Acta Cybernetica **15** (2002) 489–508.

Handling Pointers and Unstructured Statements in the Forward Computed Dynamic Slice Algorithm

Csaba Faragó* and Tamás Gergely†

Abstract

Different program slicing methods are used for debugging, testing, reverse engineering and maintenance. Slicing algorithms can be classified as a static slicing or dynamic slicing type. In applications such as debugging the computation of dynamic slices is more preferable since it can produce more precise results. In a recent paper [5] a new so-called "forward computed dynamic slice" algorithm was introduced. It has the great advantage compared to other dynamic slice algorithms that the memory requirements of this algorithm are proportional to the number of different memory locations used by the program, which in most cases is much smaller than the size of the execution history. The execution time of the algorithm is linear in the size of the execution history. In this paper we introduce the handling of pointers and the jump statements (`goto`, `break`, `continue`) in the `C` language.

1 Introduction

Program slicing methods are widely used for debugging, testing, reverse engineering and maintenance (e.g. [3], [7], [2], [4]). A slice consists of all statements and predicates that might affect the variables in a set V at a program point p [8]. A slice may be an executable program or a subset of the program code. In the first case the behaviour of the reduced program with respect to a variable v and program point p is the same as the original program. In the second case a slice contains a set of statements that might influence the value of a variable at point p. Slicing algorithms can be classified according to whether they only use statically available information (*static slicing*) or compute those statements which influence the value of a variable occurrence for a specific program input (*dynamic slice*).

In many applications (e.g. debugging) the computation of dynamic slices is more preferable since it can produce more precise results (i.e. the dynamic slice is smaller than the static one). In this paper we will focus on dynamic slicing.

*Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanúk tere 1., H-6720 Szeged, Hungary, e-mail: `csaba@petra.hos.u-szeged.hu`

†Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanúk tere 1., H-6720 Szeged, Hungary, +36 62 544143, e-mail: `gertom@inf.u-szeged.hu`

In [5] Tibor Gyimóthy, Gábor Forgács and Árpád Beszédes introduced a method for the forward computation of dynamic slices (i.e. at each iteration of the process, slices are available for all variables at the given execution point). However, the method presented was applicable only to very simple programs (with one procedure, scalar variables and simple assignment statements only). In [9] the handling of the procedures and the implementation of the algorithm were shown. In this paper we show how to handle pointers and the jump statements in the C programs. In addition to the `goto` statement it solves the problem of `break` and `continue` statements, which can be regarded as special cases of the `goto` statement. The handling of the `switch-case-dafault` statement is also mentioned.

The paper is organized as follows. After discussing the background of slicing, the "forward computed dynamic slice" method is introduced. The handling of pointers and jump statements are then elaborated on in Sections 3 and 4. Finally, we give a summary of what we have done so far.

2 Forward computing of the dynamic slice

2.1 Original algorithm

In some applications static program slices contain redundant instructions. This is the case for debugging, for instance, where we have dynamic information as well. Hence debugging may require smaller slices, which improves the efficiency of the bug finding process ([1], [6]). The goal of the introduction of dynamic slices was to determine more precisely those statements that may contain program bugs, assuming that the failure has occurred for a given input.

Consider the example program in Figure 1. The static slice of this code with respect to the variable s at vertex 12 contains all the statements.

Prior to the description of a new dynamic slice algorithm we introduce some basic concepts and notations.

A feasible path that has actually been executed will be referred to as an *execution history* and denoted by EH. Let the input be $a = 0$, $n = 2$ in the case of our example. The corresponding execution history is $\langle 1,2,3,4,5,7,8,10,11,7,$ $8,10,11,7,12\rangle$. We can see that the execution history contains instructions which come in the same order as they have been executed, so $EH(j)$ gives the serial number of the instruction executed at the j^{th} step, referred to as *execution position* j.

To distinguish between multiple occurrences of the same instruction in the execution history we make use of the notion of *action*. It is a pair (i, j) which is written as i^j, where i is the serial number of the instruction at the execution position j. For example 12^{15} is the action for the output statement of our example for the same input as above.

The *dynamic slicing criterion* is a triplet $(\mathbf{x}, i^j, V)$ where $\mathbf{x}$ denotes the input, i^j is an action in the execution history, and V is a set of the variables. For a slicing criterion a dynamic slice can be defined as the set of statements which may affect

```
     #include <stdio.h>
     int n, a, i, s;
     void main()
     {
1.     scanf("%d", &n);
2.     scanf("%d", &a);
3.     i = 1;
4.     s = 1;
5.     if (a > 0)
6.       s = 0;
7.     while (i <= n) {
8.       if (a > 0)
9.         s += 2;
         else
10.        s *= 2;
11.      i++;
       }
12.    printf("%d", s);
     }
```

Figure 1: Example program

the values of the variables in V.

We apply a program representation which only considers the definition of a variable, and use of variables, and direct control dependences. We refer to this program representation as a *D/U program representation.* An instruction of the original program has a D/U expression of the form:

$$i.\ d:\ U,$$

where i is the serial number of the instruction, and d is the variable that gets a new value from the instruction in the case of assignment statements. For an output statement or a predicate d denotes a newly generated "output variable"- or "predicate-variable"-name of this output or predicate, respectively (see the example below). Let $U = \{u_1, u_2, ..., u_n\}$ such that any $u_k \in U$ is either a variable that is used at i or a predicate-variable from which instruction i is (directly) control dependent. Note that there is at most one predicate-variable in each U. (If the *entry* statement is defined, there is exactly one predicate-variable in each U.)

Our example has a D/U representation shown in Figure 2.

Here $p5$, $p7$ and $p8$ are used to denote predicate-variables and $o12$ denotes the output-variable, whose value depends on the variable(s) used in the output statement.

Now we are ready to derive the dynamic slice with respect to an input and

$i.$	d	:	U
1.	n	:	$\emptyset$
2.	a	:	$\emptyset$
3.	i	:	$\emptyset$
4.	s	:	$\emptyset$
5.	$p5$	:	$\{a\}$
6.	s	:	$\{p5\}$
7.	$p7$	:	$\{i, n\}$
8.	$p8$	:	$\{p7, a\}$
9.	s	:	$\{s, p8\}$
10.	s	:	$\{s, p8\}$
11.	i	:	$\{i, p7\}$
12.	$o12$	:	$\{s\}$

Figure 2: D/U representation of the program

the related execution history based on the D/U representation of the program as follows. First, we process each instruction in the execution history starting from the first executed statement. Then after processing an instruction $i.\ d : U$, we derive a set *DynSlice*(d) that contains all those statements which affect d when instruction i has been executed. By applying the D/U program representation the effect of data and control dependences may be treated in the *same way*. After an instruction has been executed and the related *DynSlice* set has been derived, we determine the *last definition* (serial number of the instruction) for the newly assigned variable d denoted by $LS(d)$. Put simply, the last definition of variable d is the serial number of the instruction where d is last defined (considering the instruction $i.\ d : U$, $LS(d) = i$). Clearly, after processing the instruction $i.\ d : U$ at the execution position j each $LS(d)$ has the value i for each subsequent executions until d is redefined next time. We also use $LS(p)$ for predicates, which denotes the last definition (evaluation) of predicate p. For example, if $EH(10) = 7$ (the current action is 7^{10}) then $LS(d) = 7$.

Now the dynamic slices can be determined as follows. Assume that we are running a program having an input t. After an instruction $i.\ d :\ U$ is executed at position p, *DynSlice*(d) contains just those statements involved in the dynamic slice for the slicing criterion $C = (t, i^p, U)$. *DynSlice* sets are determined by using the relation below:

$$DynSlice(d) = \bigcup_{u_k \in U} \Big(DynSlice(u_k) \cup \{LS(u_k)\} \Big)$$

After *DynSlice*(d) has been evaluated we determine $LS(d)$ for assignment and predicate instructions, i.e.

$$LS(d) = i$$

Note that this computation order is strict since when we determine $DynSlice(d)$, we have to consider whether $LS(d)$ occurred at a former execution position instead of p (like the program line `x = x + y` in a loop).

```
program DynamicSlice
begin
  Initialize $LS$ and $DynSlice$ sets
  ConstructD/U
  ConstructEH
  for $j = 1$ to number of elements in EH
    the current D/U element is $i^j$. $d: U$
    $DynSlice(d) = \bigcup_{u_k \in U} (DynSlice(u_k) \cup \{LS(u_k)\})$
    $LS(d) = i$
  endfor
  Output $LS$ and $DynSlice$ sets for the last definition of all variables
end
```

Figure 3: Dynamic slice algorithm

A formal version of the forward dynamic slice algorithm is presented in Figure 3. Note that the construction of the execution history is achieved by instrumenting the input program and executing this instrumented code. The instrumentation procedure is discussed in [9].

We will illustrate how the above method works by applying it to our example program in Figure 1 with the execution history ⟨ 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 ⟩.

During the execution the following values are returned:

Action	d	U	$DynSlice(d)$	$LS(d)$
1^1	n	$\emptyset$	$\emptyset$	1
2^2	a	$\emptyset$	$\emptyset$	2
3^3	i	$\emptyset$	$\emptyset$	3
4^4	s	$\emptyset$	$\emptyset$	4
5^5	$p5$	$\{a\}$	{2}	5
7^6	$p7$	$\{i, n\}$	{1,3}	7
8^7	$p8$	$\{p7, a\}$	{1,2,3,7}	8
10^8	s	$\{s, p8\}$	{1,2,3,4,7,8}	10
11^9	i	$\{i, p7\}$	{1,3,7}	11
7^{10}	$p7$	$\{i, n\}$	{1,3,7,11}	7
8^{11}	$p8$	$\{p7, a\}$	{1,2,3,7,11}	8
10^{12}	s	$\{s, p8\}$	{1,2,3,4,7,8,10,11}	10
11^{13}	i	$\{i, p7\}$	{1,3,7,11}	11
7^{14}	$p7$	$\{i, n\}$	{1,3,7,11}	7
12^{15}	$o12$	$\{s\}$	{1,2,3,4,7,8,10,11}	12

The final slice is the union of $DynSlice(o12)$ and $\{LS(o12)\}$. (See Figure 4)

```
#include <stdio.h>
int n, a, i, s;
void main()
{
1.   scanf("%d", &n);
2.   scanf("%d", &a);
3.   i = 1;
4.   s = 1;
5.   if (a > 0)
6.     s = 0;
7.   while (i <= n) {
8.     if (a > 0)
9.       s += 2;
       else
10.      s *= 2;
11.    i++;
     }
12.  printf("%d", s);
}
```

Figure 4: The framed statements give the dynamic slice

2.2 Analysis of the algorithm

Let's analyze the duration and the memory requirement of the algorithm! It's very hard to figure out the exact requirements. We'll try to make an average-case analysis with referring to the worst-case, too.

First let's consider the duration. The initializations are approximately linear to the different memory locations. The DU construction is linear to the length of the executable source code. One can ask why don't we say that it is linear to the statements? The reason is that the duration of one step is dependendent to the length of the statement. For example it takes less time to build up the DU for a=b+c than for a=b*c-f(b,b+c)-c. The construction of the EH is linear to the execution of the original program. Unfortunately the constant multiplier hidden by "theta-notation" (it is used in analysis of the algorithms) is hardly predictable: it is dependent to the number of pointers etc., which can vary from zero up to the whole program. The duration of the first and the third pseudo-statement within the main for cycle is constant. The union statement's duration is critical within the algorithm, but unfortunately it is very hardly predictable. In worst case the U set can hold all the variables (i.e. different memory locations + pseudo-variables (e.g. labels etc.)), and all the dynamic slices holds all the statements within the program. It this case the main cycle's duration is proportional to `<execution history> * <memory locations> * <number of statements>`. But in the most normal programs the size of the `U` set is not so big, in most cases it holds about 4-5 elements. There exist not too much such statement where the `U` set contains more than 10 elements. A dynamic slice in most cases contains not too much statement, but it seems in many cases it is linear to the size of the program. The duration of the output depends to the numbers of slice criteria etc., but it is less than the countation, of course. According to these the average execution time of the algorithm is `O(|EH|*|statements|+|memory|)`.

Now let's analyze the memory requirements. The most relevant memory requirement takes the storage of the temporal slice results, i.e. the `U` sets; the others (e.g. memory requirements of the initialization part etc.) can be ignored. In the worst case every variable (i.e. every memory location) contains all the statements, so in this case the memory requirement is `O(<number of different memory locations> * <number of statements>)`. In fact it is very unlikely to use such a big memory, it is rather linearly proportional to the different memory locations used during the program with a bigger constant. At bigger programs in the most cases the memory requirement of the dynamic counting algorithm is linear to the memory requirement of the original program (with a bigger constant).

2.3 Extending the basic algorithm

In order to handle the pointers, the variables are identified by their addresses and not by their names. This approach has several good advantages.One is that it solves the problem of the variables with the same name but different program scope.

The address of a variable can only be determined dynamically after its declaration, but the DU is derived from the static source code. Hence there are two DU structures: a static DU which contains variable names, and a dynamically resolved DU (dynamic DU) which contains addresses. (Note that the dynamic DU may change during the program execution due to a change in variable address, pointer value, etc. The neccessary parts of the dynamic DU are computed at each step using the static DU and the (extended) execution history.)

In a C program there may be several variables present with the same name but in different scopes. The address of a variable with a specific name may depend on the scope of the expression where the variable is used. So the algorithm must keep track of the scopes and maintain a stack structure for each function in order to store the addresses of the variables. Each time a new scope is begun, a new address table is created at the top of the stack, and when a new variable declaration occurs, the name and address are recorded in this new table. For the address of a variable the address tables are searched from the top to the bottom of the stack of the actual function. When a scope leaved, the top element of the stack is discarded. The first element at the bottom of each such stack is the same: the address table of global variables (these can be accessed by every function).

3 Handling pointers

In this subsection the handling of the pointers, arrays and structures are described. The methods we introduce are for the C language, but the general idea may be applied to other languages too. When code is shown, the "common" parts of the code (such as function headers, includes) are hidden, as well as some function calls from the instrumented code.

3.1 Pointers

The address of a variable does not change in its scope, so after it is determined it can be used any number of times. But the value of a pointer can change at any time and must be determined every time the pointer occurs. Consider the following program code:

```
    int x, *p;
1.  x=1;
2.  p=&x;
3.  *p=2;
4.  print(x);
```

It is readily seen that only lines 2 and 3 affect the value of x in line 4, while the first one doesn't. Statically it is almost impossible to detect these kind of dependencies. Most of the static algorithms either include the whole program, or exclude lines 2 and 3 and include only those like the first line, which will produce incorrect slices.

Statically, only the following dependencies can be determined:

line	def	:	USE
1	x	:	$\emptyset$
2	p	:	$\emptyset$
3	$PTR1$	:	$\{p\}$
4	OUT	:	$\{x\}$

where $PTR1$ indicates that a memory location is defined via a pointer. Of course, this memory location depends on the value of the pointer itself. Using dynamic information the variables can be converted into memory locations. This means that addresses 01 and 02 can be used instead of variables x and p, respectively. Dynamically, the value of $PTR1$ is also known. Extracting the information from the execution history, the result is the following (dynamically resolved) DU:

step	line	def	:	USE	$DynSlice(def)$
1	1	01	:	$\emptyset$	$\emptyset$
2	2	02	:	$\emptyset$	$\emptyset$
3	3	01	:	{02}	{2}
4	4	OUT	:	{01}	{2,3}

The technical procedure for the extraction of the addresses is called program instrumentation. This means modifying the program code in such way that the program retains its original behaviour, but also generates some extra information (eg. execution history, runtime addresses, block information). The instrumented code is similar to the following (actually, it is slightly more involved):

```
    int x, *p;
    remember("x", &x);
    remember("p", &p);
1.  x=1;
2.  p=&x;
3.  *dump("PTR1", p)=2;
4.  print(x);
```

The `remember` function writes the address of the variable into the (extended) execution history. This information is used during the execution of the slicing algorithm to create the dynamic DU. The `dump` function writes the value of its second parameter (in this case pointer value), and simply returns it. In the static DU case the third line contains the pointer variable $PTR1$, which can be resolved within the algorithm to an address using the previously dumped pointer value.

3.2 Arrays

In the C language the arrays and the pointers are practically the same, and the conversion from one to the other is quite simple. The i^{th} element of an array `t`, denoted by `t[i]`, can be expressed as a pointer `*(t+i)`. Then, when an element of an array is referenced, it is treated as a pointer in the DU and then its address is written out. Consider the following example:

```
    int t[5];
    int *p;
1.  t[0]=1;
2.  *(t+1)=2;
3.  t[2]=2*t[1];
4.  print(t[2]);
```

The static DU of the previous program is the following:

line	def	:	USE
1	$PTR1$	:	$\emptyset$
2	$PTR2$	:	$\emptyset$
3	$PTR3$	:	$PTR4$
4	OUT	:	$PTR5$

The references to array elements are converted to pointers, as mentioned earlier. Each pointer or array element occurrence has a unique identifier in the static DU. The instrumented code is:

```
    int t[5];
    remember("t", t);
1.  *dump("PTR1",&t[0])=1;
2.  *dump("PTR2",t+1)=2;
3.  *dump("PTR3",&t[2])=2*(*dump("PTR4",&t[1]));
4.  print(*dump("PTR5",&t[2]));
```

Assume that the array is placed at address 10. In this case the dynamic DU and the slice are:

step	line	def	:	USE	$DynSlice(def)$
1	1	10	:	$\emptyset$	$\emptyset$
2	2	11	:	$\emptyset$	$\emptyset$
3	3	12	:	{11}	{2}
4	4	OUT	:	{12}	{2,3}

3.3 Structures

The offset of the members of a structure could be determined statically but the computation of dynamic addresses would be quite complicated. Instead, the members of a structure will also be treated as pointers. In this way the structures are reduced to pointers. Consider the example:

```
    struct s {
        int a;
        int b;
    };
    struct s x,y;
1.  x.a=1;
2.  x.b=2;
3.  y=x;
4.  print(y.b);
```

Here, there are three structure members. The `x.a` in line 1, `x.b` in line 2 and `y.a` in line 4 are converted to $PTR1$, $PTR2$ and $PTR3$ respectively. Structures `x` and `y` are not converted to pointers, but are scalar variables. The static DU of the example is:

line	def	:	USE
1	$PTR1$	:	$\emptyset$
2	$PTR2$	:	$\emptyset$
3	y	:	$\{x\}$
4	OUT	:	$\{PTR3\}$

Let us suppose that the addresses of the structures and elements are the following:

item	x	x.a	x.b	y	y.a	y.b
address	01	01	02	03	03	04

As can be seen the address itself does not correctly describes a variable. Although the addresses of `x` and `x.a` are the same, they are still different. While `x.a` is located in a single memory cell, `x` occupies two cells: `01` and `02`. There is another reason for recording the structure size: the expression `y=x` in line 3. If the size of the structure is ignored the relation between `x.b` and `y.b` becomes indeterminable. In the instrumented code function `remember` has an additional parameter, the size of the variable. The addresses of the three structure members are written out by use of the function used for dump pointer values. The instrumented code is:

```
    struct s {
        int a;
        int b;
    };
    struct s x, y;
    remember("x", &x, sizeof(x));
    remember("y", &y, sizeof(y));
1.  *dump(&x.a)=1;
2.  *dump(&x.b)=2;
3.  y=x;
4.  print(*dump(&y.b));
```

Using the static DU and the dynamic information (addresses), the dynamic DU and the slice are the following:

step	line	*def* : *USE*	*DynSlice*(*def*)
1	1	01 : $\emptyset$	$\emptyset$
2	2	02 : $\emptyset$	$\emptyset$
3	3	03 : {01}	{1}
		04 : {02}	{2}
4	4	*OUT* : {04}	{2,3}

There are two memory locations defined in step 3, each with its own dependency. When the expression y=x is evaluated, the value of the structure x is copied into structure y. This means that two memory cells from the address 01 are copied into the cells starting at 03. The algorithm recognizes this fact and creates the two dependencies in the dynamic DU from the (single) dependency $y : x$ in the static DU.

3.4 Size of pointers

The method still hasn't been quite refined. As well as recording the size of a structure, the size of all variables and pointers must be known. It is obvious that a pointer can point to any structure or the element of an array can be a structure also. So the function dump has one additional parameter: the size of the pointed type. In this way the algorithm can compute the correct dynamic DU. It can find all addresses defined or used.

3.5 Same memory locations used during execution

During a program execution memory locations are allocated and released dynamically for some variables. It may happen that, after releasing such a memory location (which can be implicit or explicit), another variable gets the same address. Could this have some detrimental effects on the algorithm? If all variables were initialized before its first use, the answer is no. If the second variable is used without initialization, the algorithm uses the slice of the previous variable. This behaviour of the algorithm is also correct since it shows how the (probably bad) program works.

4 Handling unstructured statements

An issue which must be dealt with is how we should handle the jump statements in the dynamic slicing algorithm. In this section C-specific jump statements are considered, but the method could be used in other programming languages as well.

In the next part the handling of the goto statement is described, along with the break, continue, and switch statements.

4.1 The goto statement

Where a goto statement occurs, the D/U structure is built up as follows. First, so-called *"label variables"* are introduced. Let the defined variable (d) be the previously introduced label variable called the real name of the label. It could also be an ordinal number, but for the sake of simplicity we use the previous name here. The use set (U) contains no "extra" variables, just the appropriate predicate variable, and we will find that it can contain label variables too.

The previously defined label variable is inserted into the use set (U) of those statements which occur after the corresponding label within the function. It is important to do this to the end of the function, not just in the appropriate block.

i.		*def*	:	*USE*
	`int i,j,k,l;`			
1.	`k=0;`	k	:	$\emptyset$
2.	`l=0;`	l	:	$\emptyset$
3.	`i=0;`	i	:	$\emptyset$
	`l1:`			
4.	`j=0;`	j	:	$\{l1\}$
	`l2:`			
5.	`k=k+i+j;`	k	:	$\{k, i, j, l1, l2\}$
6.	`l++;`	l	:	$\{l, l1, l2\}$
7.	`j++;`	j	:	$\{j, l1, l2\}$
8.	`if (j<2)`	$p8$	:	$\{j, l1, l2\}$
9.	`goto l2;`	$l2$	:	$\{p8, l1, l2\}$
10.	`i++;`	i	:	$\{i, l1, l2\}$
11.	`if (i<2)`	$p11$	:	$\{i, l1, l2\}$
12.	`goto l1;`	$l1$	:	$\{p11, l1, l2\}$
13.	`printf("%d",k);`	$o13$	:	$\{k, l1, l2\}$

Figure 5: Handling of the `goto` statement

If there are more labels, they are all handled in the same way. If the `goto` statement appears after the definition of the label, then of course it contains the just defined label variable. But this is not a problem because in the execution history it appears as a formerly defined variable. It can be defined by itself or by another `goto` statement. If no `goto` statement that jumps to a specific label is executed during the program, the last definition of that label remains undefined so it will not affect the result of the dynamic slice. The result contains all of the defined labels.

When the `goto` is executed during the program and the dynamic slice contains at least one of the statements after the definition of the label, then the result will at least contain the previous corresponding `goto` (and of course its predicate dependencies transitively). So it often unnecessarily increases the size of the dynamic slice and using lots of `goto` statements will make it hard to analyze the program.

An example is shown on Figure 5, and its results in Figure 6. As one might except, the use of `goto` statements resulted in a lot of dependencies.

Action (i^j)	*DynSlice*()	Action (i^j)	*DynSlice*()
1^1	$\emptyset$	16^{12}	$\{3,4,7,8,9,10,11\}$
2^2	$\emptyset$	17^4	$\{3,4,7,8,9,10,11,12\}$
3^3	$\emptyset$	18^5	$\{1,3,4,5,7,8,9,10,11,12\}$
4^4	$\emptyset$	19^6	$\{2,3,4,6,7,8,9,10,11,12\}$
5^5	$\{1,3,4\}$	20^7	$\{3,4,7,8,9,10,11,12\}$
6^6	$\{2\}$	21^8	$\{3,4,7,8,9,10,11,12\}$
7^7	$\{4\}$	22^9	$\{3,4,7,8,9,10,11,12\}$
8^8	$\{4,7\}$	23^5	$\{1,3,4,5,7,8,9,10,11,12\}$
9^9	$\{4,7,8\}$	24^6	$\{2,3,4,6,7,8,9,10,11,12\}$
10^5	$\{1,3,4,5,7,8,9\}$	25^7	$\{3,4,7,8,9,10,11,12\}$
11^6	$\{2,4,6,7,8,9\}$	26^8	$\{3,4,7,8,9,10,11,12\}$
12^7	$\{4,7,8,9\}$	27^{10}	$\{3,4,7,8,9,10,11,12\}$
13^8	$\{4,7,8,9\}$	28^{11}	$\{3,4,7,8,9,10,11,12\}$
14^{10}	$\{3,4,7,8,9\}$	29^{13}	$\{1,3,4,5,7,8,9,10,11,12\}$
15^{11}	$\{3,4,7,8,9,10\}$		

Figure 6: The result of program in Figure 5

4.2 The `break` statement

The `break` statement is practically equivalent to `goto` statement, which jumps out from the block of the appropriate `while`, `do...while`, `switch` or `for` statement to the first statement after this block. This statement can be handled as follows. The defined variable at every occurrence of the `break` statement should be an individual label variable. One form might be `break<Nr>`, where `<Nr>` is the ordinal number of the `break` statement within the program. All of the statements after the corresponding block are dependent on the previously defined label variable, just like in the case of `goto` statement. Note that if a label is placed just after the corresponding block and the `break` is replaced with a `goto` which jumps to that label, then the effect is the same.

An example of the `break` statement and results are shown in Figure 7 and Figure 8 respectively.

4.3 The `continue` statement

Like the `break` statement, we should define a separate label variable. This might be denoted by `continue<Nr>`, where `<Nr>` is the ordinal number of the `continue` statement within the program. It is defined in statements where `continue` occurs. The dependent statements are statements from the beginning of the block of the appropriate `for`, `while` or `do...while` statement to the end of the function. So the `continue` statement is always dependent upon itself.

$i.$		def	:	USE
	`int a,b,i;`			
1.	`a=1;`	a	:	$\emptyset$
2.	`b=1;`	b	:	$\emptyset$
3.	`i=2;`	b	:	$\emptyset$
4.	`while (i>0) {`	$p4$	:	$\{i\}$
5.	`    b--;`	b	:	$\{p4, b\}$
6.	`    i--;`	i	:	$\{p4, i\}$
7.	`    if (b==0)`	$p7$	:	$\{b\}$
8.	`        break;`	$break8$	:	$\{p7\}$
9.	`    a++;`	a	:	$\{p4, a\}$
	`}`			
10.	`printf("%d",a);`	$o10$	:	$\{a, break8\}$

Figure 7: Handling of the `break` statement

Action (i^j)	$DynSlice()$
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2,3,4\}$
6^6	$\{3,4\}$
7^7	$\{2,3,4,5\}$
8^8	$\{2,3,4,5,7\}$
9^{10}	$\{1,2,3,4,5,7,8\}$

Figure 8: The results of program in Figure 7

An example of the `continue` statement and results are shown in Figure 9 and Figure 10 respectively.

4.4 The `switch` statement

After the handling of `break` statement, the handling of the `switch` statement is quite straightforward.

At the place where the `switch` statement occurs a predicate variable is defined, just like in the case of `while` or `if`. All of the statements within the `switch` block are dependent on this predicate variable. If at least one statement within the switch block is included in the slice result, all of the `case` labels and the `default` label

$i.$		def	:	USE
	`int a,b,i;`			
1.	`a=1;`	a	:	$\emptyset$
2.	`b=1;`	b	:	$\emptyset$
3.	`i=2;`	b	:	$\emptyset$
4.	`while (i>0) {`	$p4$	:	$\{i, continue8\}$
5.	`    b--;`	b	:	$\{p4, b, continue8\}$
6.	`    i--;`	i	:	$\{p4, i, continue8\}$
7.	`    if (b==0)`	$p7$	:	$\{b, continue8\}$
8.	`        continue;`	$continue8$	:	$\{p7, continue8\}$
9.	`    a++;`	a	:	$\{p4, a, continue8\}$
	`}`			
10.	`printf("%d",a);`	$o10$	:	$\{a, continue8\}$

Figure 9: Handling of the `continue` statement

Action (i^j)	$DynSlice()$
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2,3,4\}$
6^6	$\{3,4\}$
7^7	$\{2,3,4,5\}$
8^8	$\{2,3,4,5,7\}$
9^4	$\{2,3,4,5,6,7,8\}$
10^5	$\{2,3,4,5,6,7,8\}$
11^6	$\{2,3,4,5,6,7,8\}$
12^7	$\{2,3,4,5,6,7,8\}$
13^9	$\{1,2,3,4,5,6,7,8\}$
14^4	$\{2,3,4,5,6,7,8\}$
15^{10}	$\{1,2,3,4,5,6,7,8,9\}$

Figure 10: The results of program in Figure 9

are included. Here the `break` statements are handled in the same way as described before.

An example of the `switch` statement and its results are shown in Figure 11 and Figure 12, respectively.

i.		*def*	:	*USE*
	`int a,b;`			
1.	`b=0;`	b	:	$\emptyset$
2.	`a=2;`	a	:	$\emptyset$
3.	`switch (a) {`	$p3$	:	$\{a\}$
	`case 1:`			
4.	`    b=5;`	b	:	$\{p3\}$
5.	`    break;`	$break5$	:	$\{p3\}$
	`case 2:`			
6.	`    b=3;`	b	:	$\{p3\}$
	`case 3:`			
7.	`    b++;`	b	:	$\{p3, b\}$
8.	`    break;`	$break7$	:	$\{p3\}$
	`default:`			
9.	`    b=6;`	b	:	$\{p3\}$
	`}`			
10.	`printf("%d",b);`	$o10$	:	$\{b, break5, break7\}$

Figure 11: Handling of the `switch` statement

Action (i^j)	*DynSlice*()
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\{2\}$
4^6	$\{2,3\}$
5^7	$\{2,3,6\}$
6^8	$\{2,3\}$
7^{10}	$\{2,3,6,7,8\}$

Figure 12: The results of program in Figure 11

5 Experimental results

Several experimental results confirmed that our dynamic slices are more precise than the static one. Among the test sources there are 3 medium sized: the `bzip` (a compression utility), the `bc` (a scientific calculator) and the `less` (this is a powerful text viewer program). The sizes of these programs is shown in the following table.

prog	lines	executable	files	bytes	functions
bzip	4495	1595	1	130 458	73
bc	11555	3220	20	312 722	138
less	21489	5400	43	639 036	363

The first column is the name of the program, the second one means the total lines of the source, the third is the size of the executable code (i.e. without comments etc.), the fourth is the number of source files, the fifth is the total length of the source code in bytes, and the last one means the number of the functions within the program.

With help of our program we made several executions on several slice criteria for all the 3 sources. The number of the different slice criteria and the number of the executions are shown in the next table.

program	criteria	executions	coverage
bzip	154	18	68%
bc	57	49	63%
less	50	14	45%

The last column shows the coverage of the program. A statement is defined to be covered if at least once is executed during all the tests. The coverage means the percentage of the covered statements related to the whole program.

With a static slice generator tool (CodeSurfer, [10]) we made static slices, too. The results are shown in Figure 13.

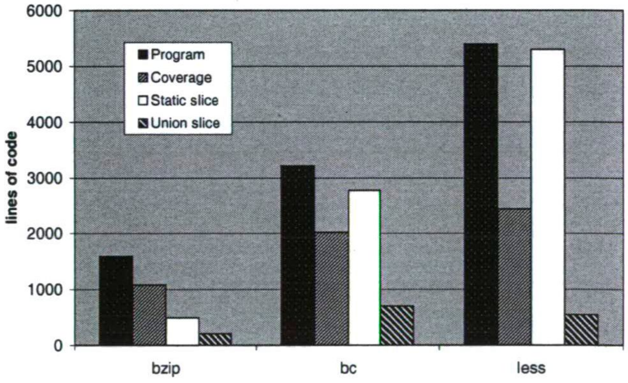

Figure 13: The average slice sizes

The first column shows the size of the executable code, the second the coverage, the third the average static slice (result of the CodeSurfer) and the last one is average of the so-called union slices generated by our dynamic slice generator tool. The union slice means the union of the all the generated slices (several executions + more results within one execution) to a certain statement.

6 Summary

Different program slicing methods are used for debugging, testing, reverse engineering and maintenance. Slicing algorithms can be categorized according to whether they use static slicing or dynamic slicing methods. In applications such as debugging, the computation of dynamic slices is more preferable as it can produce more precise results.

There are several methods for dynamic slicing available in the literature, but most of them make use of the internal representation of the program execution with dynamic dependencies called the Dynamic Dependence Graph (DDG). A big drawback of these methods is that the size of the DDGs is unbounded, because it includes a distinct vertex for each occurrence of a statement.

In [9] a new forward global method for computing dynamic slices of C programs was introduced. The algorithm determines the dynamic slices for any program instruction, in parallel with program execution, but it was worked out only for a simple program language.

To make the method usable for real programs, many problems had to be solved. This paper focused on two of them: the handling of pointers and unstructured jump statements. A method for handling the pointers, arrays, structures, `goto`, `break`, `continue` and `switch` statements of the `C` programming language was described, as well.

The main advantage of our algorithm is that it can be applied to real size `C` programs as its memory requirements are proportional to the number of different memory locations used by the program (which is in most cases much smaller than the size of the execution history—which is, actually, the absolute upper bound).

We have already developed a program where we implemented the forward dynamic slicing algorithm for C language programs. According to our preliminary trials, the memory requirements of the algorithm is indeed proportional to the number of different memory locations used by the program, which is much less than the size of the execution history.

References

[1] Agrawal, H., DeMillo, R. A., and Spafford, E. H. Debugging with dynamic slicing and backtracking. *Software—Practice And Experience*, 23(6):589-616, June 1993.

[2] Beck, J., and Eichmann, D. Program and Interface Slicing for Reverse Engineering. In *Proc. 15th Int. Conference on Software Engineering*, Baltimore, Maryland, 1993. IEEE Computer Society Press, 1993, 509-518.

[3] Fritzson, P., Shahmehri, N., Kamkar, M., and Gyimóthy, T. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems 1*, 4 (1992), 303-322.

[4] Gallagher, K. B., and Lyle, J. R. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering* 17, 8, 1991, 751-761.

[5] Gyimóthy, T., Beszédes, Á., and Forgács, I. An Efficient Relevant Slicing Method for Debugging. In *Proc. 7th European Software Engineering Conference (ESEC)*, Toulouse, France, Sept. 1999. LNCS 1687, pages 303-321.

[6] Korel, B., and Rilling, J. Application of dynamic slicing in program debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97)*, Linkoping, Sweden, May 1997.

[7] Rothermer, G., and Harrold, M. J. Selecting tests and identifying test coverage requirements for modified software. In *Proc. ISSTA'94* Seattle. 1994, 169-183

[8] Weiser M. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4, 1984, 352-357.

[9] Beszédes, Á., Gergely, T., Szabó, Zs. M., Csirik, J., and Gyimóthy T. *Dynamic Slicing Method for Maintenance of Large C Programs.* At the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001). Lisbon, Portugal, March 14-16, 2001.

[10] Homepage of `CodeSurfer`:
`http://www.grammatech.com/products/codesurfer/`