed by Agrawal et. al.[1, 2, 16, 17]. A dynamic slice is a collection of executed program statements actually affecting a variable's value at a particular program point. Since the dynamic slicing is based on an execution instance for the source program with a specific input data, non-executed parts of the source program are automatically excluded, resulting in a slice that is generally smaller than a static one. However, computing a dynamic slice is costly, requiring significant memory and time resources because of dynamic variable dependences that must be tracked.

For the purpose of bug finding, we would prefer the dynamic slicing, since the dynamic slicing gives us a narrower focus on the source program.

There have been a few proposal of reducing overhead of dynamic slicing[1]. However, we do not know how effective those methods are in comparison with dynamic and static methods.

There are various design space for choosing lightweight semi-dynamic methods which use static analysis information with lightweight dynamic information. In this paper, we propose two methods in such design space. One method focuses on collecting execution path information, and another focuses on collecting data dependences.

The contributions of this paper are summarized as follows.

- Classification and categorization of various algorithms between static and dynamic slicing methods are presented.
- Two intermediate slicing algorithms, call-mark slicing[23] and dependence-cache slicing are proposed, which are lightweight and semi-dynamic methods. These algorithms require only limited size memory for recording execution information.
- Various empirical data for slicing sizes, and analysis and execution times are collected using our Osaka Slicing System. Through this experiment, execution overhead and analysis cost are discussed.

The experiment result shows that the lightweight semi-dynamic methods reduce the sizes of slices, compared to the static ones. Especially dependence-cache slicing reduces the slice size greatly. The runtime overhead of these methods are small one acceptable to practical use. Although these results are based on our experimental slicing system, which limits the target language

and the source programs, we would think that the lightweight semi-dynamic slicing methods will be very useful to practical debugging environment.

In Section 2, we will briefly overview static and dynamic slicing, and will classify various analysis techniques between static and dynamic methods. Section 3 will present call-mark slicing, and Section 4 will propose dependence-cache slicing. We will show our experiment using Osaka Slicing System in Section 5. In Section 6, our findings will be discussed associated with related works. We will conclude this paper with some remarks in Section 7.

2 Classification of Slicing Methods

2.1 Overview of Static and Dynamic Slicing

In this section, we will briefly show static slicing and dynamic slicing for further discussions.

2.1.1 Static Slicing

Consider statements s_1 and s_2 in a source program p. When all of the following conditions are satisfied, we say that a *control dependence, CD*, from statement s_1 to statement s_2 exists:

- s_1 is a conditional predicate, and
- the result of s_1 determines whether s_2 is executed or not.

This relation is written by $s_1 \dashrightarrow s_2$.

When the following conditions are all satisfied, we say that a *data dependence, DD*, from statement s_1 to statement s_2 by a variable v exists:

- s_1 defines v, and
- s_2 refers to v, and
- at least one execution path from s_1 to s_2 without re-defining v exists.

This relation is denoted by $s_1 \xrightarrow{v} s_2$.

A *Program Dependence Graph*(PDG) is a directed graph whose edges denote dependences between statements, and whose nodes denote statements in a program such as conditional predicates, assignment statements, and so on. For a Pascal source program shown in Figure 1 (which computes an absolute value of the squared or cubed value selected by an input), we have a PDG presented in Figure 2. To handle function/procedure calls, we employed additional nodes for input and output parameters.

A *Static Slice* with respect to a variable v on a statement s (this pair (v, s) is called a *slice criterion*) in a program is a collection of statements corresponding to the nodes which possibly reach s using v and other transitively traversal CD and DD relations. The static slice of variable d at line 24 as the slice criterion for the program shown in Figure 1 is all statements except for the message output statements (lines 12, 14, 16) as shown in Figure 3.

2.1.2 Dynamic Slicing

Consider an execution trace e of a source program p for an input data d. s_i is a program statement appearing in e, and it indicates a point during an execution of p with d.

A *dynamic slice* p' with respect to s_i, d, and a variable v is a syntactic correct subset of p, which computes the same value of v for d at execution point s_i' that corresponds to s_i. A triple (d, s_i', v) is also called a *slice criterion* of a dynamic slice.

A dynamic slice is computed first by analyzing and storing the actual data and control dependences of variables in association with the program execution. Using this dependence chain, all statements in e, which affect the value of v at s_i, are extracted. Then p', which generates the same execution trace as this extracted trace, is reconstructed.

Figure 4 shows a dynamic slice of the program shown in Figure 1. The slice criterion is input data ($a = 2, b = 3, c = 0$), line 24 (of the last instance), and variable d.

Dynamic slicing is based on a single execution path, and it gives narrower slices than static slices. This nature is preferable in the debugging situation, since we could concentrate on our attention to the smaller slices.

```
 1  program Square_Cube(input,output);
 2  var a,b,c,d : integer;
 3  function Square(x : integer):integer;
 4  begin
 5     Square := x*x
 6  end;
 7  function Cube(x : integer):integer;
 8  begin
 9     Cube := x*x*x
10  end;
11  begin
12    writeln(``Squared Value ?'');
13    readln(a);
14    writeln(``Cubed Value ?'');
15    readln(b);
16    writeln(``Select Feature!  Square:0  Cube: 1'');
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20    else
21       d := Cube(b);
22    if (d < 0) then
23       d := -1 * d;
24    writeln(d)
25  end.
```

Figure 1: A Sample Source Program

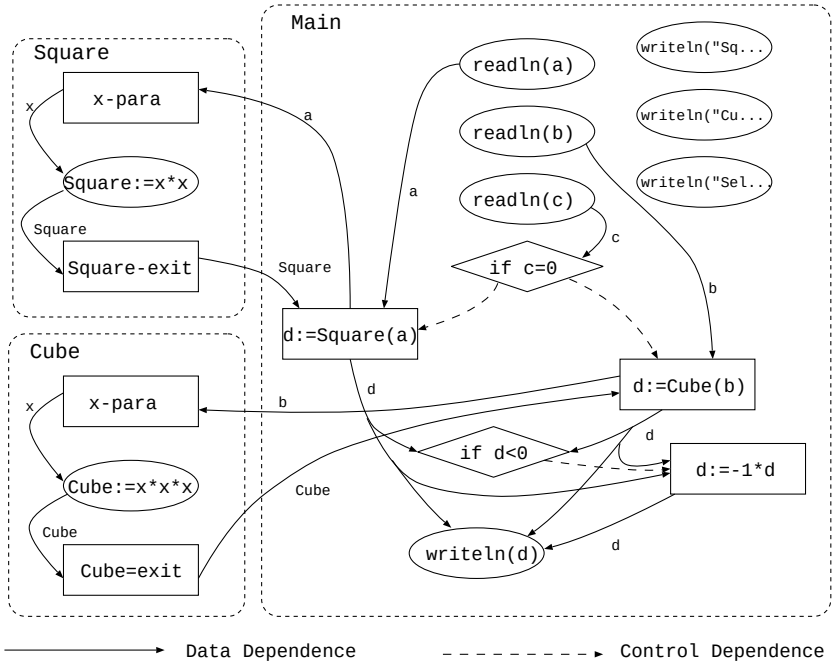

Figure 2: Program Dependence Graph (PDG) of Program Shown in Figure 1

```
 1  program Square_Cube(input,output);
 2  var a,b,c,d : integer;
 3  function Square(x : integer):integer;
 4  begin
 5     Square := x*x
 6  end;
 7  function Cube(x : integer):integer;
 8  begin
 9     Cube := x*x*x
10  end;
11  begin
12
13    readln(a);
14
15    readln(b);
16
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20    else
21       d := Cube(b);
22    if (d < 0) then
23       d := -1 * d;
24    writeln(d)
25  end.
```

Figure 3: Static Slicing Result by d at Line 24

```
 1  program Square_Cube(input,output);
 2  var a,b,c,d : integer;
 3  function Square(x : integer):integer;
 4  begin
 5     Square := x*x
 6  end;
 7
 8
 9
10
11  begin
12
13    readln(a);
14
15
16
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20
21
22
23
24    writeln(d)
25  end.
```

Figure 4: Dynamic Slicing Result by d at Line 24 with input ($a = 2, b = 3, c = 0$)

2.2 Classification of Slicing Methods

As mentioned in Section 1, various kinds of slicing method have been proposed and studied. However, there are little efforts to categorize them. In this section, we would try to sort out them from a view point of analysis efficiency and effectiveness under bug-finding phase. In this viewpoint, we are interested in narrower focus on the source program with smaller analysis time.

2.2.1 Analysis Based on Statical Flow Prediction

At first, we explore static slicing methods. The difficulties of the static methods lie on finding precise data dependencies, rather than control dependencies. Control dependencies are relatively easily determined by simple syntactic analyses of the source programs. On the other hand, data dependencies are not easily determined since we cannot determine a single program execution path statically (i.e., without input data).

If we are able to know the execution order of statements, we would predict which statement assigns the value of a variable and which statement refers to that value, and thus we can determine data dependencies (Def-Use relation[3]). There are roughly two issues of execution path prediction in the static approach[9].

- Flow Sensitiveness

  In most static slicing algorithms, the control flow is first analyzed, and the results are used by the following data flow analysis, as various compiler optimization techniques[3]. This type of slicing algorithm is called *flow sensitive* method, and it is commonly used.

  An extreme method of reducing the overhead of this control flow analysis, is proposed, in which we do not analyse the control flow at all. We assume all possible data dependencies between the definition statements of a variable and its reference statements. This method, called *flow insensitive* method, simplifies the analysis process, although the resulting slice size would be increased due to the increase of spurious data dependencies.

- Context Sensitiveness

For the analysis of multiple procedures and functions in target source programs, there are two different methods proposed and implemented. For each instance of procedure/function activation statement, the body of the procedure/function is analyzed along with its actual parameters and global variable information at caller statements. This method is called, *context sensitive* analysis[14]. In this method, we might have to repeat the analysis of a procedure/function with multiple caller statements.

Another method, in which procedures and functions are analyzed independently from actual call statements, is called *context insensitive* analysis. This method would be much simpler than the context sensitive method; however, the precision of analysis results is limited and we would get generally larger slice results than the context sensitive method.

As a basis of our discussion, we assume here static slicing algorithm with flow sensitive and context insensitive ones, without explicit note. This method is fairly straightforward and practically efficient one[25].

In this method, control flow of the program is computed through the syntax analysis of the source program. However, since the analysis is static one, we cannot generally determine which one of branches of an *if* statement statement is selected at execution time. Therefore, we have to assume all possible control flows, and this would increase the number of data dependencies, and then the resulting slice will be enlarged. This is a fundamental limitation of static slicing method.

2.2.2 Dynamic Execution Path and Data Dependence Analyses

In debugging environment we assume, program slicing techniques are used to identify faulty statements. In such cases, there are test cases and they specify execution paths of the program for specific input data. Thus, we are able to use information of the program execution for getting more narrower slices more efficiently.

As examples of this approach, dynamic slicing methods have been proposed[1, 2, 16, 17], although its overhead at program execution is fairly large. This type of dynamic slicing algorithm basically requires all history of program execution including its execution path and status of variables. To do this, we

would have to prepare history space proportional to the program execution length.

The following is a list of intermediate semi-dynamic methods which add some dynamic information to static one for improving slicing precision and efficiency.

- Collecting Execution Path

  We collect information of the program execution path (flow). The point is that we would effectively get information of the execution path with very lightweight run-time overhead.

  - Profiling Method

    This approach is proposed as a simplified method of dynamic slicing by Agrawal and et. al[1]. We delete nodes in PDG (which is made statically), when those statements (nodes) are not executed. This method can be implemented by checking whether each program statement is executed or not, and it does not require long history of program execution.

  - Call Mark Slicing Method

    We propose this method which will be described in Section 3 in detail. This method is intended to reduce the runtime overhead compared to the profiling method. Call-mark method only checks if each procedure/function call statement is executed or not. Using this information, portions of a source program which are not executed are determined and deleted from the slices. Candidate statements for the possible deletion with respect to each procedure/function call statements are statically analyzed.

  - Hybrid Slicing Method (Partial Trace by break point and call history)

    This method record execution history of break points which users have set or call history of each procedure/function activations[12]. Using this history, an execution path is predicted. To get effective execution path information, the users have to set break points at effective points in the program. Or we would have to record activation history of procedure and function activations.

- Collecting Data Flow Information

  If we could statically predict an execution path of a program, data dependencies of scalar type variables are easily determined. However, those of array and pointer type variables are not determined by simple knowing it.

  Consider the following a program portion. In this program, it is easy to know that the execution path is straightforward. However, we cannot determine that statement 4 depends on 1 or 2.

```
1: a[0]=0
2: a[1]=1
3: input(i)  % Assume integer 0 or 1 as an input
4: c=a[i]
```

  In order to resolve this, we need full execution history by which the array indices are restored and data dependencies are determined. In usual, DDG (Dynamic Dependence Graph) is used for it. This DDG keeps all of variable dependencies in its structure[1]. This method is easily established, although the overhead of constructing full DDG is extremely expensive.

  There are a few proposal to compromise between data dependence precision and run-time overhead.

  - Reduced DDG Method

    As an optimization method for constructing DDG, a reduced DDG has been proposed also in [1]. In this method, the same substructure of DDG has been identified and shared with one structure, so that the overall size of the reduced DDG is bounded not by the program execution length, but by the program size. However, the execution overhead for the construction would be increased due to checking the structure similarity, and thus, the practicability of this method is unknown. Also the size of the reduced DDG is bounded but still large.

  - Dependence-Cache Method

We propose this method in Section 4. This method uses a simple memorization technique to determine each Def-Use relation at execution time. Each data dependence found is added to partial PDG which is statically constructed with nodes and their control dependence relations. This method is easily applicable, and it is effective for non-scalar variables such as array and pointer variables. Only we need is a cache for recording last definition of each variable, and the cache size is bounded by the number of variables used by the program.

As candidates of promising approaches of practical slicing methods using both static and dynamic information, we propose, in this paper, call-mark slicing and dependence-cache slicing.

3 Call-Mark Slice

3.1 Execution Dependence and CED

Here, we introduce an *execution dependence* of two statements s_1 and s_2.

Consider a case where s_1 cannot be executed if s_2 is not executed. We say that s_1 is executionally dependent on s_2.

Finding all of the execution dependences in a program would require dynamic information of the program behavior which is known to be very expensive to compute. Thus, we choose a practical and safe approximation; namely, we use a subset obtained by static analysis of the program(with assumption of program termination).

If both s_1 and s_2 are contained in the same basic block of the control flow graph[3], i.e., there is no outgoing or incoming path on the control flow between the two statements, s_1 is executionally dependent on s_2 and vice versa. Also, if s_2 is contained in a dominant basic block of s_1's block, s_1 is executionally dependent on s_2. The control flow associated with the basic block structure and the domination relation of the basic blocks are easily obtained by static analysis[3].

Now we define a set of caller statements with execution dependence, $CED(s)$, as follow.

$$CED(s) \equiv \{t \mid t \text{ is a function/procedure call}$$

statement and s is executionally
dependent on t}

The execution of s is dominated by the call statements in $CED(s)$. If $CED(s)$ contains at least a non-executed call statement at the end of the execution, we can conclude that s was never executed.

Consider a small portion of a program such that,

```
     ...
s1: callA ;
s2: if a=1 then begin
s3:     b:= c ;
s4:     callB ;
     ...
```

In this program, $s1$ is executionally dependent on $s2$, and vice versa, and also $s3$ and $s4$ are executionally dependent on each other. In addition, $s3$ and $s4$ are executionally dependent on $s1$ and also on $s2$. Thus $CED(s2) = \{s1\}$ and $CED(s3) = \{s1, s4\}$.

3.2 Computation of Call-Mark Slice

A *call-mark slice* is defined as a subset of a static slice of the original program with respect to an execution e of the program and a slice criterion (s_c, v) where s_c is a statement and v is a variable. Each statement s in the call-mark slice is such that all of the call statements in $CED(s)$ are executed at least once in the execution e. This means that statements appearing in the static slice but not in the call-mark slice are not executed in e. The execution of each statement is determined by the record of activation of each call statement in the program.

In the following, a process of computing the call-mark slice is described.

Step 1 Pre-Execution Analysis

Similar to the computing a static slice, we construct the PDG (Program Dependence Graph) by analyzing the data dependences and control dependences among the statements. Also, the execution dependences and $CED(s)$ for each statement s are computed at the same time.

Step 2 Execution-Time Marking

The target program is executed with an input data set. Each time a call statement of a function/procedure is executed, that statement name (i.e., a pointer to the node in PDG) is marked as "executed". We refer to the set of marked call statements as CM.

Step 3 Post-Execution Slice Construction

By performing the algorithm shown in Figure 5, the call-mark slice is collected.

For the program shown in Figure 1, we have a static slice for the slicing criterion of (line 24, d) as shown in Figure 3. Consider an execution of this program with input data ($a = 2, b = 3, c = 0$). In this case, $CM = \{19\}$. For line 19 of this program, $CED(19) = \{19\}$; thus $CED(19) \subseteq CM$ and line 19 is not deleted. For line 21, $CED(21) = \{21\}$; therefore $CED(21) \not\subseteq CM$ and line 21 (and associated line 20) is deleted. Since line 21 is deleted, the statement defining b at line 15 is also deleted. The resulting call-mark slice for that execution and the same criterion, (line 24, d) is as shown in Figure 6.

4 Dependence-Cache Slice

4.1 Overview

As we have discussed in Section 2.2.2, getting precise data dependences of variables with static methods are difficult in general, although the control dependences are fairly easily collected statically. Furthermore, data dependencies of pointer variables and array elements are more difficult to obtain statically than scalar variables, since analyses of those non-scalar variables require complicated prediction of the content values of pointer variables and array indices[13, 15, 18]. If we would fail to predict possible contents of a pointer variable v, we have to use a safe assumption such that a statement s which indirectly refers to a variable u via v depends on all statements which possibly define u directly or indirectly. (Similar cases could happen for array variables.) Thus, precision of resulting slice would be low, i.e., a lot of spurious statements which do not affect the slice criteria are included.

Input

PDG: Program Dependence Graph (statically constructed)

CM: Set of nodes which are call statements executed

(s_c, v): Slice criterion where s_c is a node (a statement) and v is a variable name

Temporary

M, N: Sets of nodes

m, n: nodes

Output

M: Set of nodes of call-mark slice

Algorithm Body

1. $M := s_c$
2. $N := \{n \mid n \xrightarrow{v} s_c\} \cup \{m \mid m \dashrightarrow s_c\}$
3. While $N \neq \phi$ then execute the following steps
   (a) choose a node $n \in N$
   (b) $N := N - n$
   (c) if $CED(n) \not\subseteq CM$ then goto (a)
   (d) $M := M \cup n$
   (e) $N := N \cup \{m \mid m \notin M \wedge (m \xrightarrow{w} n \vee m \dashrightarrow n)\}$
   where w is any variable name.

Figure 5: Algorithm of Post-Execution Collection for Call-Mark Slicing

```
 1  program Square_Cube(input,output);
 2  var a,b,c,d : integer;
 3  function Square(x : integer):integer;
 4  begin
 5     Square := x*x
 6  end;
 7
 8
 9
10
11  begin
12
13    readln(a);
14
15
16
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20
21
22    if (d < 0) then
23       d := -1 * d;
24    writeln(d)
25  end.
```

Figure 6: Call-Mark Slicing Result by d at Line 24 with input $(a = 2, b = 3, c = 0)$

Once we execute the program with an input data, we are able to collect actual dependence relations between statements, although the penalty of collecting precise dependences is fairly large overhead for program execution.

Here, we introduce *Dependence-Cache Slicing* method, for a good compromise between slice precision and execution overhead. The following are major steps of computing dependence-cache slices.

Step 1 Pre-Execution Analysis

We statically construct a part of program dependence graph, named PDG_{DS} for dependence-cache slicing. We prepare at first nodes for each statement or predicate statement, and draw control dependence edges between nodes, as we do for constructing a program dependence graph PDG for static slicing. No data dependence relations are added to the graph.

Step 2 Execution-Time Data Dependence Collection

The target program is executed with an input data. Along the execution, dynamic data dependence relation is collected by using data dependency collection algorithm shown in the next section, and data dependence relation edges are added to the graph. When the program execution terminates, PDG_{DS} has been completed.

Step 3 Post-Execution Slice Construction

The completed PDG_{DS} is traversed backwardly, as we do for static slicing, from a slice criterion (s_c, v) where s_c is a statement and v is a variable. A *dependence-cache slice* is a collection of all reachable nodes by this traversal.

Since data dependences are analyzed dynamically, we would expect smaller slice size, close to that of the dynamic slice.

4.2 Data Dependence Collection Algorithm

Figure 7 shows the data dependence collection algorithm used at Step 2 in Section 4.1. For each variable v used in a program, we prepare a cache, denoted by $C(v)$, which contains the line number of a program statement (i.e., a node in PDG_{DS}). For each point of program execution, $C(v)$ always

Input

PDG_{DS}: Partially constructed Program Dependence Graph for dependence cache slicing

P: Target Program

I: Inputs for P

Temporary

Data Dependence Caches $C(v)$ for each variable v in P

Output

OUT: Output of execution of P for I

PDG_{DS}: Completely constructed Program Dependence Graph for dependence cache slicing

Algorithm Body

1. For each variable v in P, $C(v) := \perp$
   { Initialize with not assigned marks. Note that if we use a dynamically allocated variable, we prepare also a cache initialized with the not-assigned mark }
2. Repeat following untile execution of P terminates
   { Execute P with I from the beginning to the termination, statement by statement }
   (a) Execute a next single statement s of P associated with I
   (b) For each variable u used (referred) at s, if $C(u) \neq \perp$, then add a data dependence edge $C(u) \xrightarrow{u} s$ to PDG_{DS} unless the edge exists
   (c) For each variable w defined at s, $C(w) := s$

Figure 7: Data Dependence Collection Algorithm

keeps the line number of a statement which most recently defined v. When v is used (referred) by execution of a statement s, a data dependence edge is drawn from a node for $C(v)$ to a node for s, if it does not exist yet. When v is defined at s, $C(v)$ is updated with the line number of s.

We do this for all types of variables, including arrays, dynamically allocated variables, pointer variables, and so on.

We prepare caches for each element of an array. For example, for array A of ten elements $A[1]$, $A[2]$, ..., $A[10]$, we have caches $C(A[1])$, $C(A[2])$, ..., $C(A[10])$. For dynamic allocated variables, we prepare caches for each variable also dynamically. When a pointer variable p is used at statement s, we have to know that not only p itself is used, but an indirectly accessed variable $p\uparrow$ is also treated as a variable to be used. Thus, we have to include the edges of both direct and indirect references ; i.e., $C(p) \xrightarrow{p} s$ and $C(p\uparrow) \xrightarrow{p\uparrow} s$. Also, in the case of indirect assignment with a pointer variable q such as $q\uparrow := \ldots$ at statement t, we have to know that cache $C(q\uparrow)$ is updated with t, and also that q is used at t.

This algorithm requires cache space proportional to variable usage of the program execution, and runtime overhead corresponding to variable access by the program execution.

The result of dependence-cache slicing for program shown in Figure 1 is the same as the dynamic slice shown in Figure 4 for the same input data $(a = 2, b = 3, c = 0)$.

5 Experiments

5.1 Overview of Osaka Slicing System

In order to validate various slicing algorithms, we have developed a software development and debugging environment called *Osaka Slicing System*[24]. Figure 8 shows the architecture, and the target language is Pascal.

The system contains slicers, as well as a program executor and debugger. The slicing algorithms used by the system are interprocedural flow-sensitive context-insensitive static slicing and fully traced dynamic slicing. These algorithms are sometimes replaced with call-mark slicing and dependence-cache slicing for experimental data collection.

In the system, the source program in Pascal is parsed into an abstract

syntax tree that is stored in the system. The user can view and modify the source program interactively using a visual editor.

Figure 9 shows the user interface of the system. The left window displays the target source program. The statement of the slice criteria is shaded darkly, and the statements in the slice result are shaded lightly. We can also edit the source program on this window. The right upper window gives system status such as loaded file name, program size, slice size and so on. The right lower window works for the standard input and output during the execution of the target program.

The source program is analyzed and transformed into a PDG by a user request. A static slice may be computed from the PDG by specifying a slice criterion.

Both the whole source program and a computed static slice can be executed by the executor. The debugger associated with the executor contains features of ordinary runtime debuggers such as tracing, setting breakpoints, viewing and modifying variable values, and so on. Various levels of the dynamic information are recorded during execution based on the selection of slicing algorithms; dynamic slice, call-mark slice, or dependence-cache slice can be computed using this information. The total size of the system is about 19,000 lines of C code.

To handle inter-procedural dependences including recursive functions/ procedures, we have introduced auxiliary types of nodes in PDG for passing parameters and global variables. Using these nodes, the data dependences are examined inter-procedurally. In the case of self or mutual recursive structures of function/ procedure calls, the dependences become cyclic. This cyclic dependence is efficiently solved by analyzing the structure of the dependences, and a suitable solution is found[25].

The implementation of the call-mark slicing is based on the method presented in previous section. For Step 2, we need only one bit of information for each function/ procedure call statement in the program. This bit is not necessarily marked at the caller context, but rather at the callee context. At the entry of each function/ procedure, the pointer to the return context is collected as CM. By doing so, we do not need to find out all function/ procedure calls in the program, but we simply modify the entry part of each function/procedure slightly.

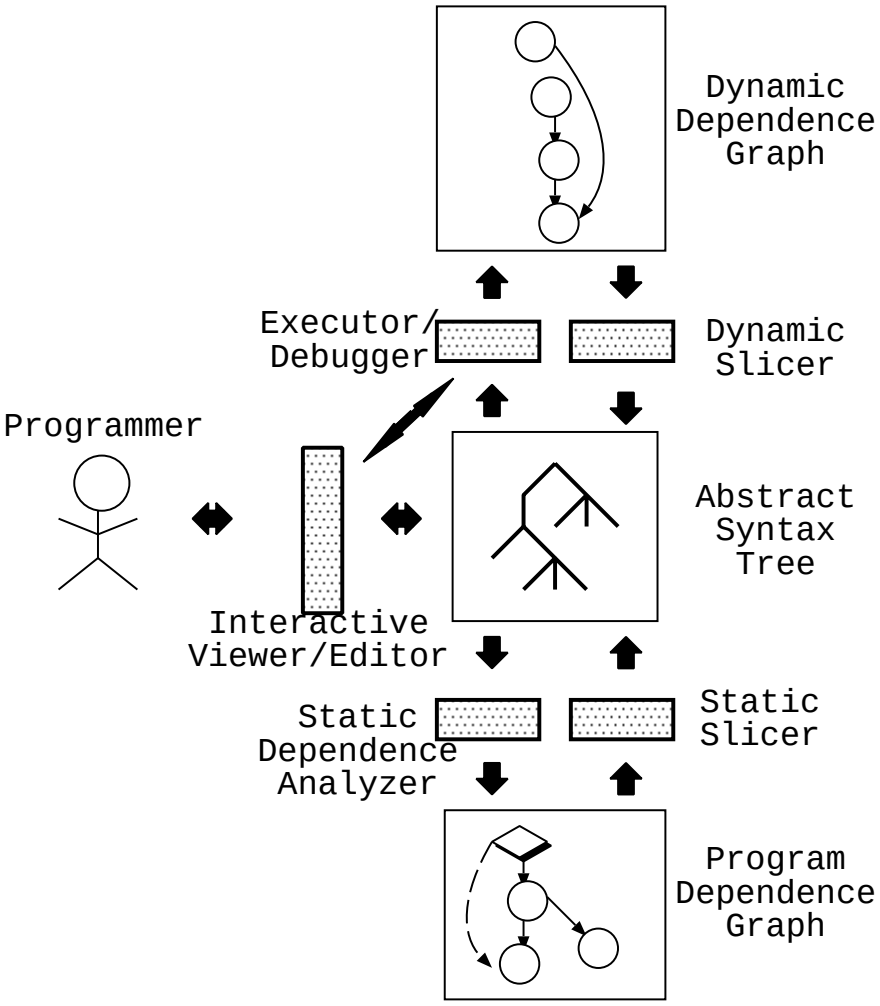

Figure 8: Architecture of Osaka Slicing System

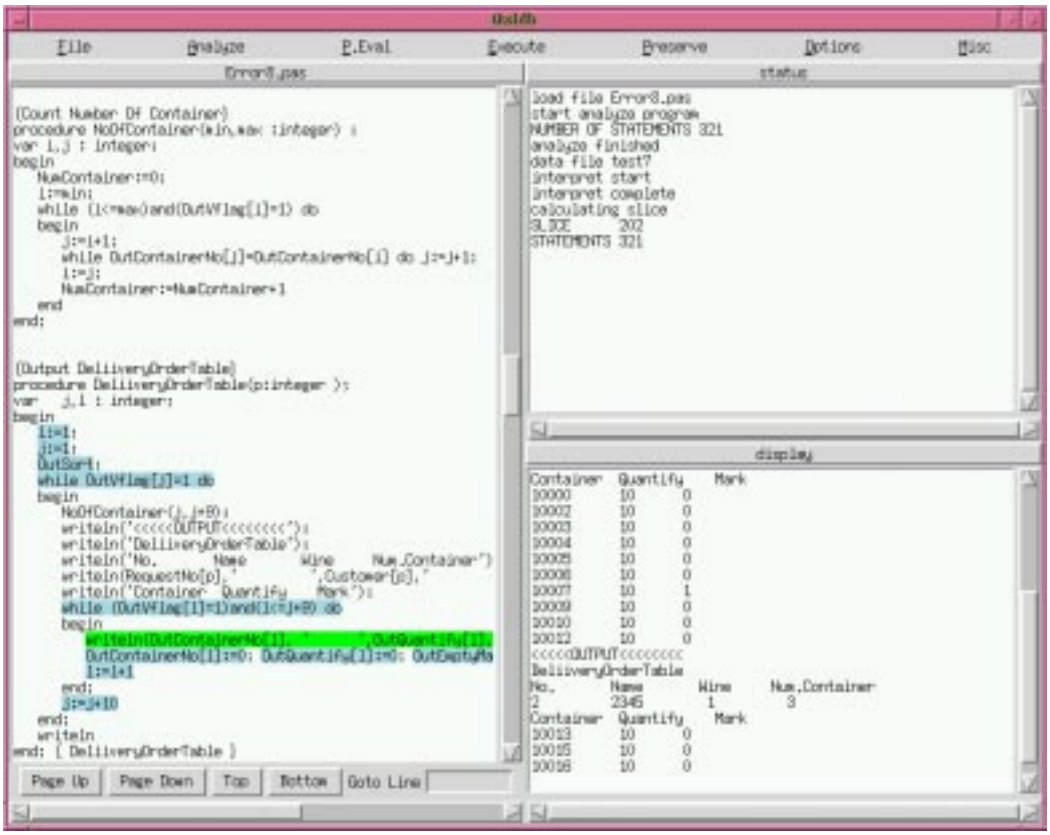

Figure 9: Snapshot of Osaka Slicing System

5.2 Execution of Sample Programs

Using this experimental system, we have executed various programs and obtained several metrics values. Program $P1$ is a calender calculation program. $P2$ is an inventory management program for a whole seller. $P3$ is also an extended version of the inventory management program of $P2$.

Table 1 shows the slice sizes of three sample programs. These values can vary with different slice criteria and input data. Here, we show average values for several criteria and inputs for a typical debugging situation. (The criteria are mostly program output variables, and the output statements are placed almost at the end part of program execution.)

Table 2 shows the time needed for the analysis before the execution. In the case of static slicing, the value is the time to construct a PDG. The time for computing both the PDG and the CED is counted for in the call-mark slicing. For the dependence-cache slicing, it is time for constructing an initial PDG_{DS}. In the case of dynamic slicing, the analysis is not necessary.

In Table 3, the execution time is shown. In the case of static slicing, the original program is executed without any extra runtime overhead; thus this value means the execution time of the original program. The execution for the dynamic slicing is performed in association with the construction of the dynamic dependence graph. Therefore, the execution time contains the time for this construction. The time for the dependence-cache slicing includes the time for caching dependencies and constructing PDG_{DS}. In the case of call-mark slicing, the time to mark callers is included.

Table 4 shows the time needed for collecting statements to be included in the resulting slices. In the case of static slicing, this could be done before execution. For dynamic slicing, the time for traversing the dynamic dependences is counted. For call-mark slicing and dependence-cache slicing, these are the time for Step 3, respectively.

We discuss these tables in detail in the next section.

6 Discussion

6.1 Interpretation of Experiment Data

- Slice Size

Table 1: Size of Slice (lines)

program	static	call-mark	dependence-cache	dynamic
P1(85 lines)	21	17	15	5
P2(387 lines)	182	162	16	5
P3(871 lines)	187	166	61	8

Table 2: Pre-execution Analysis Time (ms)

program	static	call-mark	dependence-cache	dynamic
P1	11	14	5	N/A
P2	213	215	19	N/A
P3	710	698	48	N/A

(Celeron-450MHz CPU with 128MB Memory)

Table 1 shows the sizes of slices. Those of the call-mark slicing are 80–89% of static slicing. Also, those of dependence-cache slicing are 9–71% of static slicing.

The slice sizes of the call-mark and dependence-cache slices are between the static and dynamic slices. It is always smaller and better than the static slice and bigger and worse than the dynamic slice. Also, we can say that the dependence-cache slices are much better (smaller) than the call-mark slices. This is because dependence-cache slicing reflects data dependences on a particular execution path, while call-mark slicing only cuts out unexecuted parts of the statically detected data dependences based on a calling sequence. Reduction of the slice size with the dependence-cache slicing is large for programs $P2$ and $P3$, compared with $P1$. This is because $P1$ uses only scalar variables and $P2$ and $P3$ employ array variables whose element-wise data dependencies are only analyzed by dependence-cache slicing or dynamic slicing.

- Pre-Execution Analysis:

  As shown in Table 2, call-mark slicing needs a little extra time compared to the static slicing. This is natural since we have to construct a

Table 3: Execution Time (ms)

program	static	call-mark	dependence-cache	dynamic
P1	47	47	51	174
P2	43	43	45	4,540
P3	4,700	4,731	4,834	206,464

(Celeron-450MHz CPU with 128MB Memory)

Table 4: Slice Construction Time (ms)

program	static	call-mark	dependence-cache	dynamic
P1	0.4	0.6	0.3	76.0
P2	1.9	1.8	0.7	101.0
P3	3.0	3.0	1.2	24,969.3

(Celeron-450MHz CPU with 128MB Memory)

PDG as in static slicing, and additionally we have to analyze the execution dependences. For dependence-cache slicing, we need a lightweight pre-execution analysis only for control dependences, which is fairly smaller overhead than the data dependence analysis for static and call-mark slicing.

- Execution Time

  The execution time shown in Table 3 indicates that the overhead of the dynamic slicing is extremely big. If the program execution becomes longer by say, repeated execution of loops, this overhead would cause serious decline of performance so that the programmer can hardly use this facility. On the other hand, the call-mark slicing can be executed with very little overhead increase compared to the execution of the static slicing (i.e., the execution of the original source program). It shows that the marking of the caller names during execution is a lightweight task, requiring little execution time.

  For dependence-cache slicing, it requires extra execution time to call-mark slicing; however, the values are much smaller than those of dy-

namic slicing. The execution time presented here is based on our interpretive system; therefore, the run-time overhead for these slicing might be masked by the overhead of the interpretive execution. This issue will be discussed in Section 6.3.

- Slice Construction Time

  As shown in Table 4, dynamic slicing requires a long time to collect the slice result. The time for collecting a call-mark slice is almost the same as the time to collect a static slice. For a dependence-cache slice, it is better than the static one. This is because dependence-cache slicing construct PDG_{DS} smaller than PDG, so that the searching space within the PDG_{DS} is smaller than that for static slicing. For dependence-cache slicing, it is evident that traversing a large dynamic dependence graph (DDG) is very costly.

6.2 Application Domain and Limitation of Call-Mark Slice

The pre-execution analysis, execution, and slice construction times are almost the same as those for static slicing, and the sizes of slices are about 12-20% smaller than static slices.

The reduction rate might not be so significant. However, call-mark slicing method can be easily implemented if we have already a static slicing tool and a program execution environment in which we can steal return addresses in the activation records of a system stack. Also, there is very little overhead increase. Thus we consider that this method would be a very handy improvement method of static slicing.

One of main targets of call-mark slicing method is a debugging environment. Dynamic information is considered to be essential for efficient fault detection. A call-mark slice can be directly associated with a specific test data which exposes faults in the source program, although static slice would generally include various portions which do not relate to the execution with the test data.

Programs which consist of independent function/procedure components may also be efficiently debugged with our approach. In such cases, there would be many function/procedure invocation statements, and also many

function/procedure definitions. Activation of selected functions/procedures using a specific input data will clearly reduce the slice size.

On the other hand, if the target programs contain a small number of function/procedure invocation statements or the function/procedures are tightly interleaved, the effect of our approach will be limited.

6.3 Application Domain and Limitation of Dependence-Cache Slicing

As we can see in Table 1, the size of dependence-cache slicing is much smaller than that of static and call-mark slicing, although it is bigger than that of dynamic slicing. It is worth noticing that programs using arrays, such as $P2$ and $P3$, are well analyzed with dependence-cache slicing, producing smaller slices. Thus, we can say that this approach is very effective to compute slices for programs using arrays and pointers, whose statical analysis would be complicated and impractical.

To implement dependence-cache slicing under a practical compiler-based environment, we have to consider the following.

- Control dependences has to be analyzed statically, and PDG with only control dependence edges is to be built. This might be performed by a compiler and its optimizer.

- Execution environments has to be changed to include the dependence caches for each variable. The size of the caches is proportional to the number of variables used in the program. If the program uses dynamic variables, we have to prepare their caches dynamically also. Run-time memory requirement will be increased, say, doubled by this. However, the cost of memory would not be a critical issue at program debugging phase.

- Compiler-generated codes must contain instructions for updating the dependence caches and adding dependence edges to PDG.

The experiment shown in Section 5 has been made under our interpretive execution system. We would think that the characteristics of analyses and execution times for various slicing will hold for compiler-based execution environment, although the run-time overhead needed for semi-dynamic methods would become clearer.

For example, we have written a merge sort program in C. This program has been modified to collect data dependences with the dependence caches during execution of the merge sort. The execution speed of its compiled code was 8.6 times slower than the original code. This would indicate that the run-time overhead of the dependence-cache slicing is large under a compiler-based environment*. However, this speed down can be minimized if we collect the data dependence information only for array and pointer variables. Data dependences through scalar variables are fairly easily determined statically. Thus, statically difficult analysis is only performed at execution time. Based on this idea, we have modified the C merge sort program again, so that the data dependences only for array elements were collected dynamically. The execution-time ratio between the original program and the partial dependence-cache program became 1 to 3.4, and we would think that this is practically acceptable.

As we can see with Table 1, the dependence slices are always larger than the dynamic slices. This difference is based on the rationale such that the dependence slicing does not distinguish repeated occurrences of a single statement and that it only holds the latest Def-Use relations in caches. Consider the following simple example.

```
1: a[0]:=10 ;
2: a[1]:=20 ;
3: for i:=0 to 1
4:     b[i]:= a[i] ;
5: writeln(b[0]) ;
```

Execution of this program will create a DDG (Dynamic Dependence Graph) as shown in Figure 10. Nodes 3 and 4 appear twice and lower nodes mean the second occurrences of statement 3 and 4. If we compute a slice with a criterion (statement 5, $b[0]$), the result of the dynamic slicing is a set of statements 1, 3, 4 and 5.

On the other hand, the dependence caches contain the last definition statements as shown in Table 5. PDG_{DS} created by this execution is shown in Figure 11, and resulting slice with the same criterion (statement 5, $b[0]$) is a set of statements 1, 2, 3, 4, and 5.

*Also, this result suggests that the execution overhead for dynamic slicing would be unacceptably huge.

Table 5: Change of Cache Contents for Simple Sample Program

statement executed	Cache Contents				
	$a[0]$	$a[1]$	$b[0]$	$b[1]$	i
1	1	-	-	-	-
2	1	2	-	-	-
3	1	2	-	-	3
4	<u>1</u>	2	4	-	<u>3</u>
3	1	2	4	-	<u>3</u>
4	1	<u>2</u>	4	4	<u>3</u>
5	1	2	<u>4</u>	4	3

Underlines show that the variable is used and the cache is referred to

Dependence-cache slicing includes statement 2 as its result. This is because the dependence-cache slicing method cannot distinguish the first and second occurrences of statement 4 execution, and dependences both from statement 1 and 2 are targeted to a single node for statement 4.

This limitation increases the slice size for dependence-cache slicing, compared to dynamic slicing; however, it is fairly smaller than the static slice, and we think that this approach is a practical and promising method compromising between effectiveness and overhead.

6.4 Relation to Other Methods

Others have worked on combining static and dynamic information for slicing.

Hybrid slicing[12] targets a similar goal as call-mark slicing. It reduces the static slice by using two types of dynamic information: breakpoint information and call history information. The former is supplied by the programmer and that information is used to infer the executed control flow. The latter is used to compute portions of dynamic slices for the periods between every function/procedure call and return. The result is closer to the dynamic slice than our approach since it gathers more dynamic information. The weakness of the hybrid slicing would be that we have to specify appropriate breakpoints to get a better slice. On the other hand, our approach performs everything automatically except for giving the input data and slice criterion. Also, the

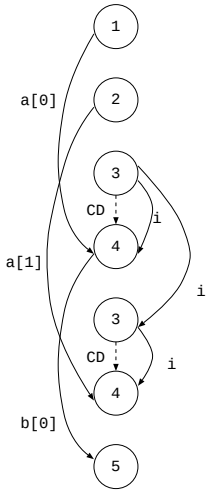

Figure 10: Dynamic Dependence Graph for Simple Sample Program

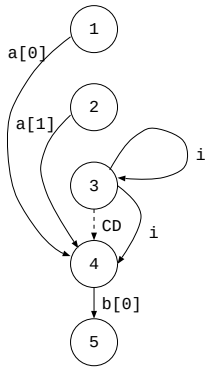

Figure 11: PDG_{DS} for Simple Sample Program

hybrid slicing requires a fairly large amount of memory space for recording the call history. The space required is roughly proportional to the program execution length. Our call-mark slicing, however, needs only the memory for the call marks, which is of a pre-determined size and is roughly proportional to the program text size. The difference is significant if the program execution becomes long. The execution method of the hybrid slicing mainly contributes to determine an execution path of the program. However, this execution path information does not always help to determine data dependences of pointer and array variables, as our dependence-cache slicing.

Our approach, call-mark slicing, uses information of whether or not each function/procedure call statement in the program is executed. The precision of slices can be improved if we take such information of all statements in the program. This approach, mentioned in [1] as type 1 method, can be implemented using a similar method to computing profiling and program coverage information. For each statement, we employ one bit flag of whether it is executed or not. The mechanism would be simple; however, it requires more run-time overhead and significant modification of the executable program. The call-mark slicing information can be obtained by minor modification of the function/procedure entry routine to collect caller statements.

We could also take a simpler context-insensitive approach than the call-mark slice, where each context of function/ procedure activation is recorded. We could only gather information about whether or not each function/procedure is activated without recording which call statement actually activated it. This approach reduces run-time overhead for collecting caller statements; however it would increase the resulting slice size.

There are little researches which collect dynamical data dependencies efficiently. In [1], an optimization method of constructing a dynamic dependence graph for dynamic slicing, where same substructures of the graph are reduced into a single one. This approach keeps the precision of dynamic slices and reduces the space for the graph; however, its execution overhead would not be reduced or might be increased by checking the duplicates.

There are researches in which pointer and array variables are statically analyzed[13, 15]. Many of these try to statically determine possible aliases of pointer variables and array elements, and they still remain uncertain cases[29]. Since our dependence-cache slicing is dynamic one and there is no complicated static analyses. It is easily applicable to any type of variables, and we can get reasonable slice precision with affordable execution

overhead.

In [8], a method to extract various slice algorithms from semantic specifications is presented. They propose a constrained slice, which is a generalization of static and dynamic slices, and which takes a subset of the inputs of the program as symbolic program execution. Using this input constraint, the program is rewritten and dependences are computed. Their approach contains very important notions of generalization of static and dynamic slicing, and also it covers the partial evaluation and program simplification methods. However, it is not known whether such a generalized approach may be implemented efficiently and whether it is useful practically.

In terms of building analysis systems, several interesting approaches have been proposed. Various ways of analyzing large programs and extracting abstracted information of the target software have been studied[4, 11, 19]. A generalized environment for developing analysis algorithm is proposed in [26]. It uses denotational frameworks to specify analysis algorithms; however, practicability of the generated algorithms for analysis tools is not known. In [20], a more practical environment for understanding Cobol programs is presented, where various slicing and program localization features are unified. Our aim is to construct an efficient and effective environment for structured languages with functions and procedures calls.

7 Conclusions

Localizing a programmer's attention to a small portion of software is very important for improving the efficiency of program debugging and maintenance. Traditional program slicing methods do not provide adequate trade-offs of effectiveness and efficiency.

We have proposed two lightweight semi-dynamic slicing methods, call-mark slicing and dependence-cache slicing. These methods use lightweight run-time execution, and have a similar or smaller analysis overhead with respect to static analysis as static slicing. The resulting slices are smaller than corresponding static ones, but larger than corresponding dynamic ones. We have implemented these slice algorithms on our experimental interpreter system. Also we have executed various sample programs, and confirmed our approach.

We are planning to design a debugging environment based on compiler-

based system, rather than the current interpreter-based system. The system will be able to compute dependence-cache slices, associated with various debugging features. This compiler generates a run-time environment with dependence caches, and the generated codes automatically collect dynamic data dependences. This information is displayed as a slice or other debugging aids when requested by users, and it gives more fruitful suggesting for fault localization.

Acknowledgments

The authors are very grateful to Akira Nishimatsu, Minoru Jihira, and Shinji Kusumoto of Osaka University who have contributed to designing and implementing the call-mark slicing system.

References

[1] Agrawal, H. and Horgan, J. : "Dynamic Program Slicing", *SIGPLAN Notices, Vol.25, No.6*, pp. 246–256, 1990.

[2] Agrawal, H., Demillo, R. A., and Spafford, E. H. : "Debugging with Dynamic Slicing and Backtracking", *Software Practice and Experience*, Vol. 23, No. 6, pp. 589–616 (1993).

[3] Aho, A. V., Sethi, R., and Ullman, J. D.: "Compilers: Principles, Techniques, and Tools", *Addison Wesley*, Massachusetts, 1986.

[4] Atkinson, D.C. and Griswold, W.G. : "The Design of Whole-Program Analysis Tools", *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, pp. 16–27, 1996.

[5] Bates, S. and Horwitz, S. : "Incremental Program Testing Using Program Dependence Graphs", *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

[6] Beck, J. and Eichmann, D. : "Program and Interface Slicing for Reverse Engineering", *Proceedings of the 15th International Conference on Software Engineering*, pp. 509–518, 1993.

[7] Binkley, D.W. and Gallagher, K.B. : "Program Slicing", *Advances in Computers, Volume 43*, Marvin Zelkowitz, Editor, Academic Press San Diego, CA, 1996.

[8] Field, J., and Ramalingam, G.: "Parametric Program Slicing", *Proc. of 22nd ACM Symposium on Principles of Programming Languages*, pp. 379–392, San Francisco, USA, January (1995).

[9] Fiutem, R. , Tonella, P. , Antoniol, G. and Merlo, E.: "Variable Precision Reaching Definitions Analysis for Software Maintenance", In *Proc. of the Euromicro Working Conf. on Soft. Maintenance and Reengineering*, pp. 60–67(1997).

[10] Gallagher, K.B. and Lyle, J.R. : "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering, 17(8)*, pp. 751–761, 1991.

[11] Griswold, W. G. and Notkin, D.: "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool", *IEEE Transactions on Software Engineering*, 21(4):275–287(1995).

[12] Gupta, R., Soffa, M.L., and Howard, J. : "Hybrid Slicing: Integrating Dynamic Information with Static Analysis", *ACM Transaction on Software Engineering and Methodology, Vol. 6, No. 4*, pp. 370–397, 1997.

[13] Hind, M., Burke, M., Carini, P., and Choi, J.:"Interprocedural Pointer Alias Analysis", ACM Trans. on Programming Languages and Systems, Vol.21, No. 4, pp. 848–894 (1999).

[14] Horwitz, S. and Reps, T.:"The Use of Program Dependence Graphs in Software Engineering", Proceedings of the 14th International Conference on Software Engineering, pp. 392–411(1992).

[15] Horwitz, S. Pfeiffer, P., and Reps, T.:"Dependence Analysis for Pointer variables", Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation, pp.28–40, SIGPLAN Notices Vol. 24, No. 6 (1989).

[16] Korel, B., and Laski, J. : "Dynamic Program Slicing", *Information Processing Letters*, Vol.29, No,10, pp. 155–163 (1988).

[17] Korel, B., and Laski, J. : "Dynamic Slicing of Computer Programs", *Journal of Systems Software*, Vol.13, pp. 187–195 (1990).

[18] Liang, D., and Harrold, M. J., : "Efficient Points-To Analysis for Whole-Program Analysis", *Proc. of 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp.199–215, Toulouse, France (1999).

[19] Murphy, G. C., and Notkin, D.: "Lightweight Lexical Source Model Extraction", *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, pp. 262–292, July (1996).

[20] Ning, J. Q., Engberts, A., Kozaczynski, W. V. : "Automated Support for Legacy Code Understanding", *Communications of the ACM*, Vol. 37, No. 5, pp.50–57, May (1994).

[21] Nishie, K., Kamiya, T., Kusumoto, S. and Inoue, K. : "Experimental Evaluation of Usefulness of Debugging Support System Based on Program Slicing", *Proceedings of Software Symposium '97*, pp. 142–147, Japan, 1997(in Japanese).

[22] Nishimatsu, A., Kusumoto, S., Inoue, K. : "An Experimental Evaluation of Program Slicing on Fault Localization Process", *Technical Report of IEICE Japan*, SS98–3, pp. 17–24, (1998)(in Japanese).

[23] Nishimatsu, A., Jihira, M., Kusumoto, S. and Inoue, K. : "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice", *Proceedings of The 21st International Conference on Software Engineering*, pp.422-431, Los Angeles, CA, USA, 1999.

[24] Sato, S., Iida, H., and Inoue, K. : "Software Debug Supporting Tool Based on Program Dependence Analysis", *Transaction on IPSJ*, Vol. 37, No. 4, pp. 536–545 (1996) (in Japanese).

[25] Ueda, R., Inoue, K., and Iida, H. : "A Practical Slice Algorithm for Recursive Programs", *Proceedings of the International Symposium on Software Engineering for the Next Generation*, pp. 96–106, Nagoya, Japan, February (1996).

[26] Vengatesh, G. A., and Fischer, C. N. : "SPARE: A Development Environment for Program Analysis Algorithms", *IEEE Trans. on Software Engineering*, Vol. 18, No. 4, pp.304–315, April (1992).

[27] Weiser, M.: "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449 (1981).

[28] Weiser, M.:"Program Slicing", *IEEE Transactions on Software Engineering*, 10, pp.352-357, July (1984).

[29] Ramalingam, G.: The Undecidability of Aliasing, *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, pp. 1467–1471 (1994).