Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

ELSEVIER

Information Processing Letters 85 (2003) 145–152

Information Processing Letters

www.elsevier.com/locate/ipl

A compact execution history for dynamic slicing

Dhananjay M. Dhamdhere *, K. Gururaja, Prajakta G. Ganu

Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India

Received 17 July 2001; received in revised form 3 May 2002

Communicated by R. Backhouse

Keywords: Software design and implementation; Dynamic slicing; Execution history; Data dependence; Control dependence

1. Introduction

A *slice* of a program P with respect to a slicing criterion $C \equiv (\{var\}, c\_stmt)$ is a subset of the program which includes all statements that directly or indirectly affect the value of variable *var* in *c_stmt* [1,10–12]. A *static slice* includes all statements which *might* affect the value of *var*. It is constructed using program analysis techniques. A *dynamic slice* consists of only those statements that actually influence the value of *var* in an execution of the program. It is built using run-time information. A dynamic slice is more precise than a static slice because it contains only those statements which have actually influenced *var* in an execution.

A dynamic slice of a program is constructed by analyzing an *execution history* of the program to discover data and control dependences.[1] A complete execution history records all actions performed during an execution of a program. A lot of this information is redundant for dynamic slicing. We develop a compact execution history for dynamic slicing of programs using the notion of *critical statements* in a program. Only critical statements need to appear more than once in an execution history; all other statements appear at most once. Performance studies show that our execution histories are at least an order of magnitude smaller than complete execution histories in most cases; for some programs they are smaller by several orders of magnitude.

2. Dynamic slicing

Slicing was first discussed by Weiser [12]. Korel and Laski introduced dynamic slicing [10]. Other relevant papers are [1,7,9,11]. This section presents dynamic slicing along the lines described in [10].

2.1. Dynamic slicing algorithm

An execution history is a sequence $\ldots J^{q-1} I^q K^{q+1} \ldots$, where I, J and K are statements in the program and $q-1$, q and $q+1$ are their positions in the sequence. I^q is called a statement occurrence of I. Many occurrences of a statement may exist in an execution history. A dynamic slicing criterion is

* Corresponding author.

E-mail addresses: dmd@cse.iitb.ac.in (D.M. Dhamdhere), kgururaj@cse.iitb.ac.in (K. Gururaja), prajakta@cse.iitb.ac.in (P.G. Ganu).

[1] An alternative approach uses program dependence graphs [1, 11].

PII: S0020-0190(02)00322-8

defined as C $\equiv$ (*input*, V, I^q) where *input* is the set of values of input variables for which the program is executed, I^q is a specific execution of I, and V is the set of variables of interest. To obtain a dynamic slice for this criterion, we use an execution history of the program constructed when it is executed with *input* as the set of input values. Hence the shorter specification (V, I^q) suffices for dynamic slicing using a specific execution history.

Notions of data dependence and control dependence analogous to those defined in [8] are used to identify statements which are relevant to the statement in the slicing criterion. Let s_j be the criterion statement. s_j is *dynamically data dependent* on a statement s_i if s_j uses a variable *var* defined by s_i and execution of the program traverses a path from s_i to s_j along which *var* is not redefined. If $var \in V$, s_i affects the value of *var* at s_j, hence s_i is included in the slice. s_j is *dynamically control dependent* on a statement s_i if s_i contains a predicate which decides whether s_j would be executed. Hence s_i is included in the slice. Control dependence is caused by statements like looping statements, *if* statements and *switch* (or *case*) statements. Static and dynamic control dependences are different in programs which contain *if* ... *then goto* ... statements.

Fig. 1 shows the dynamic slicing algorithm. It uses three functions: *Previous_def*(*var*, I^q) returns the definition of *var* on which I^q is dynamically data dependent [7]. *In_control_of*(I) returns the set of statements in the program on which I is statically control dependent. *Use*(I) returns the variables referenced in I.

Procedure *Get_dynamic_slice* is called with arguments V, I and q, where $V \subseteq Use(I)$. Set *stmt_occ* contains statement occurrences which must be processed to discover all data and control dependences. It is initialized to I^q and statement occurrences are added to it on the basis of data and control dependences. All occurrences of statements in *In_control_of*(I) which precede I^q in the execution history are included in set C. Set D is then constructed to contain definitions of variables in V on which some occurrence of I before I^q or I^q itself (i.e., occurrence I^r, $r \leqslant q$) is data dependent. The procedure now calls itself recursively to discover data and control dependences of all occurrences of statements in C and D which precede I^q. At end, *slice* is constructed as a set of statements whose occurrences exist in *stmt_occ*.

program *Dynamic_slicing*
  stmt_occ := $\{I^q\}$;
  call *Get_dynamic_slice*(V, I, q);
  slice := $\{s_i \mid \exists s_i^r \in stmt\_occ$ for some $r\}$;
end;

procedure *Get_dynamic_slice*(V, I, q)
  $C := \{J^p \mid J \in In\_control\_of(I)$ and $p \leqslant q\}$;
  $D := \{\}$;
  $\forall\, var \in V$
    $\forall\, I^r \ni r \leqslant q$
      $D := D \cup \{Previous\_def(var, I^r)\}$;
  temp := $C \cup D - stmt\_occ$;
  stmt_occ := $stmt\_occ \cup temp$;
  $\forall\, J^p \in temp$
    $W := Use(J^p)$;
    call *Get_dynamic_slice*(W, J, q);
end *Get_dynamic_slice*;

function *Previous_def*(*var*, I^q)
  return J^p, the last definition of *var* in
    execution history such that $p < q$.
end *Previous_def*;

Fig. 1. Dynamic slicing algorithm.

3. Some execution trajectories

We use the term *execution trajectory* (ET) for an execution history. Let $G = (N, E, n_0)$ be a program flow graph (PFG) [2]. For simplicity of exposition, we assume each node N_i to contain a single statement s_i, however our scheme is applicable even if this requirement is not met. (A node is broken up if it contains > 1 statement. If parts of a statement—e.g., a **for** statement—belong to different nodes, each part is analysed as a separate statement.)

We follow the approach used in [7] to construct an ET. A variable $time_i$, initialized to 0, is associated with each node N_i of PFG. A *time-stamp* is the value in an integer counter which is incremented by 1 whenever execution of a node begins. Code instrumentation [5] is used for this purpose. Every time N_i is executed, the time-stamp is copied into $time_i$. A *block* is a pair (N_i, t_j), where t_j is the time-stamp when N_i was executed. Thus block (N_i, t_j) represents an execution of node N_i. If statement I is located in node N_i, block (N_i, t_j) represents same information as I^{t_j}.

A *complete execution trajectory* (CET) contains blocks representing every execution of every node during the program's execution. The memory requirement of CET is unbounded because a node can figure in any number of blocks. A *minimal execution trajectory* (MET) [6] is constructed as follows: At the end of program execution, a block $(N_i, time_i)$ is formed for each node N_i in G such that $time_i \neq 0$. The blocks are sorted in ascending order by time-stamp to form MET. Thus each (N_i, t_j) in MET represents the last execution, if any, of N_i. The memory requirement of MET is $O(|N|)$. Fig. 3 shows CET and MET for an execution of the program of Fig. 2 in which branches (N_3, N_5), (N_3, N_4) and (N_3, N_5) are taken in the first three iterations before execution reaches the start of node N_6.

It is not possible to use MET for dynamic slicing. Consider construction of dynamic slice of the program of Fig. 2 for the slicing criterion $(\{v\}, s_6^{15})$. The slice consists of statements $s_1, s_2, s_3, s_5, s_6, s_7$. However if the MET of Fig. 3 is used, procedure *Get_dynamic_slice* fails to find dynamic data dependence of s_2, i.e., $v := a$, on s_5, i.e., $a := 5$ because the only occurrence of s_5 in MET follows the occurrence of s_2. CET of Fig. 3 is adequate, however many blocks in it are redundant for the purpose of constructing this slice, for example blocks $(N_6, 5)$, $(N_7, 6)$, $(N_2, 7)$, $(N_3, 8)$, $(N_4, 9)$, $(N_6, 10)$, $(N_3, 13)$ and $(N_5, 14)$.

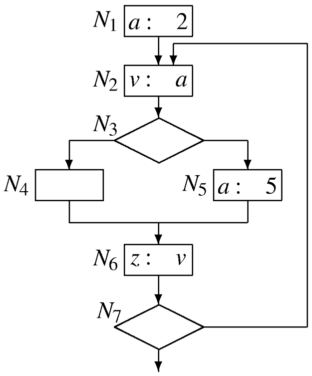

Fig. 2. A program flow graph.

CET:	$(N_1, 1), (N_2, 2), (N_3, 3), (N_5, 4), (N_6, 5), (N_7, 6), (N_2, 7), (N_3, 8), (N_4, 9), (N_6, 10), (N_7, 11), (N_2, 12), (N_3, 13), (N_5, 14), (N_6, 15)$
MET:	$(N_1, 1), (N_4, 9), (N_7, 11), (N_2, 12), (N_3, 13), (N_5, 14), (N_6, 15)$

Fig. 3. CET and MET.

A *selective execution trajectory* (SET) is a via media between a CET and a MET. It contains fewer redundancies than CET, but it is adequate for dynamic slicing.

4. A SET for dynamic slicing

c_stmt is the statement mentioned in the slicing criterion. DC_{s_i} is a set containing s_i and statements on which s_i is data or control dependent. $DC^*_{s_i}$ is the closure of DC_{s_i}. It contains a statement s_j iff a sequence $s_j \equiv s^0, s^1, \ldots, s^n \equiv s_i$ such that for $k = 1, \ldots, n$ $s^{k-1} \in DC_{s^k}$ can be constructed for some n. $node(s_i)$ denotes the node of G which contains statement s_i. Predicate $Same\_loop(s_i, s_j)$ is true only if some loop encloses both s_i and s_j. VI_{s_i} is the set of variables of interest while processing statement s_i. If $s_i \equiv c\_stmt$, $VI_{s_i} = V$, else $VI_{s_i} = Use(s_i)$. $R_{s_i,var}$ is the set of definitions of $var \in VI_{s_i}$ which reach s_i.

4.1. Critical nodes

The slicing algorithm of Fig. 1 uses an ET to discover data and control dependences. MET contains the last execution of a node. It is adequate for slicing if a statement s_i is data or control dependent only on one statement, say, s_j, and s_j always occurs earlier in MET. When these conditions do not hold, an ET must contain some previous occurrences of s_i or s_j. Such statements are called *critical statements* (CSs), and nodes containing them are called *critical nodes* (CNs).

We determine criticality of a node under the assumption that the slicing criterion is of the form (V, c_stmt^f) where c_stmt^f is the last statement occurrence for *c_stmt* in ET and $V \subseteq Use(c\_stmt)$.

Definition 1 (*Critical statement*). Statement s_i is a critical statement if it satisfies one of the following conditions:

(a) $s_i \in DC^*_{c\_stmt}$, $|R_{s_i,var} - \{s_i\}| > 1$ and $Same\_loop(s_i, s_j)$ for some $s_j \in R_{s_i,var} - \{s_i\}$.
(b) For some $s_j \in DC^*_{c\_stmt}$, $s_i \in DC_{s_j}$, $Same\_loop(s_i, s_j)$ and s_j is not included in some path $s_i \ldots c\_stmt$.

(c) s_l is a critical statement by part (b), $s_i \in DC^*_{s_l}$ and $Same\_loop(s_i, s_l)$.
(d) Some path $s_i \ldots s_k \ldots s_m$ exists in G such that s_m is a critical statement by part (a), $s_i, s_k \in R_{s_m,var} - \{s_m\}$, $Same\_loop(s_i, s_m)$ and $Same\_loop(s_k, s_m)$.

The slicing algorithm of Fig. 1 adds statements in $DC^*_{c\_stmt}$ to the slice. Part (a) of Definition 1 is obvious. Since $|R_{s_i,var} - \{s_i\}| > 1$ and $Same\_loop(s_i, s_j)$, more than one definition of *var* may dynamically reach s_i. These dynamic dependences can be discovered only if s_i appears more than once in the ET. $s_i \in R_{s_i,var}$ is ignored because $s_i \in DC^*_{c\_stmt}$ ensures that it would be added to the slice.

Let s_j be control or data dependent on s_i. Part (b) identifies conditions when s_i may occur later in ET than s_j. A previous execution of s_i must exist in ET if s_j's data and control dependences are to be discovered correctly. Hence s_i is a critical statement.

Let s_l be a critical statement by part (b) of Definition 1. The slicing algorithm of Fig. 1 may include some occurrence s_l^r, which is not its last occurrence, in *stmt_occ*. Control and data dependences of s_l^r are discovered through a backward search in ET. Hence if statements on which s_l is data or control dependent are located in the same loop as s_l, they too must appear multiple times in ET. This effect is incorporated by part (c) of Definition 1. Part (d) is motivated by the fact that if a CET contains a subsequence $s_i \ldots s_m \ldots s_i \ldots s_k \ldots s_m$, data dependence of s_m on s_i would be discovered only if s_i is a CS.

Consider the criterion $(\{v\}, s_6^{15})$ in the program of Fig. 2. Node N_2 is a CN according to part (a) of Definition 1 because s_2 uses a and two definitions of a reach it. Node N_5 is a CN because s_5, s_2 satisfy part (b) of Definition 1 due to path N_5–N_6. Nodes N_3, N_7 are CNs because s_3, s_5 and s_7, s_5 satisfy part (c) of Definition 1. These pairs of statements also satisfy part (b) of Definition 1.

4.2. Design of SET

We construct the selective execution trajectory (SET) for dynamic slicing as follows: Program instrumentation is used to build two kinds of trajectories

- MET is built using time-stamping information as described in [7].
- A *critical nodes execution trajectory* (CNET) is built by inserting a block (N_i, t_j) in CNET every time a critical node N_i is visited during an execution. Note that the size of CNET is unbounded, however CNET is smaller in size than CET for most program executions.

SET is formed by merging these trajectories, where merging is performed by arranging all blocks in MET $\cup$ CNET in ascending order by time-stamps and deleting duplicates.

Nodes N_2, N_3, N_5 and N_7 are critical nodes in the program of Fig. 2, hence CNET consists of blocks $(N_2, 2)$, $(N_3, 3)$, $(N_5, 4)$, $(N_7, 6)$, $(N_2, 7)$, $(N_3, 8)$, $(N_7, 11)$, $(N_2, 12)$, $(N_3, 13)$, $(N_5, 14)$. SET is therefore $(N_1, 1)$, $(N_2, 2)$, $(N_3, 3)$, $(N_5, 4)$, $(N_7, 6)$, $(N_2, 7)$, $(N_3, 8)$, $(N_4, 9)$, $(N_7, 11)$, $(N_2, 12)$, $(N_3, 13)$, $(N_5, 14)$, $(N_6, 15)$. Some of the redundancies of CET, viz. blocks $(N_6, 5)$ and $(N_6, 10)$ do not appear in SET.

Slice construction for criterion $(\{v\}, s_6^{15})$ by the algorithm of Fig. 1 using SET would proceed as follows: s_6^{15} would be added to *stmt_occ*. Sets C, D would be $\{s_7^{11}, s_7^6\}$ and $\{s_2^2, s_2^7, s_2^{12}\}$. Recursive call for the criterion $(\{v\}, s_2^{12})$ leads to $C = \{s_7^{11}, s_7^6\}$ and $D = \{s_5^4, s_1^1\}$. Recursive call for the criterion $(\{a\}, s_5^4)$ leads to $C = \{s_3^3, s_3^8, s_3^{13}\}$ and $D = \{\}$. At end $stmt\_occ = \{s_1^1, s_3^3, s_5^4, s_7^6, s_3^8, s_7^{11}, s_2^{12}, s_3^{13}, s_6^{15}\}$ and $slice = \{s_1, s_2, s_3, s_5, s_6, s_7\}$.

4.3. Instrumentation to build SET

We instrument the program to be sliced as follows: As described in Section 3, a variable $time_i$ is associated with every node N_i. Whenever a critical node N_i is executed, the instrumented code constructs a block $(N_i, time_i)$ and enters it into CNET. At the end of execution, time-stamps are used to build MET. This is done only for non-critical nodes.

Instrumentation analysis is performed by procedure *Instrumentation* (see Fig. 4) which constructs the set of critical nodes *SCN*. Parts (a), (b) of Definition 1 are implemented by this procedure. The basic approach is as follows: Critical nodes are identified for the criterion (V, s_i). Set DC_{s_i} contains statements on which s_i is control or data dependent. Analysis is now performed recursively for the criteria $(Use(s_k), s_k)$ for each $s_k \in DC_{s_i}$. Procedure *Include_in_CN* imple-

procedure *Instrumentation*(*criterion, flag*)
Let *criterion* $\equiv (V, s_i)$.
if *flag* = "any" **then call** *Include_in_CN*(s_i);
if some $s_k \in DC_{s_i}$ and s_i satisfy part (b) of Definition 1 $\wedge$
s_k is not i-marked **then call** *Include_in_CN*(s_k);
if s_i satisfies part (a) of Definition 1 **then**
$SCN = SCN \cup \{node(s_i)\}$;
$\forall\, s_k \in DC_{s_i}$
if s_k is not r-marked and not i-marked **then**
r-mark s_k;
call *Instrumentation*$((Use(s_k), s_k)$, "*last*");
call *Prune_CN*;
end *Instrumentation*;
Procedure *Include_in_CN*(s_l)
i-mark s_l;
$SCN = SCN \cup \{node(s_l)\}$;
$\forall\, s_k \in DC_{s_l} \ni s_k$ is not i-marked $\wedge Same\_loop(s_k, s_l)$
call *Include_in_CN*(s_k);
end *Include_in_CN*;
Procedure *Prune_CN*
$\forall\, i, j \in SCN \ni i$ is a unique predecessor of j
and j is a unique successor of i
Delete node i from *SCN*;
end *Prune_CN*;

Fig. 4. Instrumentation for dynamic slicing.

ments part (c) of Definition 1 by invoking itself recursively for all statements in DC_{s_l}. Since s_k may be included in set *DC* for many statements, we prevent repeated recursive calls by r-marking s_k and i-marking s_l in procedures *Instrumentation* and *Include_in_CN*, respectively. An i-marked statement is not processed by *Instrumentation*, but an r-marked statement is processed by *Include_in_CN* because processing performed by *Include_in_CN* subsumes that performed by *Instrumentation*.

Definition 1 identifies critical nodes if slicing is performed for the last execution of *c_stmt*. If slicing is desired for any execution of *c_stmt*, all occurrences of *c_stmt*, and all occurrences of statements in $DC^*_{c\_stmt}$ which are in same loop should also appear in ET. This requirement is incorporated by invoking *Include_in_CN* for s_i when *flag* = "any".

Procedure *Prune_CN* implements an obvious optimization by deleting a node from *SCN* if it always precedes another node in *SCN*. Occurrences of this node can be simply inserted after CNET is constructed. The only exception is if node j involves a function/procedure call, since statements of the function/procedure body may occur between i and j (see Section 4.5).

Two other simple optimizations are possible. A statement s_i which satisfies part (a) of Definition 1 need not be a CS if it is data dependent on two definitions one of which is itself data dependent on the other definition. For example, if s_i which uses i is data dependent on `i:=i+1` in same loop and `i:=...` outside the loop, s_i need not be a CS because if `i:=i+1` is added to slice, `i:=0` will also be added. A **for** statement causes both data and control dependence. Control dependence ensures that it is included in a slice if some statement in its body is included. Hence a **for** `(i=0;i<...;i++)` need not be made a CS even if use of i in its body and `i++` of the **for** statement satisfy part (b) of Definition 1.

For the program of Fig. 2 and slicing criterion $(\{v\}, s_6^{15})$, call *Instrumentation*$(\{v\}, s_6$, "last") leads to a recursive call for the criterion $(\{a\}, s_2)$. This call identifies N_2 as a CN by part (a) of Definition 1, and N_5 as a CN by part (b) of Definition 1 because of path $N_5 \ldots N_6$. It makes a call *Include_in_CN*(s_5) which marks N_3 and N_7 as CN. Other recursive calls do not identify any CNs.

4.4. *Proof of adequacy of SET*

Let s_i be a definition of v. Let $s_i^r \lessdot s_j^p$ indicate that occurrence s_i^r precedes occurrence s_j^p in SET, and let $s_i^r \to s_j^p$ indicate that s_i^r reaches s_j^p in SET (i.e., $s_i^r \lessdot s_j^p$ and no other definition of v occurs between s_i^r and s_j^p). s_j may be added to a slice to satisfy control or data dependences (see sets C and D in the slicing algorithm). We show that all dynamic control and data dependences of s_j can be discovered using SET.

Lemma 1. $\forall s_j^q \in stmt\_occ$,

(a) *If s_j is dynamically data dependent on $s_i : v := \ldots, s_i^r \to s_j^p$ for some r and some $p \leqslant q$.*
(b) *If s_j is dynamically control dependent on s_i, $s_i^r \lessdot s_j^p$ for some r and some $p \leqslant q$.*

Proof. Proof is by contradiction.

Part (a): Let CET contain a subsequence $s_i^r \ldots s_j^p$, $p \leqslant q$ but $\neg(s_i^r \to s_j^p)$. Three possibilities exist.

Case 1: CET contains the sequence $s_i^r \ldots s_j^t \ldots s_m^l \ldots s_j^q$, where s_m is a definition of v and s_j is not a CS. This implies that s_j is executed both before and after s_m, $Same\_loop(s_j, s_m)$, and two definitions of v—s_i and s_m—reach s_j. Since s_j^q is added to a slice, $s_j \in DC^*_{c\_stmt}$ (see Fig. 1). Hence s_j is a CS by part (a) of Definition 1. This is a contradiction.

Case 2: s_i is executed again such that CET contains a subsequence $s_i^r \ldots s_j^q \ldots s_i^h$ but s_i is not a CS. Subsequence $s_i^r \ldots s_j^q \ldots s_i^h$ implies that $Same\_loop(s_i, s_j)$, and a path exists from s_i to *c_stmt* which does not contain s_j. Hence s_i is a critical node by part (b) of Definition 1. This is a contradiction.

Case 3: CET contains a subsequence $s_i^r \ldots s_j^f \ldots s_k^g \ldots s_i^h \ldots s_m^l \ldots s_j^t$, where s_m is a definition of v, s_j^q is either s_j^f or s_j^t, and s_i is not a CS. Since s_j is executed before and after both s_i and s_m, $Same\_loop(s_i, s_j) \wedge Same\_loop(s_m, s_j)$. Hence s_j is a CS by part (a) of Definition 1. s_j^f is not the last occurrence of s_j in SET. If s_j^f is added to *stmt_occ* by the algorithm in Fig. 1 because $s_k \in DC^*_{c\_stmt}$ and s_k is data dependent on s_j, s_j is a CS also by part (b) of Definition 1. If s_j^t is added to *stmt_occ*, $s_j \in DC^*_{c\_stmt}$ hence s_i is a CS from part (d) of Definition 1. In both cases we have a contradiction.

Part (b): Let SET be $\ldots s_j^q \ldots s_i^t$. From the definition of dynamic control dependence, s_i must execute before s_j, hence CET must be $\ldots s_i^r \ldots s_j^q \ldots s_i^t$. Similar to Case 2 of part (a), this is a contradiction. □

4.5. Handling functions and arrays

Functions require special handling during criticality analysis because of interprocedural data dependences. Instrumentation analysis is simple if each function call is expanded in-line. However, such expansion may be infeasible or impossible if calls are deeply nested or recursive. In such cases we use summary information GMOD and GREF which represent the effect of a function call in terms of modification and use of actual parameters and global variables, respectively [3,4], to prepare a program for instrumentation analysis. Every call on a function f is replaced by a *function expression* (FE). FE consists of a single occurrence of a variable arity fictitious operator '@', whose operands are the globals and parameters of f whose values are used in computing the expression in the return statement of f. FE's operands are given by $GP \cap \bigcup_{s_i \in DC^*_{return}} Use(s_i)$, where GP is the set of globals and parameters of f. The modified call statement is followed by a *deemed definition* for each variable in GMOD of f. A deemed definition $s_k : v := @(\ldots)$ 'corresponds to' all definitions of v in the function body which reach its exit. $CO(s_k)$ is the set of such definitions. The rhs expression of a deemed definition s_k contains variables given by $GP \cap \bigcup_{s_i \in DC^*_{CO(s_k)}} Use(s_i)$.

Instrumentation analysis of the calling program is performed after these modifications. If more than one definition in the function body corresponds to a deemed definition, this fact is noted for use while applying part (a) of Definition 1. If a call statement or a deemed definition is a CS, some statements in the function body also become CSs as follows: If a statement containing a call on f is a CS, the return statement of f becomes a CS. If a deemed definition s_k is a CS, all definitions in $CO(s_k)$, and all statements in $\bigcup_{s_j \in CO(s_k)} DC^*_{s_j}$ become CSs. A recursive call site is always considered to be a CS.

Special provisions are needed when > 1 call site exists for a function. Consider a statement s_i, already included in a slice, which is data dependent on a deemed definition s_l. In effect, s_i is data dependent on an $s_j \in CO(s_l)$. This data dependence can be discovered during slice construction only if an occurrence of s_j precedes that of s_i in ET. However, the function may have been called a few times after this data dependence was created, hence s_j may occur after s_i in MET. Since s_j should precede s_i in SET, s_j should be included in *SCN* and must appear in CNET. This effect is incorporated by considering a deemed definition to be a critical statement if it is in $DC^*_{c\_stmt}$. Statements executed during different invocations of a function may be data dependent on one another. This effect is incorporated by assuming a loop to enclose the body of a function during instrumentation analysis of the function. Criticality analysis is now performed for statements containing references/definitions of nonlocal variables and parameters which are identical for two or more call sites.

Instrumentation analysis of the body of function f is performed as follows: IAS_f is the set of statements for which the instrumentation analysis of f should be performed. We begin by putting the return statement in IAS_f. If a deemed definition s_i is in $DC^*_{c\_stmt}$, then $CO(s_i)$ is added to IAS_f. $\forall s_j \in IAS_f$ a call on procedure *Instrumentation* is made with the criterion $(Use(s_j), s_j)$.

During program execution, a function call statement is entered in CNET when execution returns from that call. Additionally, some annotations regarding actual parameters are also entered in CNET at this time. Following changes are made in the dynamic slicing algorithm of Fig. 1 to handle function calls and function bodies: When a definition of a variable in GP located in a function body is added to *stmt_occ*, the function call statement is also added to *stmt_occ*. When a statement involving a function call is added to *stmt_occ*, the return statement of the function in also added to *stmt_occ*. When a recursive call on *Get_dynamic_slice* is made for a return statement J^p, $Use(J)$ contains all variables occurring in FE.

4.5.1. *Handling subscripted variables*

The scheme for recovering values of subscripted variables described in [7] can be extended for slicing programs using subscripted variables. Consider an assignment $a[i] := \ldots x \ldots b[j] \ldots$ situated in node N_i. To identify statements on which this statement is data dependent, we proceed as if the assignment is of the form $a := f_{a[i]}(i, \ldots x, \ldots, j, b[j])$. This approach ensures that assignments of i, x, j and $b[j]$ will be included in the slice. Assignments of the form $b[k] := \ldots$ for some k which reach node N_i should also be considered for inclusion unless it can be inferred during instrumentation time (using static analysis) that k cannot have the same value as the value of j when the assignment to $a[i]$ is executed. Criticality analysis should proceed along similar lines. Thus apart from nodes containing assignments to $b[j]$, nodes containing assignments to i, x, j and $b[k]$ may also become critical nodes.

5. Concluding remarks

Table 1 summarizes performance for some benchmark programs using the criterion (V, s_i) where V is the set of all variables used in the main program and s_i is a fictitious statement which immediately precedes the end statement in it. ET size is in terms of number of source statements executed; a **for** statement is counted as 1 even if its parts are executed separately. It is observed that SET is at least an order of magnitude smaller than CET in most cases. For a Korel–Laski style slicing approach, e.g., the algorithm of Fig. 1, part (c) of Definition 1 can be omitted since the slicing algorithm considers data and control dependences of all execution occurrences of a statement in $DC^*_{c\_stmt}$. This would reduce SET sizes further.

Table 1
Performance summary

Program			CET	#	SET
Name	Size	# Proc	size	CN	size
Towers of Hanoi$^+$	83	3	637E6	4	146
Heapsort	254	3	592E6	24	326E6
Dhrystone	400	13	108E6	4	3.99E6
Matrix Mult$^@$	537	14	870E5	5	542
SIM	1285	15	105E6	85	10.4E6

$^+$: For no of disks from 15 to 25.
$^@$: array size 350×350.
637E6: 637×10^6.

For some programs SET is several orders of magnitude smaller than CET. This behaviour is observed when none of the critical statements executes a large number of times. It provides an obvious hint to effectiveness of SET in practice: SET will be much smaller than CET if results of criticality analysis superimposed on an execution profile of a program show that none of its hotspots (i.e., most frequently executed statements) are critical statements.

Procedure *Instrumentation* requires data flow analysis to determine reaching definitions, and loop identification to implement *Same_loop*, both of which require quadratic effort in terms of program size. Procedure *Instrumentation* is itself quadratic in program size, because it checks criticality of each node once, if at all, and part (b) of Definition 1 involves path tracing whose effort is linear in program size. Hence complexity of program instrumentation is $O(|N|^2)$.

References

[1] H. Agrawal, J. Horgan, Dynamic program slicing, ACM SIGPLAN Notices 25 (6) (1990) 246–256.
[2] A.V. Aho, R. Sethi, J.D. Ullman, Compilers—Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1986.
[3] J.P. Banning, An efficient way to find the side effect of procedure calls and aliases of variables, in: Conference Record

of Sixth ACM Symposium on Principles of Programming Languages, 1979, pp. 29–41.
[4] K.D. Cooper, K. Kennedy, Interprocedural side-effect analysis in linear time, ACM SIGPLAN Notices 23 (7) (1988) 57–66.
[5] M. Copperman, J. Thomas, Poor man's watchpoints, ACM SIGPLAN Notices 30 (1) (1995) 37–44.
[6] D.M. Dhamdhere, Effective execution histories for debugging and dynamic slicing, Technical Report, CSE Department, IIT Bombay, 2000.
[7] D.M. Dhamdhere, K.V. Sankaranarayanan, Dynamic currency determination in optimized code, ACM TOPLAS 20 (6) (1998) 1111–1130.
[8] J. Ferrante, K. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM TOPLAS 9 (3) (1987) 319–349.
[9] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM TOPLAS 12 (1) (1990) 26–60.
[10] B. Korel, J. Laski, Dynamic program slicing, Inform. Process. Lett. 29 (1988) 155–163.
[11] F. Tip, A survey of program slicing techniques, J. Programming Languages 3 (3) (1995) 121–189.
[12] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.