Graph-Less Dynamic Dependence-Based Dynamic Slicing Algorithms

Árpád Beszédes, Tamás Gergely and Tibor Gyimóthy
University of Szeged, Department of Software Engineering
Árpád tér 2., H-6720 Szeged, Hungary, +36 62 544145
{beszedes,gertom,gyimi}@inf.u-szeged.hu

Abstract

Using Dynamic Dependence Graphs is a well understood method for computing dynamic program slices. However, in its basic form, the DDG is inappropriate for practical implementation, so several alternative approaches have been proposed by researchers. In this paper, we elaborate on different methods in which the execution trace is processed and, using local definition-use information, the dependence chains are followed "on the fly" to construct the slices without actually building any graphs. Naturally, various additional data structures still need to be maintained, but these vary on the slicing scenario. Firstly, one may want to perform the slicing in a demand-driven fashion, or to compute many slices globally. Next, one may be interested either in backward or forward slices. And finally, the slices can be produced by traversing the trace either in a forward or in a backward direction. This totals eight possibilities, of which some give useful algorithms, while there are irrelevant combinations as well. In this work we investigate all of them, give the basic algorithms where appropriate and discuss on implementation experiences and perspectives.

Keywords

Program slicing, dynamic slicing algorithms, execution trace, program dependences.

1 Introduction

Over time, a number of program slicing methods [17, 19] have been elaborated. A significant part of the practical methods compute the slices based on various *dependences* (control- and data-) among the program elements (variables, instructions, addresses, predicates, etc.). The literature is elaborate about the details of static slicing methods. For example, the work by Horwitz *et al.* [8] served as the starting point for a number of subsequent implementations and enhancements, whose basis is the program dependence graph – PDG. This is a quite natural representation of the program under investigation since it captures the program elements of interest (instructions) and the dependences among them that will eventually determine the slices.

However, relatively few publications appeared that deal with the practical sides of dynamic slicing and provide detailed algorithms. Dynamic analysis of programs is an inherently hard problem because of several reasons, the most significant one being that a very large number of events may be generated by a program run. This induces huge amount of data to be processed, which is even more apparent with program slicing, since – in a general case – very fine-grained computations are needed for an accurate result. The fact that it is hard to find practically used dynamic slicing algorithms can mean that the published methods are not suitable for handling real size programs and executions.

The basic dynamic slicing methods use different concepts, proposed by researchers like Korel and Laski [12, 13], Agrawal *et al.* [1, 2] and Kamkar *et al.* [11]. One of the most common approaches to dynamic slicing is based on computing the dynamic dependences among the program elements, which is analogous to PDG-based static slicing algorithms. This way we follow the dynamically occurring data- and control dependences among the actions (instruction occurrences). Note, that using this approach the slices will not necessarily be executable subsets of the program, but in many applications of slicing this is not a requirement anyway. The traditional dynamic dependence-based method by Agrawal and Horgan [2] uses a graph representation called the Dynamic Dependence Graph – DDG that includes a distinct vertex for each occurrence of a statement (an action), and the edges correspond to the dynamically occurring dependences. Based on this graph, the computation of a dynamic slice means finding all reachable vertices starting from the slicing criterion. According to the terminology of Zhang *et al.* [23] a full preprocessing is done before the actual slicing.

Unfortunately, most of the basic dynamic slicing algorithms have difficulties with handling large inputs. For example, the size of the DDG graph is actually unbounded

as it is determined by the number of steps of the execution history produced by the program run. Consequently, researchers have begun searching for more effective methods, like Zhang and Gupta did in their work on compacting the DDG graph [21], or Mund and Mall with their static dependence graph-based approaches [15]. In previous work we have also elaborated new efficient dynamic slicing methods that are based on dynamic dependences, but which do not necessitate huge representations like the DDG graph. One of our most significant results is a backward slicing algorithm [5, 7] that computes all possible dynamic slices globally, with only one pass through the execution history. This method significantly differs from any other previously published algorithm, and proved to be suitable for real size programs and executions as well. We elaborated the details of the algorithm in different contexts: for C programs [5] and for Java [16] for example, and its usefulness has been demonstrated in several applications [4, 7].

Based on the original idea of the global slicing algorithm it turned out to be possible to construct similar graphless algorithms based on dynamic dependences, for example demand-driven methods for both forward and backward slices. In fact, we wanted to investigate all practical ways for computing the dynamic slices based on dynamic dependences but without requiring costly global preprocessing prior to slicing. Hence, in this paper we propose alternative methods that are based on the same dynamic dependences but instead of dynamic dependence graphs various data structures are maintained. These are different depending on the slicing scenario and therefore are specialized and more effective. The different slicing scenarios that we investigated are global vs. demand driven slicing and computing backward vs. forward slices. A good property of all of the presented algorithms is that they are able to compute the same dynamic slices as the original DDG-based method. It turned out that the slices can be produced by traversing the execution history either in a forward or in a backward way, and that some processing direction fits more to a slicing scenario than the other. This totals eight possibilities, of which some give useful algorithms, while there are irrelevant combinations as well. In this work we investigate and discuss all of them and give the basic algorithms where appropriate, along with some discussion about the application fields of each. Although some researchers mention similar algorithms, we are not aware of any other such comprehensive overview of the basic dependence based dynamic slicing algorithms. The contribution of this paper is thus in providing a comprehensive list of related algorithms that can be used in different slicing scenarios.

In the next section we give all the details about the algorithms. Section 3 deals with related work, and we close our paper with conclusions in Section 4.

2 Algorithms

All of the algorithms presented operate on two data sets: the execution history and a concise static representation of the program. Since the aim of this paper is to introduce the basic algorithms, for clarity we will limit our discussion to simple programs. Consequently, in the following the presented data structures and algorithms will not include modifications that are needed for a real programming language. The execution history (or trace) is a simple list of instruction occurrences (actions) that have been executed for a specific program input. The algorithms will process the trace in either forward or backward way to follow the dynamic dependences. All that we further need for achieving this is a simple *definition-use* relationship for all instructions of the program, based on which the actually realized dynamic dependence chains can be computed during processing the trace. Furthermore, by considering the predicates in the program (the branching instructions) as regular variables, both the data- and control-dependences can be handled uniformly. In other words, the dynamic slices will be computed "on the fly" during processing the trace taking into account the local definition-use information as the static representation of the program.

Prior to describing the actual slicing algorithms we will overview some notations, which will be used throughout the rest of the paper.

The execution history will be denoted by EH, and it will contain actions denoted by i^j, where i is the serial number of an instruction in the program, while j is the serial number of an execution step in the execution history. The total number of the steps executed in a given execution of the program is denoted by J. We will also use the following:

$$i(i^j) = i\,,\ j(i^j) = j\,,\ EH = \langle {i_1}^1, {i_2}^2, \ldots, {i_J}^J \rangle\,.$$

Furthermore, $EHI(j) = i(i^j)$ denotes the statement number i at the jth step.

We will use the notation $CB = (\mathbf{x}, i^j, V)$ for the backward, and $CF = (\mathbf{x}, i^j)$ for the forward slicing criterion, where $\mathbf{x}$ is a program input corresponding to a specific execution of the program, i^j is the action for which the dynamic slice needs to be computed and V is a subset of the used variables at the ith instruction.[1]

The static representation of the program needed by the slicing algorithms is called the *D/U program representation*. It captures local definition-use relationships between

[1] We assume the following: (1) all instructions define exactly one variable and use zero or more variables, (2) for backward slicing, if we allowed the defined variable to be part of V then all used ones would also be part of V implicitly, furthermore if all of the used variables are in V then the defined one will be there too, and (3) the forward dynamic slice is computed starting from the defined variable at i, therefore we do not need V in this case.

the variable occurrences within each instruction. For simplicity we will assume that each instruction defines one variable and uses zero or more variables. An instruction of a program has the following D/U representation:

$$i.\, d_i :\, U_i\,,$$

where i is an instruction serial number. We will use I to denote the total number of instructions in the program. The defined variable at the ith instruction is $d(i) = d_i$, while $U(i) = U_i$ is used to denote the use set that is utilized for computing the value of d_i.

A useful property of our approach is that using the same D/U representation we are able to capture not only the data dependences but the control dependences as well, which will significantly simplify the slicing algorithms. Namely, each d_i defined and $u_k \in U_i$ used variable ($i = 1, \ldots, I$) can have a special meaning that we call a *predicate variable*. Predicate variables are virtual ones that are not part of the program, but are generated for each predicate instruction in the program (these are the conditional branching instructions like *if* and *for*). Predicate instructions determine the control dependences among the instructions, so we can treat the corresponding predicate variables as regular variables that can serve both as the defined variable and as used ones. More precisely, if instruction i is a predicate instruction then a generated predicate variable p_i will be the defined variable at i, $d(i) = p_i$. Furthermore, for any instruction i' its use set $U(i')$ will be extended with a corresponding predicate variable for each predicate instruction on which i' is directly control dependent.

For the formalization of the dynamic slicing algorithms some more notations will be used. The *last definition* of a variable v will be denoted by $LD(v, j)$, which is a function returning the action at which v was defined last before the jth step in the execution history:

$$LD(v, j) = i'^{j'}\,,$$

where

$$j' < j \wedge d(i') = v \wedge \nexists j''\,(j' < j'' < j, d(EHI(j'')) = v)$$

Furthermore, we will also use some shorthand notations for the last defining step $LD(v) = j(LD(v, j))$, statement $LS(v) = i(LD(v, j))$ and action $LA(v) = LD(v, j)$ for an actual step j that is being processed during the execution of the slicing algorithm. Obviously, after processing a step i^j, $LD(d(i)) = j$, $LS(d(i)) = i$ and $LA(d(i)) = i^j$ will hold for each subsequent action until $d(i)$ is defined next time.

2.1 Overview

Given the same static and dynamic representations, the D/U form and the execution history we can categorize the dynamic slicing algorithms according to the following three kinds of properties.

- **Slice direction.** This classification corresponds to the two fundamental slice kinds. Namely, if we associate a slicing criterion with a set of program locations whose earlier execution affected the value computed at the criterion, we speak of a *backward slice*. On the contrary, a *forward slice* is a set of program locations whose later execution depends on the values computed at the slicing criterion.

- **Global or Demand-driven.** The traditional approach is to compute one slice at a time, based on a given criterion. In this case one generally starts at the program point of the criterion and collects the instructions to be incorporated into the slice by traversing the dependences backward or forward, depending on the slice direction. This is what we call *demand-driven slicing*.[2] However, there is an opportunity to compute multiple slices for different criteria during a single pass over the trace, if there is need for such a set of slices. This is what we call *global slicing*, for which algorithms may be constructed for both slice direction types.

- **Processing direction.** Finally, given a trace we may process it in both directions, depending on the kind of slice needed and on which method is more practical. Note that *forward processing* of the trace seems to be the natural one (and the only feasible in some applications), however traversing the trace *backwards* can also be applied in some situations.

From the above classification types it naturally follows to investigate all of the possible 8 combinations. As we will see, some of them lead to practical algorithms, while other are virtually unfeasible. In Table 1 we list all possibilities, giving each method an identification number from zero to seven. We will use these numbers instead of detailed descriptions in the text that follows.

The last column of the table is used for a preliminary classification of the algorithm according to its usefulness, which also determines the remaining subsections of this section. Namely, we will first discuss the two practical demand-driven algorithms in Section 2.2, then we overview the two practical global algorithms in Section 2.3, and finally Section 2.4 deals with the remaining two global algorithms. The latter may be implemented, but their usability is questionable since it requires storing all slices in the memory until the whole trace has been processed, so we refer to them as parallel algorithms. This gives the total of six algorithms. The remaining two types of demand-driven algorithms are impractical to implement, since to compute a demand driven slice in a reverse direction would virtually mean performing global dependence tracking.

[2] The term 'demand driven' is used with a different meaning in static slicing, where it is referred to constructing the static graph representations of the program on demand based on the slicing request [3].

No.	Global/Demand-driven	Slice direction	Processing direction	Usefulness
0	Demand-driven	Backward	Backward	Practical
1	Demand-driven	Backward	Forward	Unfeasible
2	Demand-driven	Forward	Backward	Unfeasible
3	Demand-driven	Forward	Forward	Practical
4	Global	Backward	Backward	Parallel
5	Global	Backward	Forward	Practical
6	Global	Forward	Backward	Practical
7	Global	Forward	Forward	Parallel

Table 1. Overview of dynamic slicing algorithms

Due to space constraints, unfortunately we are unable to provide examples of all the algorithms' workings. However, we will illustrate the first algorithm on an example program, which can be seen in Figure 1, along with its D/U representation. This example has interesting execution histories for inputs $\langle \mathtt{a} = 0 \rangle$, $\langle \mathtt{a} = 1 \rangle$ and $\langle \mathtt{a} = 2 \ldots \rangle$. For illustration we will use $\mathtt{x} = \langle \mathtt{a} = 1 \rangle$, which produces the following: $EH = \langle 1^1, 2^2, 3^3, 4^4, 5^5, 6^6, 7^7, 4^8, 8^9 \rangle$.

i		$d : U$
1	`read(a)`	$a : \emptyset$
2	`y=0`	$y : \emptyset$
3	`x=1`	$x : \emptyset$
4	`while(a>0)`	$p : \{a\}$
5	`  y=x`	$y : \{x, p\}$
6	`  x=2`	$x : \{p\}$
7	`  a=a-1`	$a : \{a, p\}$
8	`z=y`	$z : \{y\}$

Figure 1. The example program

2.2 Demand-Driven Algorithms

Computing a dynamic slice in a demand-driven fashion means that given an execution of the program and a dynamic slicing criterion, a single dynamic slice is produced. This corresponds to determining one dynamic slice from a complete DDG. Here, a preprocessing step is performed in which the (complete) dynamic dependence graph is produced, after which each slicing request means traversing this graph starting from the slicing criterion to find the reachable parts.

In our case however, a similar method may be constructed in a straightforward way but without producing dependence graphs explicitly. We traverse the execution trace starting with the action of the dynamic slicing criterion, and follow the dynamic dependences with the help of the D/U representation going backward towards the first executed instruction or forward towards the end of the trace, depending on the slice direction. This allows us to construct the two demand-driven dynamic slicing algorithms as follows.

2.2.1 Backward Slice – *Algorithm 0*

The demand driven algorithm for backward slices processes the execution history starting with the action of the criterion and traces back the dependences towards the very first action. An obvious drawback of this method is that the execution history needs to be processed in reverse, which implies that it needs to be stored first completely. Our global method for backward slices works with forward processing of the trace, so it may be much more feasible in some applications (see Section 2.3).

The algorithm scans the dynamic dependences and keeps those actions not yet processed in a worklist. When it removes an action from the worklist it investigates all variables from the use set of the removed action and extends the worklist with the last defining action of those variables before the actual execution step. When all dependences have been processed and the worklist becomes empty the algorithm terminates by providing the slice with the instructions visited during the run. This operation is formalized in the algorithm in Figure 2. A similar algorithm was sketched by Korel as well [12].

Note that the last defining action in algorithm line 8 is not directly accessible as with the forward processing of the execution history. Therefore for the efficient functioning of the algorithm the trace needs to be stored in a special form that groups the actions according to the variables defined in them. This is represented by the so-called EHT (execution history) table, whose rows are constituted of the actions with the corresponding defined variables. This is needed because in every iteration of the algorithm an arbitrary definition action corresponding to the variables could be needed, which means that there is a need for searching among the execution steps backwards. So the value of $LD(u, l)$ will be attained by first selecting the row corresponding to u in the EHT table and then finding the appropriate execution step $l' < l$ in that row (this can be implemented efficiently since the rows are ordered by execution step number).

Also note, that removing the biggest action from the worklist at line 5 is not required; any action could be removed next, but it can help for the efficient implementation of the EHT table. Namely, all remaining actions in a row

program Algorithm-0(P, CB)
input: P : a program
$CB = (\mathbf{x}, i^j, V)$: dynamic slicing criterion
(assume $V = U(i)$)
output: S : dynamic slice of P for CB

begin
1 Read and store EH up to i^j
2 $S := \emptyset$
3 $worklist \leftarrow i^j$
4 **while** $worklist \neq \emptyset$
5 k^l := remove element with biggest l from $worklist$
6 **if** $l \neq j$ **then** $S := S \cup \{k\}$
7 **for** $\forall u \in U(k)$
8 $worklist \leftarrow LD(u, l)$
endfor
endwhile
9 Output S as the backward dynamic slice
for criterion $(\mathbf{x}, i^j, U(i))$
end

Figure 2. Demand driven algorithm for backward slices

that are beyond the returned one may be discarded, since the removed actions will be monotonically decreasing.

The number of iterations of the algorithm varies, it is minimum the number of instructions in the slice computed, but in the worst case it can be as much as the length of the execution. However, in the average case it will be correlated with the size of the slice. As for the space requirements, not counting the storage of the EHT table (it can be kept on the disk), it is not significant since only the worklist needs to be maintained in the memory.

We will illustrate the working of the algorithm on our example. The state of the worklist and its operations can be followed for each iteration in the table in Figure 3.

iteration	S	$worklist$	removed	added
0	$\emptyset$	$\{8^9\}$	—	8^9
1	$\emptyset$	$\{5^5\}$	8^9	5^5
2	$\{5\}$	$\{3^3, 4^4\}$	5^5	$4^4, 3^3$
3	$\{4, 5\}$	$\{1^1, 3^3\}$	4^4	1^1
4	$\{3, 4, 5\}$	$\{1^1\}$	3^3	—
5	$\{1, 3, 4, 5\}$	$\emptyset$	1^1	—

Figure 3. Example run of the demand driven backward slicing algorithm

2.2.2 Forward Slice – *Algorithm 3*

Computing forward dynamic slices starting from the slicing criterion means traversing the execution trace in a natural way, that is in a forward fashion. The algorithm is given in Figure 4.

program Algorithm-3(P, CF)
input: P : a program
$CF = (\mathbf{x}, i^j)$: dynamic slicing criterion
output: S : dynamic slice of P for CF

begin
1 Read EH
2 $mark(d(i))$
3 $S := \emptyset$
4 $k := j$
5 **while** $\exists$ marked variables **and** $k < J$
6 $k := k + 1$
7 $l := EHI(k)$
8 **if** $\exists\big(u \in U(l)$ **and** $marked(u)\big)$
9 $mark(d(l))$
10 $S := S \cup \{l\}$
else
11 $unmark(d(l))$
endif
endwhile
12 Output S as the forward dynamic slice
for criterion $(\mathbf{x}, i^j)$
end

Figure 4. Demand driven algorithm for forward slices

The basic idea of the algorithm is to collect all forward dependences of the defined variable at the criterion by marking variables that carry forward the dynamic dependences. If a variable is marked at a given point of the algorithm execution, it means that it is a "live" variable which was defined using the contribution of another live variable. So first the defined variable is set to be live (line 2), after which a *while* loop starts from the execution step of the criterion j. It terminates if there are no further live variables to process or if we reach the end of the trace. If in the next executed instruction a live variable is used (algorithm line 8), the corresponding defined variable is also set as live (line 9). At the same time the corresponding instruction serial number is made part of the resulting slice set S in line 10. Note that statement 11 is required to kill any variables that were potentially live but that are redefined without using any live variables at that point.

A possible improvement to the algorithm is if we increment k in line 6 not step-by-step but by jumping towards the first next position where some of the marked variables is de-

fined or used. In this case in line 11 the defined variable is definitely live, but the implementation of this approach may not be beneficial.

2.3 Practical Global Algorithms

Depending on the structure of the dependences, a demand driven dynamic slicing algorithm may need to process a significant part of the execution history individually for each slicing request. A demand driven algorithm will follow only the dependences belonging to the slicing criterion, however in the meantime many intermediate actions of the execution history need to be passed by. Furthermore, in a number of applications more than one slice may be needed at a time for a given execution of the program. This leads to an idea to compute more dynamic slices during only one traversal through the execution history. Naturally, this will mean more simultaneous computations, but above a certain number of distinct slices a more global algorithm will be more beneficial than executing the demand driven one multiple times. It is possible to compute many dynamic slices by executing the demand driven methods in parallel: traversing the execution history in a forward way for forward slices and in a backward way for backward slices. However, this approach is not very practical since the data structures (and the slices) for *all dynamic criteria* need to be maintained throughout the whole execution history. The algorithms incorporating this kind operation are described in Section 2.4.

Fortunately, it is possible to construct such global algorithms that are more practical in which not the whole dynamic slices need to be maintained during the execution of the algorithms but only the actual dependence sets belonging to the variables of the program. These dependence sets contain statement numbers providing the actual dependences of the given variables at the given point of execution. We derive these dependence sets based on the D/U information and maintain them for each execution step. Thus we are able to compute the dynamic slices for all possible dynamic criteria based on the actual values of these sets only.

An interesting duality in this approach is that the mentioned dependence sets for computing *backward* slices can be acquired if we process the trace in a *forward* way, and for the forward slices we need to traverse the trace in reverse direction. (Obviously, the backward historical dependence data for a given execution point is accessible only by a "natural lapse of time," while the complete forward dependences from a point of execution ("future" data) are visible by examining the execution history in a reversed way.)

In this section we will describe the two basic dynamic slicing algorithms, which are able to produce *all* dynamic slices for a given execution of a program: the global backward slicing method and the global forward slicing method.

2.3.1 Backward Slice – *Algorithm 5*

The algorithm in Figure 5 is our method for producing backward slices globally. Since this approach requires a forward processing of the execution history it is one of the most practically usable ones among all presented in this article. It allows instant processing of the trace as it is produced by the program executor. It has been presented several times in some of our past publications with different applications [4, 5, 7], and implemented in different contexts: for C programs [5] and for Java [16], for example.

program Algorithm-5$(P, \mathbf{x})$
input: P : a program
$\mathbf{x}$: a program input
output: backward slices for all $(\mathbf{x}, i^j, V_i)$ criteria $(j = 1 \ldots J, V_i = U(i))$

begin
1 Read EH
2 **for** $j = 1$ **to** J
3 $i := EHI(j)$
4 $DynDep(d(i)) :=$ $\bigcup_{u_k \in U(i)} \big(DynDep(u_k) \cup \{LS(u_k)\}\big)$
5 $LS(d(i)) := i$
6 Output $DynDep(d(i))$ as the backward dynamic slice for criterion $(\mathbf{x}, i^j, U(i))$
endfor
end

Figure 5. Global algorithm for backward slices

The algorithm starts processing the trace with the firstly executed instruction, and at each step it computes the dependence set corresponding to the defined variable at the actual instruction, which holds the depending instruction numbers ($DynDep(d(i))$). For this it uses the most recently computed dependence sets of the used variables at the instruction ($DynDep(u_k)$) and their last defining instruction ($LS(u_k)$). This way all dynamic slices corresponding to the defined variables at all instruction occurrences are attained and provided at the output (only the actually effective sets are stored in the memory). It is clear, that the average computational complexity is determined by the length of the execution history and the average size of the dependence sets, the latter being in correspondence with the average slice size. The space requirements are determined by the number of the dependence sets (the number of all defined variables) and their sizes (practically the average slice size).

2.3.2 Forward Slice – *Algorithm 6*

The algorithm in Figure 6 is our method for producing forward slices globally. This approach requires a backward

processing of the execution history. Backward processing of the trace is sometimes not as straightforward as processing it from its beginning towards its end, but otherwise this algorithm is as usable for forward slice computation as the previous one is for computing backward slices.

program Algorithm-6($P, \mathbf{x}$)
input: P : a program
$\mathbf{x}$: a program input
output: forward slices for all $(\mathbf{x}, i^j)$ criteria ($j = 1 \ldots J$)

begin
1 Read and store EH
2 **for** $j = J$ **downto** 1
3 $i := EHI(j)$
4 Output $LiveAt(d(i))$ as the forward dynamic slice for criterion $(\mathbf{x}, i^j)$
5 **for** $u_k \in U(i)$
6 $LiveAt(u_k) := LiveAt(u_k) \cup LiveAt(d(i)) \cup \{i\}$
endfor
7 **if** $d(i) \notin U(i)$
8 $LiveAt(d(i)) := \emptyset$
endif
endfor
end

Figure 6. Global algorithm for forward slices

The algorithm starts processing the trace with the last executed instruction. For each variable v it maintains a "live" set ($LiveAt$), which holds statement numbers of processed trace elements with defined variables dependent on the latest previous (not yet processed) definition of v. Thus at the beginning of each step, the $LiveAt$ set of the variable defined at the actual instruction ($d(i)$) contains the forward slice of this variable. The $LiveAt$ set of the variables used in the actual instruction ($U(i)$) must be extended with the statement number of the actual action (i), and those statements which are dependent on the defined variable $d(i)$ because all actions dependent on $d(i)$ are dependent on all elements of $U(i)$. If $d(i)$ is not dependent on itself the $LiveAt$ set of it must be emptied (line 8) because based on the processed actions, no trace elements are dependent on the previous definition of this variable.

The characteristics of this algorithm are as of the previous one. All dynamic forward slices corresponding to the defined variables at all instruction occurrences are attained and provided at the output. The average computational complexity is determined by the length of the execution history and the average size of the "live" sets, and the space requirements are determined by the number of the "live" sets and their sizes.

2.4 Parallel Global Algorithms

The trace processing directions of the global algorithms presented in the previous section are the opposite of the directions of the slices they compute. They have the advantage of computing the final slice of a criterion as soon as its action has been processed in the execution history. On the other hand, the two demand-driven algorithms use the direction of the slice for processing the trace.

In this section we present two more global algorithms that use the slice direction for processing the trace and compute the slices for all criteria. We called these *parallel global* algorithms because they compute all slices in parallel – virtually having many parallel demand-driven algorithms –, and potentially no slices are finished before all trace elements are processed. However, they still have the advantage over computing all the slices with the demand-driven algorithms: the dependences arising from a specific action are computed only once.

2.4.1 Forward Slice – *Algorithm 7*

Figure 7 shows our parallel algorithm for computing forward slices. Although it consumes more memory than the global forward slice algorithm and produces final slices after processing all actions in the trace, this algorithm is a usable alternative of the other global one because of the forward processing of the trace.

The first part of the algorithm is similar in its structure to the global algorithm for backward slices (number 5), the main difference being that in this case we need to track actions in the $DynDep$ sets instead of only instruction numbers. In the next step of each iteration the forward slice of all actions that the defined variable depends on must be extended with the line number of the actually processed action ($EHI(j)$). Finally, the slices of all criteria are written.

The computational complexity of the algorithm is determined by the trace length and the average size of the dependence sets. The space requirements are determined by the number of trace elements and the average slice sizes, and the number of dependence sets and their average size. Tracking actions in the dependence sets can mean a significant overhead, which can be overcome by working with only one occurrence of each statement (and using the original $DynDep$ sets), this way accumulating the dependences for all related actions. Naturally, this will have the tradeoff of loosing the precision of the slices.

Potentially, no slices can be treated as done before the last trace element is processed. Although we could determine if a slice cannot be extended anymore (all contributing variables become dead), it would require to maintain a dual dependence set structure or increase the computational complexity.

```
program  Algorithm-7(P, x)
input:   P : a program
         x : a program input
output:  forward slices for all (x, i^j) criteria
         (j = 1 ... J)
begin
1   Read EH
2   for j = 1 to J
3       i := EHI(j)
4       DynDep(d(i)) :=
            ⋃_{u_k ∈ U(i)} (DynDep(u_k) ∪ {LA(u_k)})
5       LA(d(i)) := i^j
6       for a_k ∈ DynDep(d(i))
7           S(a_k) := S(a_k) ∪ {i}
        endfor
    endfor
8   for ∀ i^j ∈ EH
9       Output S(i^j) as the forward dynamic slice
            for criterion (x, i^j)
    endfor
end
```

Figure 7. Forward algorithm for forward slices

2.4.2 Backward Slice – *Algorithm 4*

Our algorithm for parallel backward slice computation can be seen in Figure 8. Due to the backward processing of the trace and greater space requirements, it is not a real alternative to the global backward slicing algorithm, but we will present it for the sake of completeness.

Since the logic of dependence tracking is the same, the core part of the algorithm is similar to the practical global forward slicing method (algorithm 6). Here as well, the main difference is that we track actions in the $LiveAt$ dependence sets instead of instructions. The slices are maintained at the beginning of each step: namely, the slices of actions dependent on the actually defined variable $d(i)$ are extended with the actual program line i. $LiveAt(d(i))$ in line 4 contains these actions. With this algorithm as well, the slices of all criteria are written after the processing of the execution history has been completed.

The computational complexity of the algorithm is determined by the trace length and the average size of the dependence and "live" sets, while the space requirements are determined by the number of trace elements and the average slice sizes, and the number of "live" sets (the number of all defined variables) and their average size. The same overhead with storing actions in the dependence sets applies here as with the other parallel algorithm mentioned above.

As with the previous algorithm, potentially no slices can be written before the last (first) trace element is processed. With more computation or space usage it could be determined if a slice will not change in the later iterations.

```
program  Algorithm-4(P, x)
input:   P : a program
         x : a program input
output:  backward slices for all (x, i^j, V_i) criteria
         (j = 1 ... J, V_i = U(i))
begin
1   Read and store EH
2   for j = J downto 1
3       i := EHI(j)
4       for a_k ∈ LiveAt(d(i))
5           S(a_k) := S(a_k) ∪ {i}
        endfor
6       for u_k ∈ U(i)
7           LiveAt(u_k) :=
                LiveAt(u_k) ∪ LiveAt(d(i)) ∪ {i^j}
        endfor
8       if d(i) ∉ U(i)
9           LiveAt(d(i)) := ∅
        endif
    endfor
10  for ∀ i^j ∈ EH
11      Output S(i^j) as the backward dynamic slice
            for criterion (x, i^j, U(i))
    endfor
end
```

Figure 8. Backward algorithm for backward slices

3 Related work

Our methods significantly differ from existing dynamic slicing algorithms. The novelty lies in handling control- and data dependences uniformly, and in providing a comprehensive list of related algorithms that can be used in different slicing scenarios. As already mentioned earlier, our methods for dynamic slicing are based on the same dynamic dependences as some other dynamic slicing algorithms employ. Such is the Dynamic Dependence Graph-based method by Agrawal and Horgan [2], which requires a significant amount of preprocessing (building the graph) prior to any (demand-driven) slicing request. However, the complete DDG graph is impractical in most real life situations because of its significant costs, even with its optimized version [17]. Based on this method several enhanced algorithms were published later [11, 21]. For example, Zhang and Gupta propose a reduced version of the DDG with which an improvement of about one order of magnitude can

be achieved in terms of the graph size. Zhang *et al.* provide some other possibilities as well for reducing the costs of this method [23]. Mund and Mall tried to improve the efficiency of dynamic slicing using *static* dependence graphs as intermediate representation and maintaining various additional data structures during processing the execution history [15].

The first dynamic slicing algorithm was given by Korel and Laski that produced executable slices [12]. As it is known, executable slices are generally significantly larger than those that deal with the dependences only (according to Venkatesh's measurements the ratio is about 2–3 times [18]), however in our work we do not require this property. The same authors published another dynamic slicing method that is based on dataflow equations [13], while Korel and Yalamanchili give a forward computation method [14]. The basis for these methods are the so-called removable blocks, which are selected for inclusion into the slice by the algorithm based on the execution of the program. Apart from this approach Korel sketched a demand driven method based on dynamic dependences similar to ours [12].

Concrete evidence of dynamic slicing implementations that are usable in real life scenarios is very hard to find. Some of the work in this field are Agrawal's [1], Kamkar's [9] and Venkatesh's results [18]. Zhang and Gupta also reported implementation of different efficient algorithms, which contain some very interesting enhancements to the existing basic methods [22, 23], some of which can be well adapted to our algorithms. For example, their usage of reduced ordered binary decision diagrams is an interesting enhancement of our forward computation method [22].

Excellent surveys of different slicing methods, including dynamic slicing have been published by several authors: Tip [17], De Lucia [6], Kamkar [10] and Xu *et al.* [20].

4 Conclusions

In this paper we presented six algorithms for computing backward and forward dynamic slices. All of them are based on computing the dynamic dependences by traversing the execution history, and using a simple static representation of the program containing local definition-use information, which uniformly incorporates both data and control dependences. Although some researchers mention similar algorithms, we are not aware of any other, such a comprehensive overview of the basic dependence-based dynamic slicing algorithms.

As for our experiences, we have already implemented three of the presented algorithms. Algorithms 5 and 0 were implemented for the C language [5], for which we had to solve some special problems. To handle pointers, arrays and other memory references correctly, we used memory locations instead of variables, keeping information of the assignment of variables and memory locations. After converting all memory accesses to pointer operations, this change to memory locations simplified all data flow problems. The problem of unstructured control flow was solved by applying the traditional postdominance-based control dependence method. The trace generation for C was done by source code instrumentation. For the Java language we have a ready implementation of algorithms 5 and 7 [16]. This implementation works on Java byte code, but the result can be converted to the source. Slicing Java programs arises some special problems as well like multi-threading and reflection. To handle these correctly our slicer simulates some necessary tasks of the virtual machine while processing the trace, such as context switches, the stack, etc. To produce the trace an instrumented Java Virtual Machine was used.

Naturally, the main question is in which contexts is most useful each of the presented algorithms. Here we sketch our views on the applicability of the algorithms, however a more elaborate investigation of their usefulness remains for future work.

If more than one slice is sought for an execution of the program the choice between a global and a demand driven approach may be generally based on comparing the costs of computing all possible slices with the global method and executing the demand driven algorithm several times. The threshold above which performing global slicing is more beneficial is hard to determine, since it depends on many aspects, like the layout of the execution history, the slice criteria of interest and the arising dynamic dependences. With our implementation of the global and demand driven backward slicing methods (algorithms 5 and 0) for C, we performed some measurements on medium size test programs. We recorded the iteration numbers and other complexity factors of the algorithms and the execution times as well. The major complexity factor of the global method is the length of the execution history (J) and the set operations at each step, while of the demand driven one is the iteration number. The latter is at most J, however it is determined by the number of arising dynamic dependences, which is, according to our measurements, about a magnitude smaller than J. This allows us to give a rough estimation that if at least one or two dozens of slices need to be computed, a global method may be more beneficial.

Global slicing methods are very useful for applications in which more (all) slices are required. An example is the computation of union slices [4], which is a very useful technique for software maintenance-related problems. Global backward slicing can be used for, among others, program comprehension, debugging and decomposition slicing, while global forward slicing is useful in regression testing and impact analysis. For debugging and program comprehension, probably a highly optimized demand driven approach or a method with limited preprocessing would be optimal.

We presented four global slicing algorithms, of which

two are practically implementable (algorithms 5 and 6), since only a limited amount of intermediate data needs to be maintained during the execution of the algorithms. The backward slicing method is indeed very practical since it processes the execution history in a forward way; we have already utilized our implementation in a number of applications. However, the main drawback with the forward slicing algorithm (number 6) is that it needs to process the execution history in a backward fashion, which may be impractical in many real life situations. In this case we propose instead the parallel forward slicing algorithm (number 7) that may be implemented with a small extension to the original global backward method (number 5) – we have already done so. The overhead of keeping many dynamic dependence sets in the memory can be overcome by loosening the precision and working with only one occurrence of each statement and accumulating the dependences for all related actions.

Our main directions for future work will be a more detailed elaboration on the complexities of the presented algorithms (giving O-notations) and their prototype implementation for the empirical investigation of the costs in various slicing situations. The possibilities for improvement will be addressed as well in order to find the real application fields for each of the algorithms.

Acknowledgements

This work was supported by The Péter Pázmány Program of the Hungarian National Office of Research and Technology (no. RET-07/2005).

References

[1] H. Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1992.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, number 6 in SIGPLAN Notices, pages 246–256, White Plains, New York, June 1990.

[3] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, Berlin, Germany, Mar. 1996.

[4] Á. Beszédes, Cs. Faragó, Zs. M. Szabó, J. Csirik, and T. Gyimóthy. Union slices for program maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–21. Oct. 2002.

[5] Á. Beszédes, T. Gergely, Zs. M. Szabó, J. Csirik, and T. Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. Mar. 2001.

[6] A. De Lucia. Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 142–149, Nov. 2001.

[7] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proceedings of ESEC/FSE'99*, number 1687 in Lecture Notes in Computer Science, pages 303–321. Springer-Verlag, Sept. 1999.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[9] M. Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linköping University, 1993.

[10] M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197–214, Dec. 1995.

[11] M. Kamkar, N. Shahmehri, and P. Fritzson. Interprocedural dynamic slicing. In *Proceedings of PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 1992.

[12] B. Korel and J. W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[13] B. Korel and J. W. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[14] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, Washington, Aug. 1994.

[15] G. B. Mund and R. Mall. An efficient interprocedural dynamic slicing method. *The Journal of Systems and Software*, 79(6):791–806, 2006.

[16] A. Szegedi and T. Gyimóthy. Dynamic slicing of Java bytecode programs. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 35–44. Sept. 2005.

[17] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[18] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, Mar. 1995.

[19] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[20] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[21] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, Washington, D. C., June 2004.

[22] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 502–511, Edinburgh, United Kingdom, May 2004.

[23] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 27(4):631–661, July 2005.