Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information

Tomonori Takada, Fumiaki Ohata, Katsuro Inoue
Department of Informatics,
Graduate School of Engineering Science,
Osaka University
t-takada@ics.es.osaka-u.ac.jp

Abstract

When we try to debug or to comprehend a large program, it is important to separate suspicious program portions from the overall source program. Program slicing is a promising technique used to extract a program portion; however, such slicing sometimes raises difficulties. Static slicing sometimes produces a large portion of a source program, especially for programs with array and pointer variables, and dynamic slicing requires unacceptably large run-time overhead.

In this paper, we propose a slicing method named "dependence-cache slicing", which uses both static and dynamic information. An algorithm has been implemented in our experimental slicing system, and execution data for several sample programs have been collected. The results show that dependence-cache slicing reduces a slice size by 30–90% from static slice size with an increased and affordable run-time overhead, even for programs using array variables. In the future, the dependence-cache slicing will become an important feature for effective debugging environments.

1. Introduction

Finding faults in source programs is a time-consuming activity in software testing and maintenance phases. Looking over an entire source program to find a fault is inefficient. We want to focus our attention on a specific portion of the source program to improve efficiency.

As a candidate for focusing aids, program slicing techniques [15] have been studied. Intuitively, a program slice is a collection of program statements that affect the value of a variable in a statement we are interested in. We want to concentrate our attention only on the statements in the slice so that we can effectively debug, test, and comprehend the source program. We have empirically evaluated the validity of program slicing techniques for debugging and program comprehension [9].

Much research and many applications for program slicing have emerged from the original work of Mark Weiser [15]. These slicing techniques are roughly categorized into two classes: static slicing and dynamic slicing.

Static slicing was first proposed by Weiser [15]. A static slice is a collection of program statements possibly affecting a variable's value at a particular program point. 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 lack of reduction is due to the nature of the analysis of the static slicing such that all possible input data and all possible control flows must be considered. Also, many difficult issues of analysis exist, e.g., aliasing of variable names, separation of array and structure data elements, and the tracking of pointer variables. In addition, object oriented programs make static analysis much more difficult because those programs contain dynamically bound methods.

Dynamic slicing was first proposed by Agrawal et. al. [1, 8]. A dynamic slice is a collection of executed program statements actually affecting a variable's value at a particular program point. Since dynamic slicing is based on an execution instance for a source program with specific input data, non-executed parts of the source program are automatically excluded. This makes the size of a dynamic slice generally smaller than a static one. However, computing a dynamic slice is costly, requiring significant memory and time resources because dynamic variable dependence relations must be tracked.

In this paper, we propose a new slicing method using both static and dynamic information.

In Section 2 we briefly review static and dynamic slicing. In Section 3 we propose dependence-cache slicing. We show our experiment using an Osaka Slicing System

in Section 4. In Section 5, our findings are discussed in association with related works. We conclude this paper with additional remarks in Section 6.

2. Overview of Static and Dynamic Slicing

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

2.1. 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)* relation, 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 denoted by $s_1 \dashrightarrow s_2$.

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

- s_1 defines v, and
- s_2 refers to v (we say s uses v), and
- at least one execution path from s_1 to s_2 without redefining v exists.

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

A *Program Dependence Graph (PDG)* is a directed graph whose edges show dependence relations (CD or DD) between statements, and whose nodes are statements such as conditional predicates or assignment statements in a program. For the 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 *slicing criterion*) in a program is a collection of statements corresponding to the nodes in the PDG, which possibly reach s using v first, following transitive CD and DD edges. The static slice for variable d at line 24 as the slicing criterion for the program is a collection of all statements except for the message output statements (lines 12, 14, 16) shown in Figure 3.

```
 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

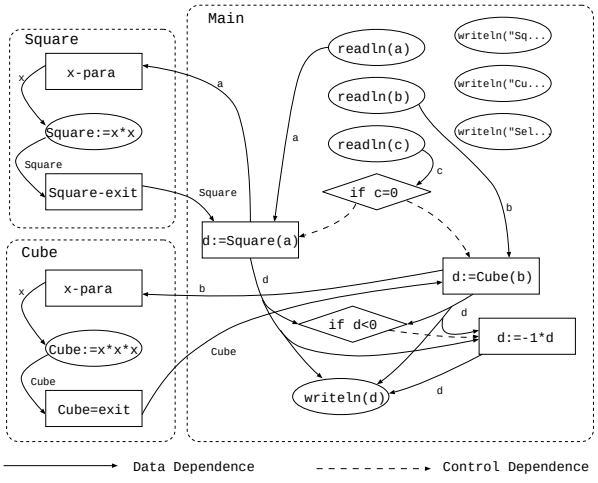

Figure 2. Program Dependence Graph (PDG) of the Program shown in Figure 1.

```
 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. Dynamic Slicing

For dynamic slicing, the analysis target is an *execution trace* in contrast to a source program for static slicing. An execution trace is a sequence of statements that are actually executed for an input data. The rth-executed statement in an execution trace is called execution point r.

When all of the following conditions are satisfied, we say that a *dynamic control dependence (DCD)* relation, from execution point r_1 to execution point r_2, exists:

- r_1 is a conditional predicate, and
- the result of r_1 determines whether r_2 is executed or not.

When the following conditions are all satisfied, we say that a *dynamic data dependence (DDD)* relation, from execution point r_1 to execution point r_2 by a variable v, exists:

- r_1 defines v, and
- r_2 uses v, and
- no execution path from s_1 to s_2 with a re-defining v exists.

A *dynamic dependence graph (DDG)* is created using the dynamic dependence relations: DCD and DDD.

Consider an execution point r and variable v in an execution trace e whose input variables set is $\mathcal{X}$. The triple ($\mathcal{X}$, r, v) is called the *dynamic slicing criterion*.

A *dynamic slice* for dynamic slicing criterion ($\mathcal{X}$, r, v) is computed by tracing DDG's edges backward from the node for r, and by mapping all reachable nodes into the source program.

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

Dynamic slicing is based on a single execution path, and it can give narrower slices than static slices. This situation is preferable in a debugging situation, since it is easier for us to focus our attention on smaller slices.

2.3. Approaches to reducing the Dynamic Slicing Overhead

Dynamic slicing is useful for software testing and maintenance phases because it can extract smaller slices than the slices extracted by static slicing. But dynamic slicing requires a huge run-time overhead.

To reduce this overhead, several methods using both static and dynamic information exist. These method can be categorized into two types.

- Methods collecting an execution path
  These methods collect and use the information of execution paths to reduce slice-sizes. Importantly, these methods effectively collect the information of execution paths with lightweight run-time overhead. The Profiling Method [1], the Call Mark Slicing Method [13] and the Hybrid Slicing Method [4] are based on this idea.

- Methods collecting data flow information
  These methods collect and use the information from the data flow information to determine the data dependence relation of non-scalar (pointer, array, or structured) variables. The Reduced DDG Method [1] is based on this idea.

In the next section, we propose a new slicing method called, "dependence-cache slicing" that collects data flow information.

3. Dependence-Cache Slicing

3.1. Overview

Collecting the precise data dependence relations of variables with a static method is difficult in general, although the control dependence relations are fairly easily collected statically [6, 7, 10].

Once we execute a program with an input data set, we are able to collect actual dependence relations between statements, although the penalty for collecting precise dependence relations has a fairly high overhead.

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

Step 1 Static Control Dependence Analysis

> We statically construct a part of PDG, named PDG_{DS}. First, we prepare nodes for each statement or predicate statement, and then draw control dependence edges between nodes, as we do when constructing a PDG for static slicing. No data dependence edges are added to the graph at this step.

Step 2 Dynamic Data Dependence Collection

> The target program is executed with an input data set. Along the execution, dynamic data dependence relations are collected using the *data dependence collection algorithm* shown in the next section, and data dependence 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 in a backward manner, 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.

3.2. A Data Dependence Collection Algorithm

Figure 5 shows the data dependence collection algorithm used at Step 2 in Section 3.1.

For each variable v in a program, we prepare a cache, denoted by $C(v)$. For a static variable like a global variable, a cache is prepared before the program starts. For a dynamic variable like an automatic variable on a stack or a variable in the heap, a cache is prepared when the variable is allocated. At each point of the program execution, $C(v)$ keeps a node for a statement that most recently defined v.

When v is used (referred to) at a statement s, a data dependence edge from the node kept in $C(v)$ to the node for s is added to PDG_{DS} if the edge does not exist yet. When v is defined at s, $C(v)$ is updated to the node for s. We do this for all variables in s.

For an array or a structured variable, we prepare caches for each element of the variable. For example, for an array variable A that has ten elements $A[1]$, $A[2]$, ..., $A[10]$, we prepare caches $C(A[1])$, $C(A[2])$, ... $C(A[10])$.

When a pointer variable p is used in a statement s, we must consider not only p, but $p\uparrow$. Thus, direct and indirect references must be contained in PDG_{DS} as data dependence edges; 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 update cache $C(q\uparrow)$ with t, and we also add an edge $C(q) \xrightarrow{q} t$ (if not existing) since q is used at t.

For a dynamic variable, we prepare caches for each instance. Consider a case when a structured variable v has two elements: v_1 and v_2. We prepare caches $C(v_1)$ and $C(v_2)$. We perform the same operation as with the algorithm shown in Figure 5 when v_1 or v_2 is used or defined independently. When the whole structure v is defined at statement s, we do $C(v_1) := s$ and $C(v_2) := s$. When v is used at s, we add data dependence edges $C(v_1) \xrightarrow{v_1} s$ and $C(v_2) \xrightarrow{v_2} s$ to PDG_{DS} (v_1 and v_2 are element names that can be statically specified).

The cache space required by this algorithm is proportional to the number of variables (variable elements) used at the program execution, and the run-time overhead is proportional to the number of the variable access.

Input

PDG_{DS}: Partially constructed PDG

P: Target Program

I: Input set 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 PDG

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 also dynamically prepare a cache initialized with the not-assigned mark }
2. Repeat following until 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 to) 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 5. Data Dependence Collection Algorithm

4. Experiments

4.1. Overview of the Osaka Slicing System

In order to investigate various slicing algorithms, we have developed a software development and debugging environment called the *Osaka Slicing System* [14]. The target language is Pascal.

This system contains a program executor and debugger. We can compute static slices and dynamic slices on this system. In this work, we added the functions for computing call-mark slicing [13] and dependence-cache slicing.

4.2. Execution of Sample Programs

Using this system, we have executed various programs and obtained many metric values. Program $P1$ is a calendar calculation program. $P2$ is an inventory management program for a wholesaler. $P3$ is also an extended version of the inventory management program $P2$.

Figure 6 shows the slice sizes of three sample programs. These values can vary with different slice criteria and input data sets. Here, we show the 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 the program execution.)

Figure 7 shows the time needed for the analysis before the execution. In the case of the static slicing, this value is the time needed to construct a PDG. The time for computing both PDG and CED (caller statements with execution dependence) is counted for the call mark slicing. For the dependence-cache slicing, the value is the time needed for constructing an initial PDG_{DS}. In the case of the dynamic slicing, this kind of the analysis is not necessary.

In Figure 8, the execution time is shown. In the case of the static slicing, the original program is executed without any extra run-time overhead; thus this value represents the execution time of the original program. The execution for the dynamic slicing is performed in association with the construction of the DDG. Therefore, the execution time contains the time for this construction. The time for the dependence-cache slicing includes the time for caching data dependence relations and for adding data-dependence edges in PDG_{DS}. In the case of the call mark slicing, the time to mark callers is included.

Figure 9 shows the time needed for collecting statements to be included in the resulting slices. In the case of the static slicing and dependence-cache slicing, these are the times needed for traversing PDG and PDG_{DS} respectively. For the dynamic slicing, the time needed for traversing dynamic dependence relations is counted. For the call mark slicing,

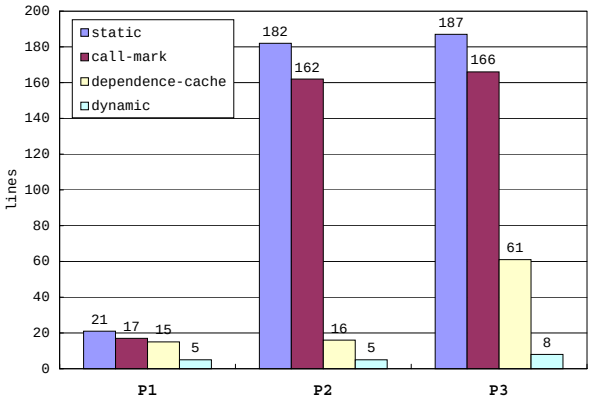

Figure 6. Size of Slice (lines)

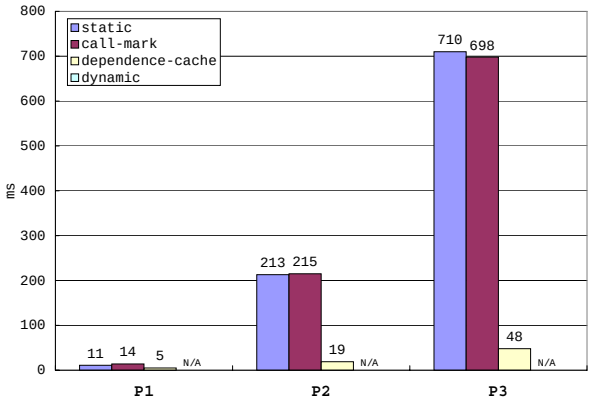

Figure 7. Pre-execution Analysis Time (ms)

this is the time needed for traversing PDG and for deleting nodes that are in non-executed functions from the slice.

We discuss these figures in detail in the next section.

5. Discussion

5.1. Interpretation of Experiment Data

- Slice Size

  Figure 6 shows the sizes of slices. The sizes of the dependence-cache slices are 9–71% of the static slices. These are smaller and better than the static slice sizes and bigger and worse than the dynamic slice sizes. Also, we can say that the dependence-cache slices are much better (smaller) than the call mark slices. This is because the dependence-cache slicing reflects data dependence relations on a particular execution path, while the call mark slicing only removes unexecuted

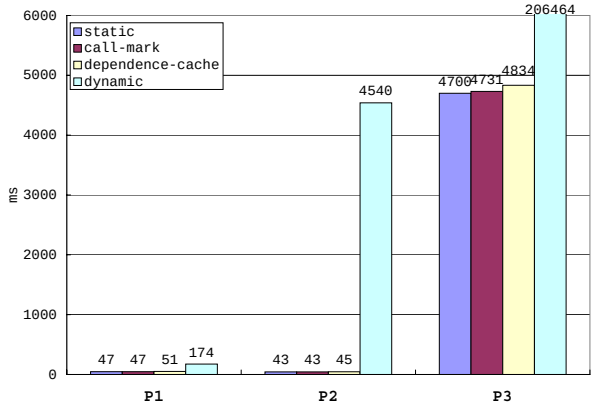

Figure 8. Execution Time (ms)

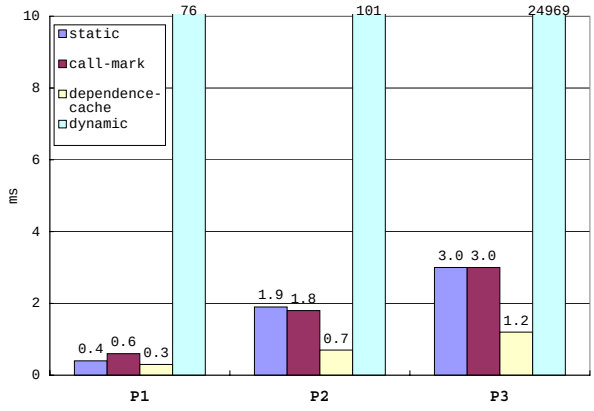

Figure 9. Slice Construction Time (ms)

parts of the statically detected data dependence relations based on the collected calling sequence. Reduction of $P2$'s and $P3$'s slice size by the dependence-cache slicing is larger than that of $P1$'s. This is because $P1$ uses only scalar variables and $P2$ and $P3$ employ array variables whose element-wise data dependence relations are only analyzed by dependence-cache slicing or the dynamic slicing.

- Pre-Execution Analysis

  As shown in Figure 7, dependence-cache slicing needs a lightweight pre-execution analysis only for control dependence relations, which is a fairly smaller overhead than the data dependence analysis for the static and call mark slicing.

- Execution Time

  The execution time shown in Figure 8 indicates that the overhead of dynamic slicing is huge. If the program execution becomes longer by, for example, a repeated execution of loops, then this overhead will cause a serious decline of performance so that a programmer will not find the dynamic slicing acceptable.

  For the dependence-cache slicing, extra execution time is required; however, the values are much smaller than for those of the dynamic slicing.

  The execution time presented here is based on our interpretive system; therefore, the run-time overhead for these slicings might be masked by the overhead of the interpretive execution. This issue will be discussed in Section 5.2.

- Slice Construction Time

  As shown in Figure 9, dynamic slicing requires a long time to collect the slice result. For dependence-cache slicing, collecting the slice result takes less time than for the static slicing. This is because dependence-cache slicing constructs PDG_{DS}, which is smaller than PDG, so that the searching space within the PDG_{DS} is smaller than that for static slicing. For dynamic slicing, traversing a large DDG is evidently quite costly.

5.2. Application Domain and Limitation of Dependence-Cache Slicing

The above experiment was done under the interpreter environment of the Osaka Slicing System. Here, we discuss its characteristics of the analysis and execution times.

We have written a merge sort program in C. This program has been modified to collect data dependence relations with the dependence-caches during the execution of this merge sort. The execution speed of the compiled code with the dependence-caches was 8.6 times slower than the original code. This indicates that the run-time overhead of the dependence-cache slicing is large under a compiler-based environment [1]. However, this overhead can be minimized if we collect the data dependence information only for array and pointer variables. Data dependence relations for non-array or non-pointer variables are fairly easily determined statically. Thus, we should analyze only difficult dependence relations at the execution time. Based on this idea, we have modified the C merge sort program again, so that the data dependence relations for only array elements were collected dynamically. The execution-time ratio between the original program and the partially dependence caching program became 1 to 3.4, and we consider this practically acceptable.

As we can see in Figure 6, the dependence-cache slices are larger than the dynamic slices. This difference is based on the rationale that the dependence-cache 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]) ;
```

The dynamic slice for this program is a set of statements 1, 3, 4 and 5. On the other hand, the dependence-cache slice is a set of statements 1, 2, 3, 4, and 5. The dependence-cache slice includes statement 2 as its result. This is because dependence-cache slicing cannot distinguish the first and second occurrences of statement 4 execution, and dependence relations from both statements 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, the dependence-cache slice size is fairly smaller than the static slice, and we think that this approach is a practical and promising method which compromises between effectiveness and overhead.

5.3. Related Works

A few works focus on collecting data flow information dynamically. The Reduced DDG Method [1] is one of them. In this method, the same sub-structure of DDG has been identified and shared with one structure. This method has achieved both the precision of dynamic slicing and the reduction of required memory space. However, such a method

[1] Also, this result suggests that the execution overhead for dynamic slicing is unacceptably huge.

requires a similarity checking of DDG at execution time, so the run-time overhead is serious.

Researches exist in which pointer and array variables are statically analyzed [6, 7]. Much of this research tries to statically determine possible aliases of pointer variables and array elements, but these still remain the uncertain cases [16]. Since our dependence-cache slicing uses dynamic information, we can get reasonable slice precision with affordable execution overhead.

In [3], a constrained slice, which is a generalization of static and dynamic slices, is proposed. This method takes a subset of the inputs of the program as a symbolic program execution. Using this input constraint, the program is rewritten and dependence relations are computed. However, the efficient implementation of such a generalized approach does not exist. It is also not known whether or not it is useful practically.

Hybrid slicing [4] reduces the static slice size by using two types of dynamic information: breakpoint information and call history information. The former information, which is supplied by the programmer, 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 weakness of hybrid slicing is that we have to specify appropriate breakpoints to get a better slice, and this method requires a fairly large amount of memory space to record the call history proportional to the program execution length.

Call mark slicing [13] uses the 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 for all the statements in the program. This approach, mentioned in [1] as the type 1 method, can be implemented using a similar method to computing profiling and program coverage information. For each statement, we employ a one bit flag whether the statement is executed or not. The mechanism is simple; however, it requires more run-time overhead and a 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.

Other methods focus on the semantics of programs. Conditioned slicing [2] employs a condition in a slicing criterion. Statements that do not match the condition are deleted from the slice. Amorphous slicing [5] allows for simplifying transformations that preserve semantic projection. These methods cannot be compared to the dependence-cache slicing directly because the approaches differ. However, we think dependence-cache slicing can be combined with these methods if both a semantic approach and a reduction of run-time overhead are needed.

6. Conclusions

We have proposed a lightweight dynamic slicing method, dependence-cache slicing. This method requires simple static analysis and lightweight run-time data dependence collection. The resulting slices are smaller than the corresponding static ones, but larger than the corresponding dynamic ones. We have implemented this slice algorithm on our experimental interpreter system. We have also executed various sample programs, and confirmed our approach.

For future work, we will also evaluate dependence-cache slicing through user testing. In addition, we plan to design a debugging environment based on a compiler-based system, rather than on the current interpreter-based system. The compiler-based system will be able to compute dependence-cache slices, associated with various debugging features. This compiler will generate a run-time environment with dependence-caches, and the generated codes will automatically collect dynamic data dependence relations. This information will be displayed as a slice or an another debugging aid when requested by users, and will provide more fruitful possibilities for fault localization.

Acknowledgments

The authors are grateful to Akira Nishimatsu, Minoru Jihira, Yoshiyuki Ashida, and Shinji Kusumoto of Osaka University who have contributed to designing and implementing the dependence-cache slicing system.

References

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

[2] Canfora, G., Cimitile, A., and De Lucia, A.: “Conditioned Program Slicing”, *Information and Software Technology*, vol. 40, no. 11/12, November 1998, pp. 595-607.

[3] 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).

[4] 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.

[5] Harman, M. and Danicic, S.: “Amorphous program slicing”, *IEEE International Workshop on Program Comprehesion (IWPC’97)*, pp. 70-79, Dearborn,

Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[6] 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).

[7] 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).

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

[9] Kusumoto, S., Nishimatsu, A., Nishie K., and Inoue, K.: “Experimental Evaluation of Program Slicing for Fault Localization”, Empirical Software Engineering, 7, pp.49-76 (2002).

[10] 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).

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

[12] Nishimatsu, A., Kusumoto, S., and 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).

[13] 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.

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

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