Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice

Akira Nishimatsu† **Minoru Jihira**‡ **Shinji Kusumoto**† **Katsuro Inoue**†

† Graduate School of Engineering Science,
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
+81 6 6850 6571
{a-nisimt, kusumoto, inoue}@ics.es.osaka-u.ac.jp

‡ Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma,
Nara 630-0101, Japan
+81 743 72 5236
minoru-j@itc.aist-nara.ac.jp

ABSTRACT

When we debug and maintain large software, it is very important to localize the scope of our concern to small program portions. Program slicing is one of promising techniques for identifying portions of interest. There are many research results on the program slicing method. A static slice, which is a collection of program statements possibly affecting a particular variable's value, limits the scope, but the resulting collections are often still large. A dynamic slice, which is a collection of executed program statements affecting a particular variable's value, generally reduces the scope considerably, but its computation is expensive since the execution trace of the program must be recorded.

In this paper, we propose a new slicing technique named call-mark slicing that combines static analysis of a program's structure with lightweight dynamic analysis. The data dependences and control dependences among the program statements are statically analyzed beforehand, and procedure/function invocations (calls) are recorded (marked) during execution. From this information, the dynamic dependences of the variables are explored.

This call-mark slicing mechanism has been implemented, and the effectiveness of the method has been investigated.

Keywords
Static Slicing, Dynamic Slice, Dependence Analysis, Execution Overhead

1 INTRODUCTION

Debugging and maintaining large software is one of a central theme of software engineering research and practice.

Various ways of analyzing large programs and extracting abstracted information of the target software have been studied[5, 11].

One approach to ease the difficulty of handling large software is to localize a developer's attention to specific parts of the program that are directly and indirectly related to the developer's concerns.

Program slicing is one promising technique to realize this program localization[8, 17].

We have investigated the effectiveness of program slicing for program debugging and maintenance processes using a controlled method[13]. 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.

There are many research results on the program slicing method[7, 8]. Static slicing was first proposed by Weiser[17]. 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.

Dynamic slicing was proposed by Agrawal et. al.[1, 2, 9, 10]. 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 during execution.

We have explored various slicing methods under the following policy.

- The static analysis information and lightweight dynamic information are combined.
- The reduction rate of slicing is greater than the static slice, and is hopefully close to the dynamic slice.
- The execution overhead is minimized as much as possible.

In this paper, we propose a new slicing technique named *call-mark slicing.* Data dependences and control dependences among the program statements are statically analyzed beforehand, and procedure/function invocations (calls) are recorded (marked) during the program's execution. From this information, the dynamic dependence of the variables are explored.

This call-mark slice mechanism has been implemented as part of our Osaka Slicing System. Using this system, we have executed various sample programs for the evaluation. The result shows that the call-mark slice limits the scope better than the static slice. Also, there is little burden of run-time information collection.

In Section 2, we will overview the static and dynamic slicing methods. We propose the call-mark slicing method in Section 3. Section 4 describes the implementation of the call-mark slicing method as part of our slicing system, and also shows the results of using our method on sample programs. In Section 5, we discuss our approach compared to other method. We conclude with some remarks in Section 6.

2 STATIC AND DYNAMIC SLICING

Static Slicing

Consider statements s_1 and s_2 in a source program p. When all of the following conditions are satisfied, *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.

```
 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: Pascal Source Program

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

When the following conditions are all satisfied, *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 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 as discussed in Section 5.

A Static Slice with respect to a variable v on a statement s (this pair (v, s) is called the *slice criterion*) in a program is a collection of statements corresponding to the nodes which possibly reach v on s through the edges in the PDG, using CD relations and DD relations in the program. 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.

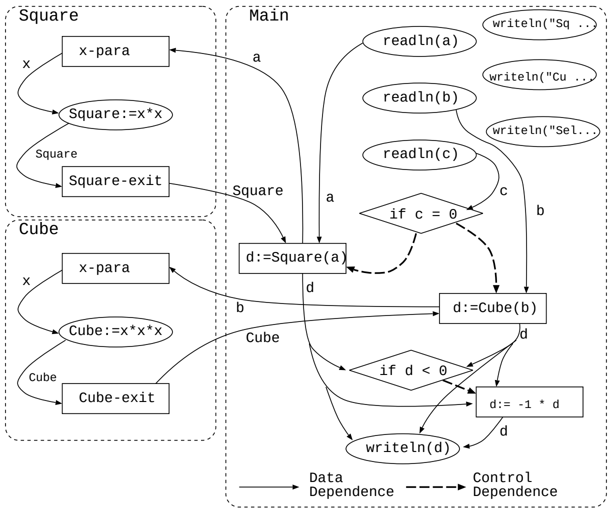


Figure 2: Program Dependence Graph (PDG)

```
 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

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 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/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.

Some Issues on Static and Dynamic Slicing
In static slicing, the analysis is performed without executing the program. The control dependences are fairly easily obtained by parsing the source program. The data dependences are computed by solving the data flow equations[3]. We devised an optimized algorithm which analyzes dependences among the equations as mentioned in Section 4. The time complexity of this algorithm is generally square to the source program size, where one of the factors grows very slowly to the program size[15]. Actually, we have constructed a slicing system that performs the static slicing in a very short amount of time[14] (this system has been extended to our Osaka Slicing System discussed in Section 4).

The result of static slicing is a subset of the original program. By supplying appropriate declarations, the obtained result is an executable program and it computes the value of the slice criterion correctly as the original program does.

From the perspective of reducing the scope of concern for the programmer, the static slicing method may not be a helpful approach. This is because the slice result is often a fairly large part of the original source program in many cases. This is generally due to large control and data dependences in a PDG, which cover all possible program execution paths (and infeasible paths also). This is a limitation of static analysis.

On the other hand, the dynamic slice is generally smaller than the static one, since the dynamic slice is computed based on an execution trace for a specific input data. Unrelated statements for that execution are deleted from the slice result.

Also, computing a dynamic slice seems to be a well-suited process for debugging, since the source of many faults could be more easily found by executing programs with suitable test data and examining the faulty program outputs. These executions may be directly used as the traces for dynamic slicing where the names of the variables with faulty output are used as the slice criteria.

```
 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 $(a = 2, b = 3, c = 0)$

For computing dynamic slice, pre-execution analysis is not needed. However, during the execution, we have to keep the dependences of data and control in the memory space or other area. This is a very large overhead in memory space and computation time. Also, since the stored dependences are typically numerous, the collection and reconstruction of the slice after the execution typically takes a long time.

3 CALL-MARK SLICING

The weakness of the static and dynamic slices discussed above is resolved by combining static and dynamic information. We propose here a new method named *call-mark slicing*.

The overhead of the dynamic slice is due to the following two reasons.

- Recording the execution trace and constructing the dynamic dependences of the statements
- Traversing the dynamic dependences and collecting the slice result

For the first issue, we abandon collecting a precise execution trace, instead gathering partial execution information. In call-mark slicing, we only record whether or not each function/procedure call statement in the program is executed.

The second issue is based on the fact that the lengths of execution traces are generally far longer than the source program lengths. The data dependences and control dependences in the execution trace may easily become long chains. These long chains tend to contain repeated dependence structures that are caused by iterations in the program. Thus, the traversal of dependences from a slice criteria can take a fairly long time[1] . To resolve this, we use static dependences, instead of the dynamic ones, for computing a slice; this has the advantage that the static dependences are typically less numerous than the dynamics dependences.

Execution Dependence and CED

Here, we introduce *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.

$$CED(s) \equiv \{t \mid t \text{ is a function/procedure call statement and } s \text{ is executionally dependent on } t\}$$

The execution of s is dominated by the call statements in $CED(s)$. If we know that any call statement in $CED(s)$ is not performed during an execution, we can conclude that s is never executed.

Consider a small portion of a program such that,

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

[1] However, the result of the dynamic slice would be smaller than the static one in general, since the repeated structures are mapped to single statements in the result.

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 $CED(s3) = \{s1, s4\}$.

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 Collection

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.

```
Inputs

    PDG: Program Dependence Graph
    $CM$: Set of nodes which are executed call
        statements
    ($s_c$, $v$): 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. $M \leftarrow s_c$
    2. $N \leftarrow \{n \mid n \xrightarrow{v} s_c\} \cup \{m \mid m \dashrightarrow s_c\}$
    3. While $N \neq \phi$ then execute the following
       steps
        (a) choose a node $n \in N$
        (b) $N \leftarrow N - n$
        (c) if $CED(n) \not\subseteq CM$ then goto (a)
        (d) $M \leftarrow M \cup n$
        (e) $N \leftarrow N \cup$
            $\{m \mid m \notin M \wedge (m \xrightarrow{w} n \vee m \dashrightarrow n)\}$
            where $w$ is any variable name.
```

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

4 IMPLEMENTATION OF CALL-MARK SLICING

Overview of Slice System

In order to validate the call-mark slicing approach, we have implemented the algorithm within our Osaka Slicing System[14].

The target language is a subset of Pascal, which is used in an undergraduate course of our department. It contains essential control structures and recursive function/procedure invocations, allowing most introductory programming.

Figure 7 shows the architecture. It contains static and dynamic slicers, as well as an executor and debugger. The source program in Pascal is parsed into an abstracted source program that is stored in the system. The user can view and modify the source program interactively using a visual editor.

```
 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 $(a = 2, b = 3, c = 0)$

Figure 8 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. The dynamic variable dependences are recorded during execution; a dynamic slice can be computed using this information.

The total size of the system is about 19,000 lines of C code including the call-mark slicing part.

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.

To handle inter-procedural dependences including recursive functions/procedures, we have introduced auxiliary types of nodes in a PDG. Table 1 shows these nodes, which do not directly correspond to source program statements as other nodes. Using these nodes, the data dependences are examined inter-procedurally. In the case of self/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[15].

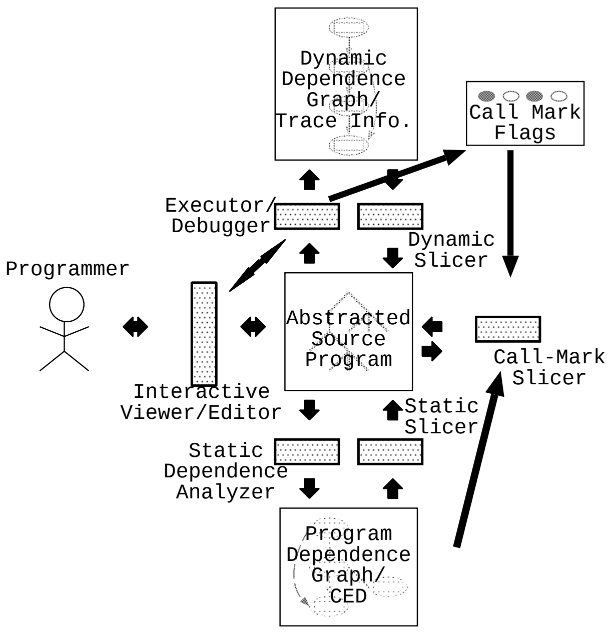
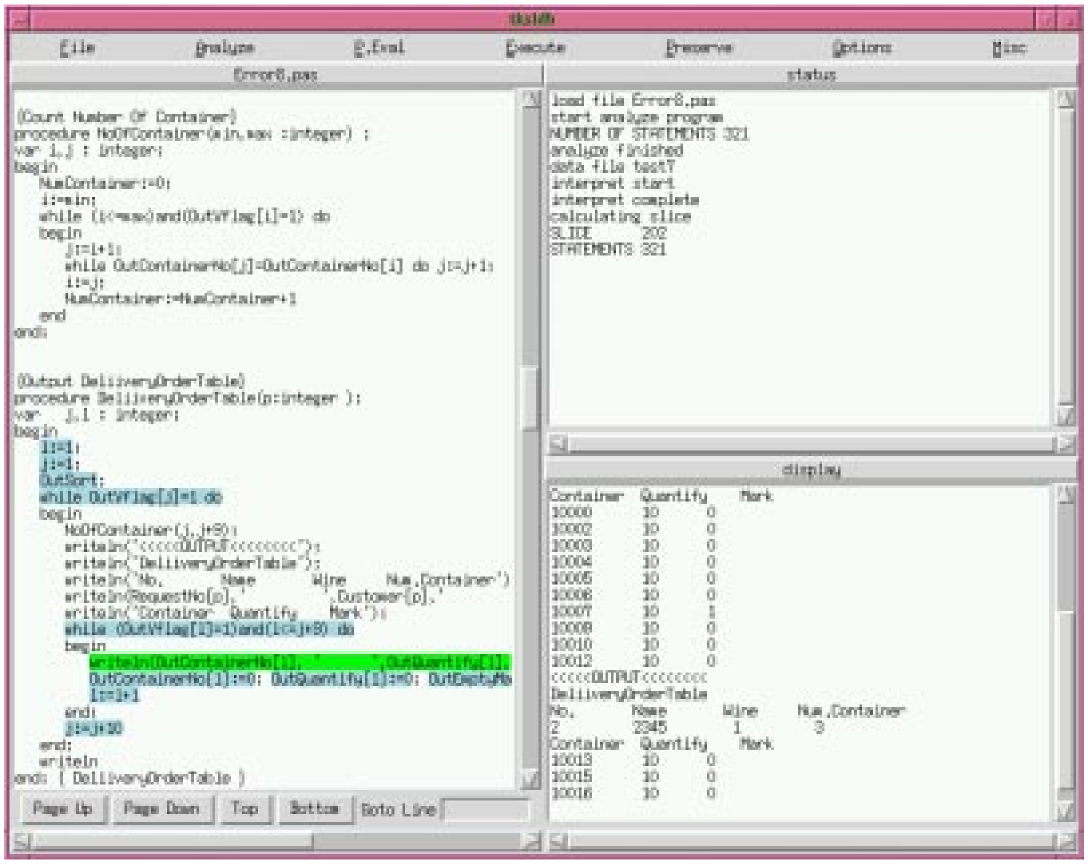

Dynamic Dependence Graph/ Trace Info.
Call Mark Flags
Executor/ Debugger
Dynamic Slicer
Programmer
Abstracted Source Program
Call-Mark Slicer
Interactive Viewer/Editor
Static Slicer
Static Dependence Analyzer
Program Dependence Graph/ CED

Figure 7: Architecture of Osaka Slicing System

Execution of Sample Programs
Using this experimental system, we have executed various programs and obtained several metrics values. Table 2 shows the statistics of three sample programs. These values can vary with different slice criteria and input data. In these cases, we have chosen the criteria and inputs for a typical debugging situation. (The criteria are mostly program output variables, and the output statements are placed at the end part of program execution.)

Program $P1$ is a calender calculation program and the size is 88 lines. $P2$ is an inventory management program for a whole seller, and it is 387 lines long. $P3$ is also an extended version of the inventory management program



```
{Count Number Of Container}
procedure NoOfContainer(min,max :integer) ;
var i,j : integer;
begin
   NumContainer:=0;
   i:=min;
   while (i<=max)and(OutVFlag[i]=1) do
   begin
      j:=i+1;
      while OutContainerNo[j]=OutContainerNo[i] do j:=j+1;
      i:=j;
      NumContainer:=NumContainer+1
   end
end;


{Output DelliveryOrderTable}
procedure DelliveryOrderTable(p:integer );
var   j,l : integer;
begin
   l:=1;
   j:=1;
   OutSort;
   while OutVFlag[j]=1 do
   begin
      NoOfContainer(j,j+9);
      writeln('<<<<<OUTPUT<<<<<<<<');
      writeln('DelliveryOrderTable');
      writeln('No,        Name        Wine      Num,Container')
      writeln(RequestNo[p],'           ',Customer[p],'
      writeln('Container   Quantify     Mark');
      while (OutVFlag[l]=1)and(l<=j+9) do
      begin
         writeln(OutContainerNo[l],'        ',OutQuantify[l],
         OutContainerNo[l]:=0; OutQuantify[l]:=0; OutEmptyMo
         l:=l+1
      end;
      j:=j+10
   end;
   writeln
end; { DelliveryOrderTable }
```

```
load file Error8.pas
start analyze program
NUMBER OF STATEMENTS 321
analyze finished
data file test7
interpret start
interpret complete
calculating slice
SLICE      202
STATEMENTS 321
```

```
Container  Quantify     Mark
10000        10        0
10002        10        0
10003        10        0
10004        10        0
10005        10        0
10006        10        0
10007        10        1
10008        10        0
10010        10        0
10012        10        0
<<<<<OUTPUT<<<<<<<<
DelliveryOrderTable
No,        Name        Wine      Num,Container
2          2345         1           3
Container  Quantify     Mark
10013        10        0
10015        10        0
10016        10        0
```

Figure 8: Snapshot of Osaka Slicing System

of $P2$, and is 941 lines long. These programs include no pointer variables.

Table 3 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. In the case of dynamic slicing, the analysis is not necessary.

In Table 4, the execution time is shown. In the case of static slicing, the original program is executed; 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 dynamic dependences of variables. Therefore, the execution time contains the time for this construction. In the case of call-mark slicing, the execution time to mark callers is included.

Table 5 shows the time needed for collecting statements to be included in the resulting slices. In the case of static slicing, this would be done before execution. For dynamic slicing, the time for traversing the dynamic dependences is counted. For call-mark slicing, this is the time for Step 3.

We discuss these tables in detail in the next section.

Table 2: Size of Various Slicing Results (lines of code)

program	static	dynamic	call-mark
$P1$ (88 lines)	27	14	22
$P2$ (387 lines)	175	139	156
$P3$ (941 lines)	324	50	166

5 DISCUSSION

Interpretation of Program Execution Data

- Slice Size

  As shown in Table 2, the result of the call-mark slice is between the static and dynamic slices. It is always smaller and better than the static slice and bigger and worse than the dynamic slice.

- Pre-Execution Analysis:

  As shown in Table 3, call-mark slicing needs a little extra time compared to the static slicing. This is natural since we have to construct a PDG as in

Table 1: Auxiliary nodes

Auxiliary Nodes		Notation
exit-node	node to propagate the influence through the return value of a function, every procedure has its own exit node.	f–*exit*
in-node	node to propagate the influence of global variables from the outside of a procedure to the inside of it, every procedure has in-nodes of each global variables.	f_g–*in*
out-node	node to propagate the influence of global variables from the inside of a procedure to the outside of it, every procedure has out-nodes of each global variables.	f_g–*out*
parameter-node	node to propagate the influence through the parameters of a procedure, every procedure has its own parameter-nodes corresponding to its parameters.	f_p–*par*

Table 3: Pre-Execution Analysis Time (*ms* by Pentium-II 300MHz with 256MB Memory)

program	static	dynamic	call-mark
$P1$	22	N/A	23
$P2$	1,275	N/A	1,362
$P3$	5,652	N/A	8,670

Table 4: 4 Execution Time (*ms* by Pentium-II 300MHz with 256MB Memory)

program	static	dynamic	call-mark
$P1$	38	87	47
$P2$	48	903	53
$P3$	4,046	31,635	4,104

static slicing, and additionally we have to analyze the execution dependences.

- Execution Time

  The execution time shown in Table 4 indicates that the overhead of the dynamic slicing is very big. For $P3$, it is almost 32 seconds where about 200,000 lines of code are executed with maximum 90 M bytes memory use. 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.

Table 5: Slice Collection Time (*ms* by Pentium-II 300MHz with 256MB Memory)

program	static	dynamic	call-mark
$P1$	1	199	1
$P2$	5	2,863	8
$P3$	93	1,182	80

- Slice Collection Time

  As shown in Table 5, 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. Furthermore, in $P3$, it is better than the static one. This is because call-mark slicing removes parts of the PDG so that the searching space within the PDG is smaller than that for static slicing.

Relation to Other Methods

As discussed above, the call-mark slicing is a very promising approach to provide efficient and practical tools to localize a programmer's attention.

The pre-execution analysis, execution, and slice collection times are almost the same as those for static slicing, and the effect of the slicing is much better than the static one. It seems that this approach provides a good compromise between effectiveness and overhead.

Others have worked on combining static and dynamic information for slicing[4, 6].

Hybrid slicing[6] targets a very similar goal as ours. 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 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 approach, 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. In [6], the idea of the hybrid slicing was proposed, although no implementation nor execution data was presented.

In [4], 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[12, 16]. A generalized environment for developing analysis algorithm is proposed in [16]. It uses denotational frameworks to specify analysis algorithms; however, practicability of the generated algorithms for analysis tools is not known. In [12], 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.

Application Domain and Limitation of Call-Mark Slice
Our main target of this slicing method is a debugging environment as discussed in previous sections. 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 is limited.

Relation to Program Profile Information
Our approach, call-mark slicing, uses information of whether or not each function/procedure call statement in the program is executed.

The precision of our slices can be improved if we take such information of all statements in the program. This approach 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 approach than the call-mark slice. 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 result slice size.

6 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 an efficient and effective slicing method, call-mark slicing. This method uses lightweight run-time execution, and has a similar 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 this slice algorithm. Also we have executed sample programs, and confirmed our approach.

We are planning to extend our approach to reduce static analysis overhead, using call-mark information. The pre-execution analysis for the call-mark slicing may be done after the execution and before the slice collection. Using the call-mark information, unnecessary dependence analysis would be deleted, and total analysis time could be improved.

Also, empirical evaluation of the call-mark slicing will be made. We have performed an experiment to evaluated the effectiveness of static slicing. We plan to similarly explore the effectiveness of call-mark slicing using controlled experiments.

ACKNOWLEDGEMENTS
The authors are very grateful to the reviewers for many helpful comments.

This work is partly supported by Ministry of Education, Science, Sports, and Culture, grant-in-aid for priority areas "Principles for Constructing Evolutionary Software", #10139223.

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] 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).

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

[6] Guputa,R., and Soffa, M. L. : "Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information", *Proceedings of the 3rd International Symposium on the Foundation of Software Engineering*, pp. 29–40, October (1995).

[7] Harrold, M. J., and Ci, N. : "Reuse-Driven Interprocedural Slicing", *Proceedings of the 20th International Conference on Software Engineering*, pp. 74–83, Kyoto, Japan, April (1998).

[8] 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).

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

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

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] 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).

[16] 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).

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