# Dynamic Program Slicing

*Hiralal Agrawal*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-2004

*Joseph R. Horgan*
Bell Communications Research
Morristown, NJ 07960-1910

## Abstract

Program slices are useful in debugging, testing, maintenance, and understanding of programs. The conventional notion of a program slice, the *static slice*, is the set of all statements that *might* affect the value of a given variable occurrence. In this paper, we investigate the concept of the *dynamic slice* consisting of all statements that *actually* affect the value of a variable occurrence for a given program input. The sensitivity of dynamic slicing to particular program inputs makes it more useful in program debugging and testing than static slicing. Several approaches for computing dynamic slices are examined. The notion of a Dynamic Dependence Graph and its use in computing dynamic slices is discussed. The Dynamic Dependence Graph may be unbounded in length; therefore, we introduce the economical concept of a Reduced Dynamic Dependence Graph, which is proportional in size to the number of dynamic slices arising during the program execution.

## 1 Introduction

Finding all statements in a program that directly or indirectly affect the value of a variable occurrence is referred to as Program Slicing [Wei84]. The statements selected constitute a *slice* of the program with respect to the variable occurrence. A slice has a simple meaning: it should evaluate the variable occurrence identically to the original program for *all* testcases.

Uses of program slicing have been suggested in many applications, e.g., program verification, testing, maintenance, automatic parallelization of program execution, automatic integration of program versions, etc. (see, e.g., [Wei84, HPR89]). In this paper we are primarily concerned with its use in program debugging [Wei82]. Often during debugging the value of a variable, *var*, at some program statement, $S$, is observed to be incorrect. Program slicing with respect to *var* and $S$ gives that relevant subset of the program where one should look for the possible cause of the error. But the above notion of program slicing does not make any use of the particular inputs that revealed the error. It is concerned with finding all statements that *could* influence the value of the variable occurrence for *any* inputs, not all statements that *did* affect its value for the *current* inputs. Unfortunately, the size of a slice so defined may approach that of the original program, and the usefulness of a slice in debugging tends to diminish as the size of the slice increases. Therefore, in this paper we examine a narrower notion of "slice," consisting only of statements that influence the value of a variable occurrence for specific program inputs.[1] We refer to this problem as *Dynamic Program Slicing* to distinguish it from the original problem of *Static Program Slicing*.

Conceptually a program may be thought of as a collection of *threads*, each computing a value of a program variable. Several threads may compute values of the same variable. Portions of these threads may overlap one-another. The more complex the control structure of the program, the more complex the intermingling of these threads. Static program slicing isolates all possible threads computing a particular variable. Dynamic slicing, on the other hand, iso-

Part of the work described here was done while the first author worked at Bell Communications Research, Morristown, New Jersey, during the summer of 1989. Other support was provided by a grant from the Purdue University/University of Florida Software Engineering Research Center, and by the National Science Foundation grant 8910306-CCR.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1990 ACM 0-89791-364-7/90/0006/0246 $1.50

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

[1] A slice with respect to a set of variables may be obtained by taking the union of slices with respect to individual variables in the set.

lates the unique thread computing the variable for the given inputs.

During debugging programmers generally analyze the program behavior under the test-case that revealed the error, not under any generic test-case. Consider, for example, the following scenario: A friend while using a program discovers an error. He finds that the value of a variable printed by a statement in the program is incorrect. After spending some time trying to find the cause without luck, he comes to you for help. Probably the first thing you would request from him is the test-case that revealed the bug. If he only tells you the variable with the incorrect value and the statement where the erroneous value is observed, and doesn't disclose the particular inputs that triggered the error, your debugging task would clearly be much more difficult. This suggests that while debugging a program we probably try to find the *dynamic* slice of the program in our minds. The concrete test-case that exercises the bug helps us focus our attention to the "cross-section" of the program that contains the bug.[2] This simple observation also highlights the value of automatically determining dynamic program slices. The distinction between static and dynamic slicing and the advantages of the latter over the former are further illustrated in Section 3.

In this paper we sketch several approaches to computing dynamic program slices. A more detailed discussion with precise algorithmic definitions of these approaches may be found in [AH89]. In Section 2 we briefly review the program representation called the Program Dependence Graph and the static slicing algorithm. Then we present two simple extensions to the static slicing algorithm to compute dynamic slices in Sections 3.1 and 3.2. But these algorithms may compute overlarge slices: they may include extra statements in the dynamic slice that shouldn't be there. In Section 3.3 we present a data-structure called the Dynamic Dependence Graph and an algorithm that uses it to compute accurate dynamic slices. Size of a Dynamic Dependence Graph depends on the length of the program execution, and thus, in general, it is unbounded. In Section 3.4, we introduce a mechanism to construct what we call a Reduced Dynamic Dependence Graph which requires limited space that is proportional to the number of distinct dynamic slices arising during the current program execution, not to the length of the execution. The four approaches to dynamic slicing presented here span a range of solutions with varying space-time-accuracy trade-offs.

```
        begin
S1:         read(X);
S2:         if (X < 0)
            then
S3:             Y := f1(X);
S4:             Z := g1(X);
            else
S5:             if (X = 0)
                then
S6:                 Y := f2(X);
S7:                 Z := g2(X);
                else
S8:                 Y := f3(X);
S9:                 Z := g3(X);
                end_if;
            end_if;
S10:        write(Y);
S11:        write(Z);
        end.
```

Figure 1: Example Program 1

## 2 Program Dependence Graph and Static Slicing

The program dependence graph of a program [FOW87, OO84, HRB88] has one node for each simple statement (assignment, read, write etc., as opposed to compound-statements like if-then-else, while-do etc.) and one node for each control predicate expression (the condition expression in if-then-else, while-do etc.). It has two types of directed edges—data-dependence edges and control-dependence edges.[3] A data-dependence edge from vertex $v_i$ to vertex $v_j$ implies that the computation performed at vertex $v_i$ directly depends on the value computed at vertex $v_j$.[4] Or more precisely, it means that the computation at vertex $v_i$ uses a variable, *var*, that is defined at vertex $v_j$, and there is an execution path from $v_j$ to $v_i$ along which *var* is never redefined. A control-dependence

[2] When we say the slice contains the bug, we do not necessarily mean that the bug is textually contained in the slice; the bug could correspond to the absence of something from the slice—a missing if statement, a statement outside the slice that should have been inside it, etc. We can discover that something is missing from the slice only after we have found the slice. In this sense, the bug still "lies in the slice."

[3] In other applications like vectorizing compilers program dependence graphs may include other types of edges besides data and control dependence, e.g., anti-dependence, output-dependence etc., but for the purposes of program slicing, the former two suffice.

[4] At other places in the literature, particularly that related to vectorizing compilers, e.g., [KKL+81, FOW87], direction of edges in Data Dependence Graphs is reversed, but for the purposes of program slicing our definition is more suitable.

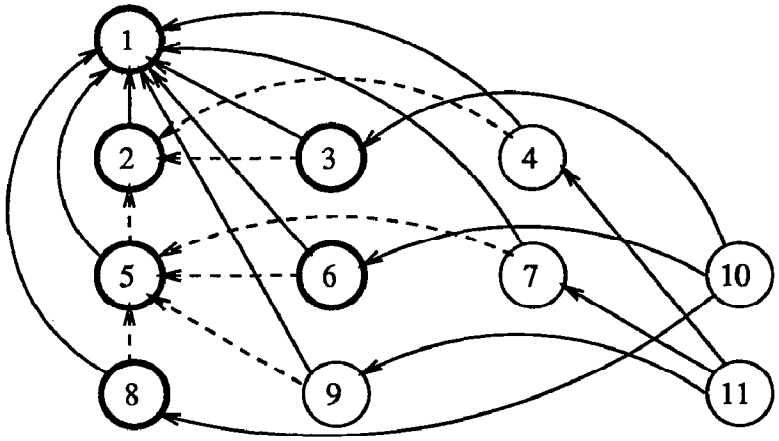


Figure 2: Program Dependence Graph of the Program in Figure 1. The solid edges denote data dependencies and the dashed edges denote control dependencies. Nodes in bold denote the Static Slice with respect to variable Y at statement 10 in the program.

edge from $v_i$ to $v_j$ means that node $v_i$ may or may not be executed depending on the boolean outcome of the predicate expression at node $v_j$.[5] Consider, for example, the program in Figure 1. Symbols $f_i$ and $g_i$ in the assignment statements are used to denote some unspecified side-effect-free functions with which we are not presently concerned. Figure 2 shows the Program Dependence Graph of this program. Solid edges denote data dependencies and dashed edges denote control dependencies. We do not distinguish between the two types of edges from now on; both are drawn as solid edges.

The static slice of a program with respect to a variable, *var*, at a node, *n*, consists of all nodes whose execution could possibly *affect* the value of *var* at *n*. The static slice can be easily constructed by finding all reaching definitions of *var* at node *n* [ASU86], and traversing the Program Dependence Graph beginning at these nodes. The nodes visited during the traversal constitute the desired slice [OO84, HRB88]. For example, to find the static slice of the program in Figure 1 with respect to variable Y at statement 10, we first find all reaching definitions of Y at node 10. These are nodes 3, 6, and 8. Then we find the set of all reachable nodes from these three nodes in the Program Dependence Graph of the program shown in Figure 2. This set, {1, 2, 3, 5, 6, 8}, gives us the desired slice. These nodes are shown in bold in the figure.

[5]This definition of control-dependence is for programs with structured control flow. For such programs, the control-dependence subgraph essentially reflects the nesting structure of statements in the program. In programs with arbitrary control flow, a control-dependence edge from vertex $v_i$ to vertex $v_j$ implies that $v_j$ is the nearest inverse dominator of $v_i$ in the control flow graph of the program (see [FOW87] for details).

# 3 Dynamic Slicing

As we saw above the static slice for the program in Figure 1 with respect to variable Y at statement 10 contains all three assignment statements, namely, 3, 6 and 8, that assign a value to Y. We know that for any input value of X only one of these three statements may be executed. Consider the test-case when X is $-1$. In this case only the assignment at statement 3 is executed. So the dynamic slice, with respect to variable Y at statement 10, will contain only statements 1, 2, and 3, as opposed to the static slice which contains statements 1, 2, 3, 5, 6, and 8. If the value of Y at statement 10 is observed to be wrong for the above test-case, we know that either there is an error in $f_1$ at statement 3 or the if predicate at statement 2 is wrong. Clearly, the dynamic slice, {1, 2, 3}, would help localize the bug much more quickly than the static slice, {1, 2, 3, 5, 6, 8}.

In the next few sections, we examine some approaches to computing dynamic slices. We denote the execution history of the program under the given test-case by the sequence $<v_1, v_2, \ldots, v_n>$ of vertices in the program dependence graph appended in the order in which they are visited during execution. We use superscripts 1, 2, etc. to distinguish between multiple occurrences of the same node in the execution history. For example, the program in Figure 3 has the execution history $<1, 2, 3, 4, 5^1, 6^1, 7^1, 8^1, 5^2, 6^2, 7^2, 8^2, 5^3, 9>$ when N is 2.

Given an execution history *hist* of a program $P$ for a test-case *test*, and a variable *var*, the dynamic slice of $P$ with respect to *hist* and *var* is the set of all statements in *hist* whose execution had some *effect* on the value of *var* as observed at the end of the execution. Note that unlike static slicing where a slice is defined

```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         Y := 0;
S4:         I := 1;
S5:         while (I <= N)
            do
S6:           Z := f1(Z, Y);
S7:           Y := f2(Y);
S8:           I := I + 1;
            end_while;
S9:         write(Z);
        end.
```

Figure 3: Example Program 2

with respect to a given location in the program, we define dynamic slicing with respect to the end of execution history. If a dynamic slice with respect to some intermediate point in the execution is desired, then we simply need to consider the partial execution history up to that point.

## 3.1 Dynamic Slicing: Approach 1

We saw above that the static slice with respect to variable Y at statement 10 for the program in Figure 1 contains all three assignment statements—3, 6, and 8; although for any given test-case, only one of these statements is executed. If we mark the nodes in the Program Dependence Graph that get executed for the current test-case, and traverse only the marked nodes in the graph, the slice obtained will contain only nodes executed for the current test-case. So our first simple approach to determining dynamic slices is informally stated as follows:

> To obtain the dynamic slice with respect to a variable for a given execution history, first take the "projection" of the Program Dependence Graph with respect to the nodes that occur in the execution history, and then use the static slicing algorithm on the projected Dependence Graph to find the desired dynamic slice.

Figure 4 shows the application of this approach for the program in Figure 1 for test-case X = −1, which yields the execution history <1, 2, 3, 4, 10, 11>. All nodes in the graph are drawn dotted in the beginning. As statements are executed, corresponding nodes in the graph are made solid. Then the graph is traversed only for solid nodes, beginning at node 3, the last definition of Y in the execution history. All nodes reached during the traversal are made bold. The set of all bold nodes, {1, 2, 3} in this case, gives the desired slice.

Unfortunately, the above naive approach does not always yield precise dynamic slices: It may sometimes include extra statements in the slice that did not affect the value of the variable in question for the given execution history. To see why, consider the program in Figure 3 and the test-case N = 1, which yields the execution history <1, 2, 3, 4, $5^1$, 6, 7, 8, $5^2$, 9>. Figure 5 shows the the result of using the above approach to obtain the dynamic slice of this program with respect to the variable Z at the end of the execution. Looking at the execution history we find that statement 7 assigns a value to Y which is never used later, for none of the statements that appear after 7 in the execution history, namely, 8, 5, and 9, uses variable Y. So statement 7 should not be in the dynamic slice. It is included in the slice because statement 9 depends on statement 6 which has a data dependence edge to statement 7 in the Program Dependence Graph. In the next section we present a refinement to the above approach that avoids this problem.

## 3.2 Dynamic Slicing: Approach 2

The problem with Approach 1 lies in the fact that a statement may have multiple reaching definitions of the same variable in the program flow-graph, and hence it may have multiple out-going data dependence edges for the same variable in the Program Dependence Graph. Selection of such a node in the dynamic slice, according to that approach, implies that all nodes to which it has out-going data-dependence edges also be selected if the nodes have been executed, even though the corresponding data-definitions may not have affected the current node. In the example above (Figure 3), statement 6 has multiple reaching definitions of the same variables: two definitions of variable Y from statements 3 and 7, and two of variable Z from statements 2 and 6 itself. So it has two outgoing data dependence edges for each of variables Y and Z: to statements 3 and 7, and 2 and 6 respectively (besides a control dependence edge to node 5). For the test-case N = 1, each of these four statements is executed, so inclusion of statement 6 in the slice leads to the inclusion of statements 3, 7, and 2 as well, even though two of the data dependencies of statement 6—on statement 7 for variable Y and on itself for variable Z—are never activated for this test-case.

In general, a statement may have multiple reaching definitions of a variable because there could be multiple execution paths leading up to that statement, and each of these paths may have different statements assigning a value to the same variable. For any single

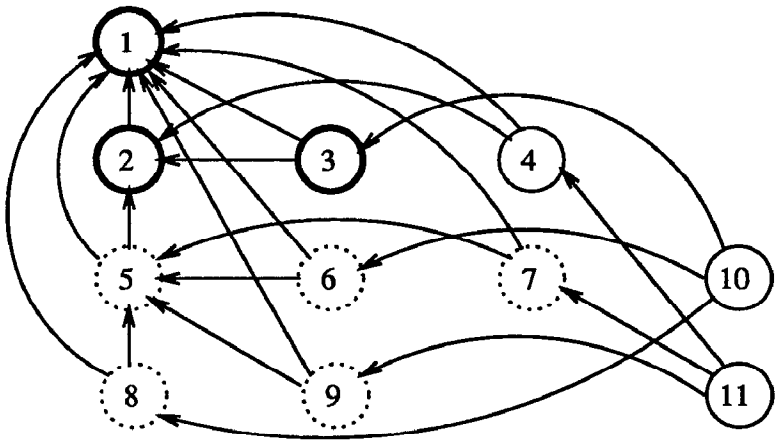


Figure 4: Dynamic Slice using Approach 1 for the program in Figure 1, test-case X = −1, with respect to variable Y at the end of the execution. All nodes are drawn as dotted in the beginning. A node is made solid if it is ever executed; and is made bold if it gets traversed while determining the slice.

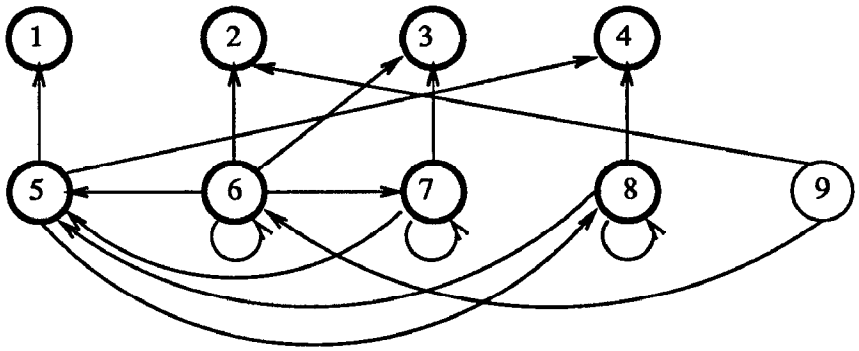


Figure 5: Dynamic slice using Approach 1 for the program in Figure 3, test-case N = 1, for variable Z, at the end of execution. Node 7 should not belong to the slice!

path, there can be at most one reaching definition of any variable at any statement; and since, in dynamic slicing, we are interested in examining dependencies for the single execution path under the given inputs, inclusion of a statement in the dynamic slice should lead to inclusion of only those statements that actually defined values used by it under the current test-case. This suggests our Approach 2 to computing dynamic slices:

> Mark the edges of the Program Dependence Graph as the corresponding dependencies arise during the program execution; then traverse the graph only along the marked edges to find the slice.

Consider again the program in Figure 3 and the test-case N = 1. Using Approach 2 on its execution history <1, 2, 3, 4, $5^1$, 6, 7, 8, $5^2$, 9> for variable Z yields the dynamic slice {1, 2, 3, 4, 5, 6, 8}. This is depicted in Figure 6. Imagine all edges to be drawn as dotted lines in the beginning. As statements are executed, edges corresponding to the new dependencies that occur are changed to solid lines. Then the graph is traversed only along solid edges and the nodes reached are made bold. The set of all bold nodes at the end gives the desired slice. Note that statement 7 that was included by Approach 1 in the slice is not included under this approach.

If a program has no loops then the above approach would always find accurate dynamic slices of the program (see [AH89] for details). In the presence of loops, the slice may sometimes include more statements than necessary. Consider the program in Figure 7 and the test-case where N = 2 and the two values of X read are −4 and 3. Then, for the first time through the loop statement 6, the **then** part of the **if** statement, is executed and the second time through the loop statement 7, the **else** part, is executed. Now suppose the execution has reached just past statement 9 second time through the loop and the second value of Z printed is found to be wrong. The execution history thus far is <1, 2, $3^1$, $4^1$, $5^1$, 6, $8^1$, $9^1$, $10^1$, $3^2$, $4^2$, $5^2$, 7, $8^2$, $9^2$>. If we used Approach 2 to find the slice for variable Z for this execution history, we

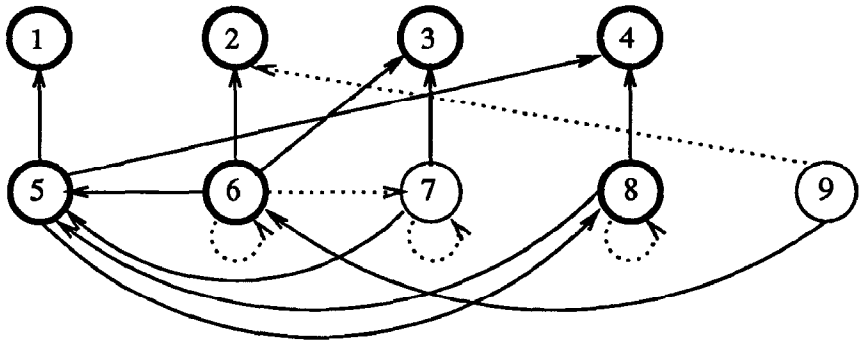


Figure 6: Dynamic Slice using Approach 2 for the program in Figure 3, test-case N = 1, for variable Z, at the end of execution. All edges are drawn as dotted at the beginning. An edge is made solid if the corresponding dependency is ever activated during execution. Only solid edges are traversed while slicing; nodes in the bold denote the slice obtained.

```
        begin
S1:         read(N);
S2:         I := 1;
S3:         while (I <= N)
            do
S4:           read(X);
S5:           if (X < 0)
              then
S6:               Y := f1(X);
              else
S7:               Y := f2(X);
              end_if;
S8:           Z := f3(Y);
S9:           WRITE(Z);
S10:          I := I + 1;
            end_while;
        end.
```

Figure 7: Example Program 3

would have both statements 6 and 7 included in the slice, even though the value of Z in this case is only dependent on statement 7. Figure 8 shows a segment of the Program Dependence Graph (only statements 4, 6, 7, 8, and 9) along with the effect of using Approach 2. The data dependence edge from 8 to 6 is marked during the first iteration, and that from 8 to 7 is marked during the second iteration. Since both these edges are marked, inclusion of statement 8 leads to inclusion of both statements 6 and 7, even though the value of Z observed at the end of second iteration is only affected by statement 7.

It may seem that the difficulty with the above approach will disappear if, before marking the data-dependence edges for a new occurrence of a statement in the execution history, we first *unmarked* any outgoing dependence edges that are already marked for this statement. This scheme will work for the above example, but unfortunately it may lead to wrong dynamic slices in other situations. Consider, for example, the program in Figure 9. Consider the case when the loop is iterated twice, first time through statements 7 and 11, and second time through statement 8 but skipping statement 11. If we obtain the dynamic slice for A at the end of execution, we will have statement 8 in the slice instead of statement 7. This is because when statement 9 is reached second time through the loop, the dependence edge from 9 to 7 (for variable Y) is unmarked and that from 9 to 8 is marked. Then, while finding the slice for A at statement 13, we will include statement 11, which last defined the value of A. Since statement 11 used the value of Z defined at statement 9, 9 is also included in the slice. But inclusion of 9 leads to inclusion of 8 instead of 7, because the dependence edge to the latter was unmarked during the second iteration. Value of Z at statement 11, however, depends on value of Y defined by statement 7 during the first iteration, so 7 should be in the slice, not 8. Thus the scheme of unmarking previously marked edges with every new occurrence of a statement in the execution history does not work.

## 3.3 Dynamic Slicing: Approach 3

Approach 2 discussed above sometimes leads to overlarge dynamic slices because a statement may have multiple occurrences in an execution history, and different occurrences of the statement may have different reaching definitions of the same variable used by the statement. The Program Dependence Graph does not distinguish between these different occurrences, so inclusion of a statement in the dynamic slice by virtue of one occurrence may lead to the inclusion of statements on which a different occurrence of that

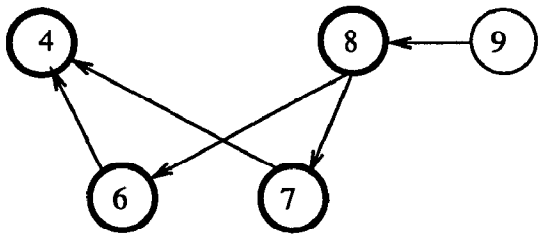


Figure 8: A subset of the dynamic slice obtained using Approach 2 for the program in Figure 7, test-case (N = 2, X = −4, 3), for Variable Z. Node 6 should not be in the slice!

```
        begin
S1:         read(N);
S2:         A := 0;
S3:         I := 1;
S4:         while (I <= N)
            do
S5:           read(X);
S6:           if (X < 0)
              then
S7:               Y := f1(X);
              else
S8:               Y := f2(X);
              end_if;
S9:           Z := f3(Y);
S10           if (Z > 0)
              then
S11               A := f4(A, Z);
              else
              end_if;
S12:          I := I + 1;
            end_while;
S13:        write(A);
        end.
```

Figure 9: Example Program 4

statement is dependent. In other words, different occurrences of the same statement may have different dependencies, and it is possible that one occurrence contributes to the slice and another does not. Inclusion of one occurrence in the slice should lead to inclusion of only those statements on which this occurrence is dependent, not those on which some other occurrences are dependent. This suggests our third approach to dynamic slicing:

> Create a separate node for each occurrence of a statement in the execution history, with outgoing dependence edges to only those statements (their specific occurrences) on which this statement occurrence is dependent.

Every node in the new dependence graph will have at most one out-going edge for each variable used at the statement. We call this graph the *Dynamic Dependence Graph*. A program will have different dynamic dependence graphs for different execution histories. Miller and Choi also define a similar dynamic dependence graph in [MC88]; however, their approach differs from ours in the way the graph gets constructed (see Section 4).

Consider, for example, the program in Figure 7, and the test-case (N = 3, X = −4, 3, −2), which yields the execution history <1, 2, $3^1$, $4^1$, $5^1$, $6^1$, $8^1$, $9^1$, $10^1$, $3^2$, $4^2$, $5^2$, $7^1$, $8^2$, $9^2$, $10^2$, $3^3$, $4^3$, $5^3$, $6^2$, $8^3$, $9^3$, $10^3$, $3^4$>. Figure 10 shows the Dynamic Dependence Graph for this execution history. The middle three rows of nodes in the figure correspond to the three iterations of the loop. Notice the occurrences of node 8 in these rows. During the first and third iterations, node 8 depends on node 6 which corresponds to the dependence of statement 8 for the value of Y assigned by node 6, whereas during the second iteration, it depends on node 7 which corresponds to the dependence of statement 8 for the value of Y assigned by node 7.

Once we have constructed the Dynamic Dependence Graph for the given execution history, we can easily obtain the dynamic slice for a variable, *var*, by first finding the node corresponding to the last definition of *var* in the execution history, and then finding all nodes in the graph reachable from that node. Figure 10 shows the effect of using this approach on the Dynamic Dependence Graph of the program in Figure 7 for the test-case (N = 3, X = −4, 3, −2), for variable Z at the end of the execution. Nodes in bold belong to the slice. Note that statement 6 belongs to the slice whereas statement 7 does not. Approach 2, on the other hand, would have included statement 7 as well.

## 3.4 Dynamic Slicing: Approach 4

The size of a Dynamic Dependence Graph (total number of nodes and edges) is, in general, *unbounded*.

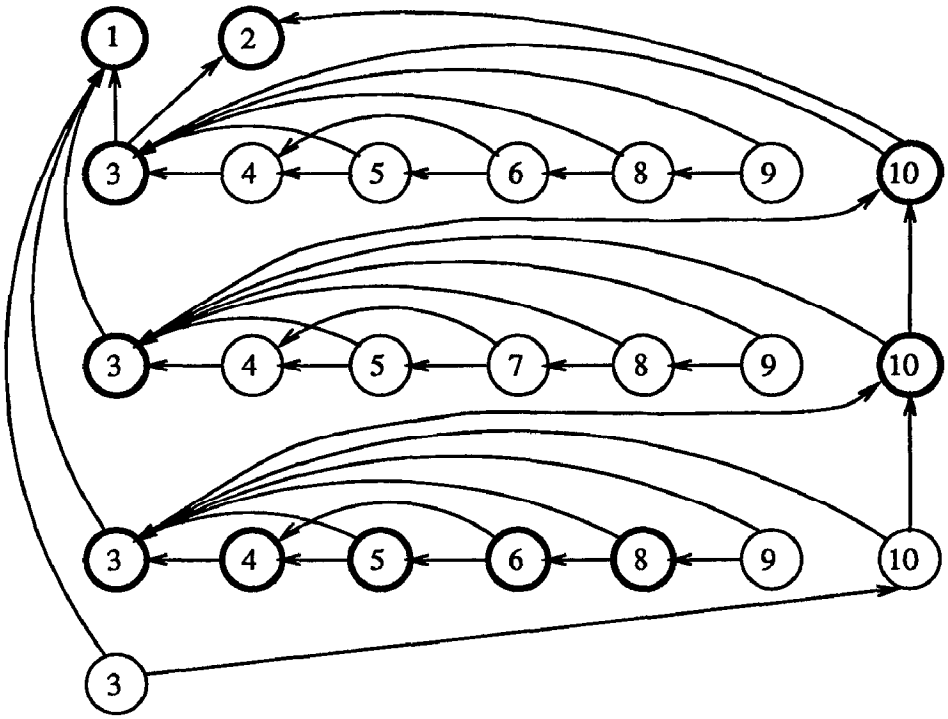


Figure 10: Dynamic Dependence Graph for the Program in Figure 7 for the test-case (N = 3, X = −4, 3, −2). Nodes in bold give the Dynamic Slice for this test-case with respect to variable Z at the end of execution.

This is because the number of nodes in the graph is equal to the number of statements in the execution history, which, in general, may depend on values of run-time inputs. For example, for the program in Figure 3 the number of statements in its execution history, and hence the size of its Dynamic Dependence Graph, depends on the value read by variable N at statement 1. On the other hand, we know that every program can have only a finite number of possible dynamic slices — each slice being a subset of the (finite) program. This suggests that we ought to be able to restrict the number of nodes in a Dynamic Dependence Graph so its size is not a function of the length of the corresponding execution history. Our fourth approach exploits the above observation:

> Instead of creating a new node for every occurrence of a statement in the execution history, create a new node only if another node with the same transitive dependencies does not already exist.

We call this new graph the *Reduced Dynamic Dependence Graph*. To build it without having to save the entire execution history we need to maintain two tables called *DefnNode* and *PredNode*. *DefnNode* maps a variable name to the node in the graph that last assigned a value to that variable. *PredNode* maps a control predicate statement to the node that corresponds to the last occurrence of this predicate in the execution history thus far. Also, we associate a set, *ReachableStmts*, with each node in the graph. This set consists of all statements one or more of whose occurrences can be reached from the given node. Every time a statement, $S_i$, gets executed, we determine the set of nodes, $D$, that last assigned values to the variables used by $S_i$, and the last occurrence, $C$, of the control predicate node of the statement. If a node, $n$, associated with $S_i$ already exists whose immediate descendents are the same as $D \cup C$, we associate the new occurrence of $S_i$ with $n$. Otherwise we create a new node with outgoing edges to all nodes in $D \cup C$. The *DefnNode* table entry for the variable assigned at $S_i$, if any, is also updated to point to this node. Similarly, if the current statement is a control predicate, the corresponding entry in *PredNode* is updated to point to this node.

If there were no circular dependencies in the dependence graph then the above scheme of looking for a node with the same set of immediate descendents would work fine. But in presence of circular dependencies (i.e., in presence of loops in the program dependence graph), the graph reduction described above won't occur: for every iteration of a loop involving circular dependencies we will have to create new node occurrences. We can avoid this problem, if whenever we need to create a new node, say for statement $S_i$, we first determine if any of its immediate

descendents, say node $v$, already has a dependency on a previous occurrence of $S_i$ and if the other immediate descendents of the new occurrence of $S_i$ are also reachable from $v$. This is easily done by checking if the *ReachableStmts* set to be associated with the new occurrence is a subset of the *ReachableStmts* set associated with $v$. If so, we can merge the new occurrence of $S_i$ with $v$. After this merge, during subsequent iterations of the loop the search for a node for $S_i$ with same immediate descendents will always succeed.

Consider again the program in Figure 7, and test-case (N = 3, X = −4, 3, −2), which yields the execution history <1, 2, $3^1$, $4^1$, $5^1$, $6^1$, $8^1$, $9^1$, $10^1$, $3^2$, $4^2$, $5^2$, $7^1$, $8^2$, $9^2$, $10^2$, $3^3$, $4^3$, $5^3$, $6^2$, $8^3$, $9^3$, $10^3$, $3^4$>. Figure 11 shows the Reduced Dynamic Dependence Graph for this execution history. Every node in the graph is annotated with the set of all reachable statements from that node. Note that there is only one occurrence of node 10 in this graph, as opposed to three occurrences in the Dynamic Dependence Graph for the same program and the same test-case. Also note that the second occurrence of node 3 is merged with its immediate descendent node 10 because the *ReachableStmts* set, {1, 2, 3, 10}, of the former was a subset of that of the latter. The third occurrence of node 3 in the execution history has node 1 and node 10 as immediate descendents. Since these immediate dependencies are also contained in the merged node (10,3), the third occurrence of node 3 is also associated with this node.

Once we have the Reduced Dynamic Dependence Graph for the given execution history, to obtain the dynamic slice for any variable *var* we first find the entry for *var* in the *DefnNode* table. The *ReachableStmts* set associated with that entry gives the desired dynamic slice. So we don't even have to traverse the Reduced Dynamic Dependence Graph to find the slice. For example, the dynamic slice for variable Z in case of the Reduced Dynamic Dependence Graph in Figure 7 is given by the *ReachableStmts* set, {1, 2, 3, 4, 5, 6, 8, 10}, associated with node 8 in the last row, as that was the last node to define value of Z.

# 4 Related Work

The concept of program slicing was first proposed by Weiser [Wei84, Wei82]. His solution for computing static program slices was based on iteratively solving data-flow equations representing inter-statement influences. Ottenstein and Ottenstein later presented a much neater solution for static slicing in terms of graph reachability in the Program Dependence Graph [OO84], but they only considered the intra-procedural case. Horwitz, Reps, and Binkley have proposed extending the Program Dependence Graph representation to what they call System Dependence Graph to find inter-procedural static slices under the same graph-reachability framework [HRB88]. Dependence Graph representation of programs was first proposed by Kuck et al. [KKL+81]; several variations of this concept have since been used in optimizing and parallelizing compilers [FOW87] besides their use in program slicing.

Korel and Laski extended Weiser's static slicing algorithms based on data-flow equations for the dynamic case [KL88]. Their definition of a dynamic slice may yield unnecessarily large dynamic slices. They require that if any one occurrence of a statement in the execution history is included in the slice then all other occurrences of that statement be automatically included in the slice, even when the value of the variable in question at the given location is unaffected by other occurrences. The dynamic slice so obtained is executable and produces the same value(s) of the variable in question at the given location as the original program. For our purposes, the usefulness of a dynamic slice lies not in the fact that one can execute it, but in the fact that it isolates only those statements that affected a particular value observed at a particular location. For example, in the program of Figure 7 each loop iteration computes a value of Z, and each such computation is totally independent of computation performed during any other iteration. If the value of variable Z at the end of a particular iteration is found be incorrect and we desire the dynamic slice for Z at the end of that iteration, we would like only those statements to be included in the slice that affected the value of Z observed at the end of that iteration, not during all previous iterations, as the previous iterations have no effect on the current iteration. It is interesting to note that our Approach 2 (which may yield an overlarge dynamic slice) would obtain the same dynamic slice as obtained under their definition. So our algorithm for dynamic slicing based on the graph-reachability framework may be used to obtain dynamic slices under their definition, instead of using the more expensive algorithm based on iterative solutions of the data-flow equations.

Miller and Choi also use a dynamic dependence graph, similar to the one discussed in Section 3.3, to perform flow-back analysis [Bal69] in their Parallel Program Debugger PPD [MC88]. Our approach, however, differs from theirs in the way the graph is constructed. Under their approach, separate data-dependence graphs of individual basic blocks are constructed. The dynamic dependence graph is build by combining, in order, the data-dependence graphs of all basic blocks reached during execution and in-

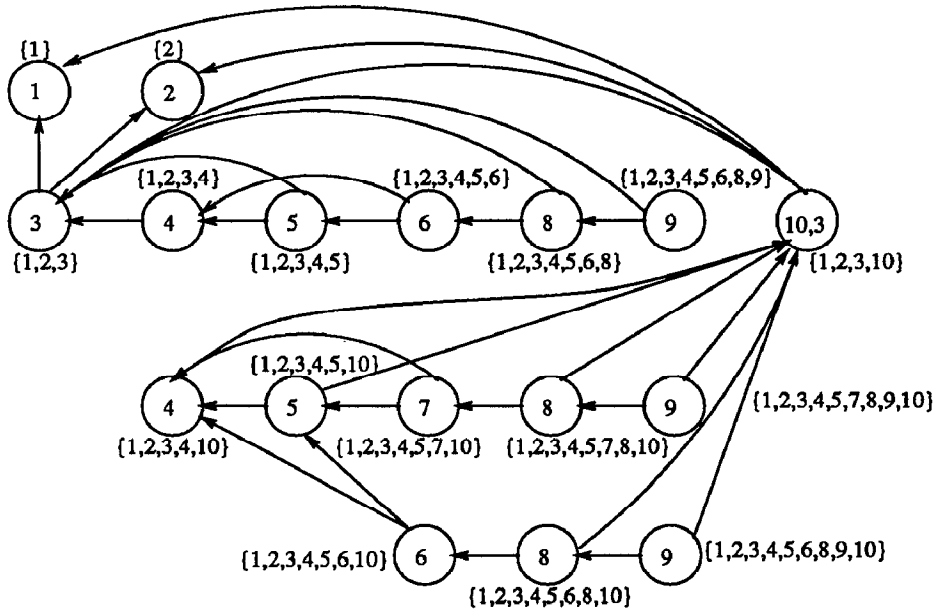


Figure 11: The Reduced Dynamic Dependence Graph for the Program in Figure 7 for the test-case (N = 3, X = −4, 3, −2), obtained using Approach 4. Each node is annotated with *ReachableStmts*, the set of all statements reachable from that node.

serting appropriate control dependence edges among them. They use a notion of incremental tracing where portions of the program state are checkpointed at the start and the end of segments of program-code called emulation-blocks. Later these emulation blocks may be reexecuted to build the corresponding segments of the dynamic dependence graph. The size of their dynamic dependence graph may not be bounded for the same reason as that discussed in Section 3.4.

# 5 Summary

In this paper we have examined four approaches for computing dynamic program slices. The first two are extensions of static program slicing using Program Dependence Graph. They are simple and efficient; however, they may yield bigger slices than necessary. The third approach uses Dynamic Dependence Graph to compute accurate dynamic slices but the size of these graphs may be unbounded, as it depends on the length of execution history. Knowing that every program execution can have only a finite number of dynamic slices it seems unnecessary having to create a separate node in the Dynamic Dependence Graph for each occurrence of a statement in the execution history. We then proposed the notion of a Reduced Dynamic Dependence Graph where a new node is created only if it can cause a new dynamic slice to be introduced. The size of the resulting graph is proportional to the actual number of dynamic slices that arose during the execution and not to the length of the execution.

## Acknowledgements

We would like to thank Rich DeMillo, Stu Feldman, Gene Spafford, Ryan Stansifer, and Venky Venkatesh for their many helpful comments on an earlier draft of this paper.

# References

[AH89] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. Technical Report SERC-TR-56-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, November 1989.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bal69] R. M. Balzer. Exdams—extendable debugging and monitoring system. In *AFIPS Proceedings, Spring Joint Computer Conference*, 1969, volume 34, pages 567–580.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkeley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988. SIGPLAN Notices, 23(7):35–46, July 1988.

[KKL+81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, Williamsburg, Virginia, January 1981. pages 207–218.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, October 1988.

[MC88] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988. SIGPLAN Notices, 23(7):135–144, July 1988.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Pittaburgh, Pennsilvania, April 1984. SIGPLAN Notices, 19(5):177–184, May 1984.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.