Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information *

Rajiv Gupta and Mary Lou Soffa
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{gupta,soffa}@cs.pitt.edu

Abstract

Program slicing is an effective technique for narrowing the focus of attention to the relevant parts of a program during the debugging process. However, imprecision is a problem in static slices since they are based on all executions that reach a given program point rather than the specific execution under which the program is being debugged. Dynamic slices are precise but require a large amount of run time overhead due to the tracing information that is collected during the program's execution. We present a general approach for improving the precision and quality of static slices by incorporating dynamic information in static slicing, enabling the computation of *hybrid slices*. The technique exploits dynamic information that is readily available during debugging, namely breakpoint information and the dynamic call graph. The precision of static slicing is improved in hybrid slicing by more accurately estimating the potential paths taken by the program. The breakpoints and call/return points, used as reference points, divide the execution path into intervals. By associating each statement in the slice with an execution interval, hybrid slicing technique also improves the quality of slicing information.

1 Introduction

Slicing has proven to be a useful tool in the debugging of programs. Static slicing algorithms determine the set of program statements that may affect the computation of the value of a variable at a specified program point [4, 1, 19, 15, 10]. However, since static slices are computed under all possible executions of the program that reach a specified point, the generated slices are large and imprecise (for a particular execution), which limits their usefulness in practice. Therefore techniques that improve the precision and reduce the size of the slice are of significant interest.

One approach for improving the precision of slices is to employ dynamic slicing [2, 13, 7]. Dynamic slices are constructed for a single program execution in contrast to static slices which are constructed for all possible executions of the program. However, the construction of a dynamic slice is expensive since it requires tracing of the program's execution. Therefore we would like to develop techniques that construct accurate slices without utilizing expensive tracing techniques.

An approach to improve the efficiency of dynamic slicing focuses on the use of static information to reduce the run-time overhead by limiting the amount of tracing that is needed during execution [14, 5, 7, 16]. Static information is used to determine a subset of program points that should be traced. Although this approach does reduce the tracing effort, the size of traces can still be quite large, since the program state must be saved at the selected points.

Our approach in this paper is to improve the precision of static slices by incorporating a limited amount of dynamic information into the computation of static slices. We present the *hybrid slice*, an intraprocedural and interprocedural slicing technique, that exploits information readily available during debugging when computing slices statically. Inaccuracies in static slices result from the conservative prediction of potential control flow and data dependencies. Our technique uses dynamic information to more accurately predict control flow and thus eliminate paths that could not have been involved in the execution. The particular dynamic information exploited to re-

* Supported in part by the National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh.

fine static slices is breakpointing information and dynamic procedure call and return information. The breakpoint information consists of breakpoint positions in the code that are encountered as well as breakpoint positions that are not encountered. For interprocedural slicing, procedure calls and returns are used, including the position of the call sites.

The hybrid slicing approach improves the effectiveness of static slicing by computing more accurate and therefore smaller slices. Another important contribution of the hybrid slice is that it decomposes the overall slice into subslices. The decomposition is accomplished by using the dynamic information as reference points, thus dividing the execution path into intervals. Each statement in a slice is associated with the particular execution interval(s) in which it is found. Therefore the user can examine slices in increments and draw conclusions that cannot be inferred simply by examining an overall slice, which is the usual approach.

The data flow analysis used for the computation of hybrid slices is unique in the following two respects. In our analysis, when a dependency being sought is found (e.g., a definition corresponding to a use) after a particular execution point, it is not immediately assumed to be part of the slice. The algorithm first must ensure that the path on which the dependency is found is feasible with respect to the dynamic information. Thus, a previous execution point must be reached, where this execution point is either a previous breakpoint or procedure call or procedure entry. Also, instead of simply propagating a variable name of interest, the breakpoint or call/return point relative to an execution point is also propagated. Thus, the dynamic information and analysis technique enables the factoring of the overall slice into subslices and the exclusion of statements that would have been included if no dynamic information were available.

Intraprocedurally, hybrid slicing degenerates to static slicing if no breakpointing information is provided. It precisely predicts control flow if breakpoints are placed to capture the outcome of each predicate in the program. Hybrid slicing adapts to the user's demands since more accurate information is provided for program areas that are of most interest to the user, that is, where the user has introduced breakpoints.

Our technique uses the control flow graph as the program representation. Another representation that has been used in slicing is the program dependence graph [12, 8]. The program dependence graph is not directly applicable to the hybrid slicing technique for it represents control dependence and not control flow. In order to utilize breakpointing information during PDG slicing, a mapping would have to be provided between various points in the control flow graph and the program dependence graph.

The notion of a constrained/quasi-static slice is another approach to reduce the size of static slices [18]. An algorithm for computing constrained slices appears in [9]. Constraints on input values are provided and, using this information, a static slice is produced that excludes program executions requiring inputs that do not satisfy the given constraints. In contrast, hybrid slicing exploits dynamic information for improving the precision and quality of slicing information during debugging. The usefulness of constrained slicing for understanding legacy codes has been demonstrated in [17]. While constrained slicing is useful for program understanding, hybrid slicing is more appropriate for program debugging.

In section 2 of this paper, we describe the hybrid slicing technique for a single module that uses breakpoint information. In section 3 we describe interprocedural slicing that uses call/return information. The algorithms developed for both types of dynamic information can be used separately or combined into a hybrid slicing algorithm that uses both call/return information and breakpoint information to compute the slices. Concluding remarks are given in section 4.

2 Hybrid Slicing on a Single Program Module

Program slicing coupled with the breakpointing of a program provides an effective tool for debugging programs. A typical debugging session based upon breakpoints and slicing is characterized as follows:

- The user sets breakpoints and starts program execution.
- When a breakpoint is encountered, the user examines the values of variables at the breakpoint.
- If the values are as expected, the user resumes the program's execution. However, before resuming execution the user may disable some existing breakpoints and add new ones.
- If the values are incorrect, the user requests slicing information for selected variables to locate the potential cause of the error.

The debugging session continues in this way. However, there is no attempt during the slicing to use the occurrence of a prior breakpoint or the non-occurrence of a breakpoint to refine the slice or to provide more specific information as to whether a statement was encountered before or after a prior breakpoint. Thus, in our approach to slicing we have two primary goals. One goal is to provide the user with more accurate slices; that is, we would like to exclude statements that could not have affected the values of requested variables at the breakpoint, given the particular execution. Another goal is to split the overall slice into subslices corresponding to relevant statements that could have been executed between every successive pair of breakpoints encountered so far. This approach allows the user to follow the execution of the program as it occurred and therefore leads to a better understanding of how the values of selected variables were computed. For example, if the same statement is executed multiple times, it may be included in multiple subslices. Our approach in computing hybrid slices is to save the breakpoint positions and use them in the computation of more accurate slices as well as in the splitting of a slice into subslices. Before presenting the hybrid algorithms, we provide definitions of breakpointing history and hybrid slices.

Definition 1: The **breakpoint history** of a program execution is of the form

$BH =< (b_0, -), (b_1, N_1), ...(b_m, N_m) >$,

where b_0 is the start node of the program, b_m is the latest breakpoint encountered, b_i is the i^{th} breakpoint encountered during program execution and N_i is the set of breakpoints that were set but not encountered as the program execution proceeded from breakpoint b_{i-1} to breakpoint b_i.

Definition 2: A **slicing criteria** is of the form $SC = (V, b)$, where V is a set of variables whose values are of interest at breakpoint b.

Definition 3: For a given breakpoint history, $BH =< (b_0, -), (b_1, N_1), ...(b_m, N_m) >$, the **hybrid slice** with respect to a slicing criteria $SC = (V, b_m)$ is defined as follows:

$$HSLICE(b_0, b_m) \leftarrow \bigcup_{i=1}^{m} HSLICE(b_{i-1}, b_i),$$

where $HSLICE(b_{i-1}, b_i)$ contains those statements that were possibly executed after breakpoint b_{i-1} and prior to breakpoint b_i and their execution, directly or indirectly, influenced the computation of the value of some variable in V at b_m.

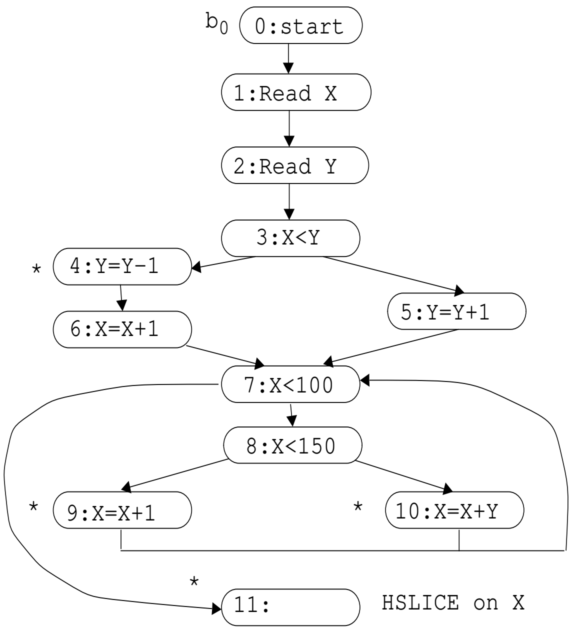
Data slices are computed by taking the transitive closure over data dependences. An executable slice can be obtained by taking the transitive closure over both control and data dependences. As the breakpoint history for a given program execution grows, so does the number of subslices computed by the hybrid slicing algorithm. The complexity of computing hybrid slices can be limited by restricting the size of the breakpoint history. It is reasonable to expect that the subslices corresponding to the recent breakpoints are of more interest to the user. Therefore the size of the history can be limited by considering the recent breakpoints and eliminating the earlier breakpoints. For example, let us assume that the complete breakpointing history under a program execution is given by $BH =< (b_0, -), (b_1, N_1), ...(b_m, N_m) >$.
We can replace this history by the following shortened breakpoint history:

$$SBH =< (b_0, -), (b_{m-\Delta-1}, -), (b_{m-\Delta}, N_{m-\Delta})..(b_m, N_m) >.$$

The shortened history only considers the most recent Δ breakpoints. The consequence of this approach is that the accuracy of the subslice corresponding to the execution from b_0 to $b_{m-\Delta-1}$ is sacrificed to limit the cost of computing hybrid slices. In fact the subslice for interval b_0 to $b_{m-\Delta-1}$ degenerates to the static slice for that interval.

The example in Figure 1 illustrates the usefulness of hybrid slices. We set breakpoints at statements $4, 9, 10$, and 11 before the program execution begins (see Figure 1a). The results of computing hybrid data slices for a number of breakpoint histories are shown in Figure 1b. We focus on the computation of the hybrid slice for variable X when the breakpoint at statement 11 is encountered. First consider the execution in which no breakpoint other than 11 was encountered (see row 1 of Figure 1b). Our algorithm is able to utilize this information to conclude that the value of X at 11 is the value that was read at statement 1 and therefore the hybrid slice contains only statement 1. On the other hand, a static data slicing algorithm will report that statements $1, 2, 4, 5, 6, 9$, and 10 are all part of the data slice. Therefore by using breakpointing history, the sizes of slices may be considerably reduced.

Let us consider the last data slice shown in Figure 1b (row 4). In this execution the overall hybrid slice contains all statements that are in the static slice except for statement 5. By using the breakpoint history, we can determine that the path through node 5 could not have been executed. By examining the subslices in the example, we obtain useful information about the execution of the program. The subslices show that the values of X and Y are initialized at statements 1 and 2 and then updated at statements 6 and 4 respectively. Next the value of X is modified at statement 9. Using this current value of X and the value of Y from statement 4, a new value of X



Breakpoints set at statements marked *
at the beginning of program execution.

(a)

Static data slice for variable X
at statement 11 = {1,2,4,5,6,9,10}.

Breakpoints Encountered	Hybrid Subslices	Hybrid Slices
b_0,b_1 0,11	HSLICE(b_0,b_1)={1}	HSLICE(b_0,b_1) {1}
b_0,b_1,b_2 0, 4,11	HSLICE(b_0,b_1)={1} HSLICE(b_1,b_2)={6}	HSLICE(b_0,b_2) {1,6}
b_0,b_1,b_2,b_3 0, 4,10,11	HSLICE(b_0,b_1)={1,2} HSLICE(b_1,b_2)={4,6} HSLICE(b_2,b_3)={10}	HSLICE(b_0,b_3) {1,2,4,6,10}
$b_0,b_1,b_2,$ 0, 4, 9, b_3,b_4,b_5 10, 9,11	HSLICE(b_0,b_1)={1,2} HSLICE(b_1,b_2)={4,6} HSLICE(b_2,b_3)={9} HSLICE(b_3,b_4)={10} HSLICE(b_4,b_5)={9}	HSLICE(b_0,b_5) {1,2,4,6,9,10}

(b)

Figure 1: Examples of Hybrid Data Slices.

is computed at statement 10. Finally at statement 9 the value of X is update again, which is then available at statement 11. It should be noted that statement 9 is encountered twice during the execution and therefore contained in two subslices. As we can see from this example, by providing subslices we allow the user to understand the flow of values including during the execution of loops. This flow of values could not have been inferred by simply examining a overall slice for a single module.

Next we develop the algorithms for computing hybrid slices by first focusing on the computation of data slices. The key problem to be solved for the computation of data slices is that of identifying those immediate data dependencies (i.e., use to a definition) that can be established along a path that is feasible under the given breakpointing history. Once this problem is solved, the resulting algorithm can be repeatedly applied for computing the transitive closure over the data dependencies.

Definition 4: Given a breakpoint history, $BH = < (b_0, -), (b_1, N_1), ...(b_m, N_m) >$, a **path** P from b_0 to b_m is **feasible** if and only if path P is composed of subpaths as follows:

$$P = PATH(b_0 \rightsquigarrow b_1).\; PATH(b_1 \rightsquigarrow b_2).\; \cdots PATH(b_{m-1} \rightsquigarrow b_m)$$

such that for $1 \leq i \leq m$,

$$NODES(PATH(b_{i-1} \rightsquigarrow b_i)) \bigcap N_i = \phi,$$

where $NODES(PATH(b_{i-1} \rightsquigarrow b_i))$ includes all nodes along the path from b_{i-1} to b_i excluding the endpoints b_{i-1} and b_i.

Let us assume that we are interested in the slice for variable v at statement s when s is encountered after b_{i-1} and before b_i. Further assume that s is reachable from the definition of variable v at statement s'. The statement s' is included in subslice $HSLICE(b_{j-1}, b_j)$, where $j \leq i$, if and only if there exists a feasible path P such that:

- the subpath $PATH(b_{j-1} \rightsquigarrow b_j)$ in P contains s'.
- the subpath from s' to s in P is definition clear with respect to variable v.

The algorithm we present computes immediate data dependencies in two steps: *detection* and *verification*. The *detection* step takes as its input, triples of the form (v, s, b_i), indicating an interest in the value of variable v immediately preceding the execution of s prior to reaching b_i and after reaching b_{i-1}. The output of the detection step is a set of pairs of the form (s', b_j) such that there is a path from s' to s through which a definition of variable v in s' reaches statement s. In addition, it is guaranteed that there is a path from breakpoints b_j and b_i. In other words the path is feasible with respect to the portion of breakpoint history $(b_j, N_j), (b_{j+1}, N_{j+1}), ..., (b_i, N_i)$. The *verification* step ensures that there is a path from b_{j-1} to b_j along which s' is encountered therefore implying that s' should be included in the slice. In order to take the transitive closure over data dependencies, new criteria are generated from the statements included in the slice and the above steps are repeated.

Our presentation of the hybrid slicing algorithm is organized into three algorithms. The algorithm *ComputeTentativeSlice* implements the *detection* step and the algorithm *ComputeActualSlice* implements the *verification* step. The algorithm *ComputeHybridSlice* in Figure 2 calls these algorithms repreatedly to compute the transitive closure over data dependencies. The input to algorithm *ComputeHybridSlice* is the breakpointing history (BH) and slicing criteria (SC). Based upon the slicing criteria a set of *triples* is generated. The immediate data dependencies corresponding to these triples are detected by *ComputeTentativeSlice* and a set of *pairs* is generated for *ComputeActualSlice*. The pairs that are successfully verified by *ComputeActualSlice* are included in the appropriate subslices. From the newly added statements to the slice, new sets of triples are generated for the transitive closure, and the above steps are repeated until no more statements are added to the slice. The algorithm *ComputeHybridSlice* also initializes the data flow sets for algorithms *ComputeTentativeSlice* and *ComputeActualSlice* to process a given set of triples and pairs respectively.

The algorithm *ComputeTentativeSlice* in Figure 3 performs backward propagation of variables whose definitions are being sought, based on the slicing criteria. The data flow sets corresponding to the entry and exit of a node n are denoted by $VAR_{en}^{all}[n]$ and $VAR_{ex}^{all}[n]$. Each variable v in a data flow set is associated with a breakpoint b_i which indicates that the search for the definition of v has progressed to a point prior to the execution of b_i. Therefore during propagation, if this variable reaches the statement that is the breakpoint b_{i-1}, then the propagated breakpoint is modified to b_{i-1} as it is propagated past breakpoint b_{i-1} (see lines 15-17). On the other hand, if the variable reaches a statement in N_i, then its propagation is discontinued since this implies that we are on an infeasible path (in line 9 $VAR_{ex}^{f}[n]$ is the feasible subset of $VAR_{ex}^{all}[n]$). Finally when variable v reaches a definition of v, we examine the associated breakpoint. If the associated breakpoint is b_j, then the statement becomes a potential candidate for inclusion in subslice $HSLICE(b_{j-1}, b_j)$.

The algorithm *ComputeActualSlice* in Figure 4 performs backward propagation of statements that are potential candidates for inclusion in the slice. The data flow sets corresponding to the entry and exit of a node n are denoted by $THSLICE_{en}^{all}[n]$ and $THSLICE_{ex}^{all}[n]$. A statement s with associated breakpoint b_i is included in subslice $HSLICE(b_{i-1}, b_i)$ if a path from b_{i-1} to statement s can be found along which no node from N_i is encountered. Therefore if the statement s reaches a statement in N_i, then its propagation is discontinued since this implies that we are on an infeasible path (in line 14, $THSLICE_{ex}^{f}[n]$ is the feasible subset of $THSLICE_{ex}^{all}[n]$). Once a statement is included in subslice $HSLICE(b_{i-1}, b_i)$ new *triples* are generated corresponding to variables referenced by the statement so that the transitive closure over data dependencies can be computed (see line 6). In case of executable slices, the control dependences must also be considered as indicated by line 7.

Next let us consider the control dependencies in the computation of executable slices. Given a statement s that has just been included in subslice $HSLICE(b_{i-1}, b_i)$, the algorithm *ComputeControlTriples* in Figure 5 describes the treatment of control dependencies involving s. An immediate control ancestor of s, say c, must be included in the hybrid slice. However, we must first determine the subslice in which c must be placed. It is possible that c may be placed in a subslice several breakpoints back as these breakpoints may have been encountered since the execution of c and prior to the execution of s. The predicate c along with breakpoint b_i is propagated backwards starting at statement s. The data flow sets corresponding to the entry and exit of a node n are denoted by $CD_{en}^{all}[n]$ and $CD_{ex}^{all}[n]$. The propagation along infeasible paths is avoided (see line 12) and breakpoints associated with the statements being propagated are appropriately modified during propagation (see lines 14-16). In each data flow set only the maximum k for which (c, b_k) is added to the set is retained, for we are interested in the latest execution of c preceding the execution of s. Finally, the data flow set at the predicated node c is examined and c is included in the subslice $HSLICE(b_{k-1}, b_k)$ such that (c, b_k) is contained in the data flow set at the entry of predicate c (see line 21). Once a statement is included in a slice the appropriate triples are generated for further processing (see line 22). The algorithm *ComputeControlTriples* calls itself recursively since we must take the transitive closure over control dependencies (see line 23).

algorithm ComputeHybridSlice ($SC = (V, b_m)$, $BH =< (b_0, -), ...(b_m, N_m) >$)
$HSLICE(b_0, b_1) \leftarrow HSLICE(b_1, b_2) \leftarrow ... \leftarrow HSLICE(b_{m-1}, b_m) \leftarrow \phi$
$triples \leftarrow \{(v, s, b_m) : v \in V, b_m \textit{ is at point immediately preceding } s\}$
repeat
    $\forall n$, $VAR_{en}^{all}[n] \leftarrow VAR_{ex}^{all}[n] \leftarrow \phi$
    – *Detection Step*
    for each triple $(v, s, b_i) \in triples$ **do**
        $VAR_{en}^{all}[s] \leftarrow VAR_{en}^{all}[s] \bigcup \{(v, b_i)\}$
        $Varlist \leftarrow Varlist \bigcup Pred(s)$
    endfor
    $pairs \leftarrow \phi$
    ComputeTentativeSlice()
    – *Verification Step*
    $\forall n$, $THSLICE_{en}^{all}[n] \leftarrow THSLICE_{ex}^{all}[n] \leftarrow \phi$
    for each pair $(s, b_i) \in pairs$ **do**
        $THSLICE_{en}^{all}[s] \leftarrow THSLICE_{en}^{all}[s] \bigcup \{(s, b_i)\}$
        $Tslicelist \leftarrow Tslicelist \bigcup \{s\}$
    endfor
    $triples \leftarrow \phi$
    ComputeActualSlice()
until $triples = \phi$
end ComputeHybridSlice

Figure 2: Overview of Hybrid Slicing Algorithm.

algorithm ComputeTentativeSlice ( )
while $Varlist \neq \phi$ **do**
    get n from the $Varlist$
    $change \leftarrow false$
    $NEWVAR \leftarrow \bigcup_{w \in Succ(n)} VAR_{en}^{all}[w]$
    if $NEWVAR \not\subseteq VAR_{ex}^{all}[n]$ **then**
        $change \leftarrow true$
        $VAR_{ex}^{all}[n] \leftarrow VAR_{ex}^{all}[n] \bigcup NEWVAR$
        $VAR_{ex}^{f}[n] \leftarrow VAR_{ex}^{all}[n] - \{(v, b_i) : n \in N_i\}$
        for each $(v, b_i) \in VAR_{ex}^{f}[n]$ *st* $v \in Def(n)$ **do**
            $pairs \leftarrow pairs \bigcup \{(n, b_i)\}$
            $VAR_{ex}^{f}[n] \leftarrow VAR_{ex}^{f}[n] - \{(n, b_i)\}$
        endfor
        $VAR_{en}^{all}[n] \leftarrow VAR_{en}^{all}[n] \bigcup VAR_{ex}^{f}[n]$
        for each $(v, b_i) \in VAR_{en}^{all}[n]$ *st* $b_{i-1} = n$ **do**
            $VAR_{en}^{all}[n] \leftarrow (VAR_{en}^{all}[n] - \{(v, b_i)\}) \bigcup \{(v, b_{i-1})\}$
        endfor
    endif
    if *change* **then** $Varlist \leftarrow Varlist \bigcup Pred(n)$ **endif**
endwhile
end ComputeTentativeSlice

Figure 3: Identification of Potential Statements in the Slice.

algorithm ComputeActualSlice ( )
  while $Tslicelist \neq \phi$ **do**
    get n from the $Tslicelist$
    for each $(s, b_i) \in THSLICE_{en}^{all}[n]$ and $b_{i-1} = n$ **do**
      $HSLICE(b_{i-1}, b_i) \leftarrow HSLICE(b_{i-1}, b_i) \bigcup \{s\}$
      $triples \leftarrow triples \bigcup \{(v, s, b_i) : v \in Ref(s)\}$
      if executable slice required **then** ComputeControlTriples($s \in HSLICE(b_{i-1}, b_i)$) **endif**
      $THSLICE_{en}^{all}[n] \leftarrow THSLICE_{en}^{all}[n] - \{(s, b_i)\}$
    endfor
    for each predecessor $p \in Pred(n)$ **do**
      $NEWTHSLICE \leftarrow \bigcup_{w \in Succ(p)} THSLICE_{en}^{all}[w]$
      if $NEWTHSLICE \not\subseteq THSLICE_{ex}^{all}[p]$ **then**
        $THSLICE_{ex}^{all}[p] \leftarrow THSLICE_{ex}^{all}[p] \bigcup NEWTHSLICE$
        $THSLICE_{ex}^{f}[p] \leftarrow THSLICE_{ex}^{all}[p] - \{(s, b_i) : s \in N_i\}$
        $THSLICE_{en}^{all}[p] \leftarrow THSLICE_{en}^{all}[p] \bigcup THSLICE_{ex}^{f}[p]$
        $Tslicelist \leftarrow Tslicelist \bigcup \{p\}$
      endif
    endfor
  endwhile
end ComputeActualSlice

Figure 4: Identification of Actual Statements in the Slice.

algorithm ComputeControlTriples ($s \in HSLICE(b_{i-1}, b_i)$)
  $\forall n,\ CD_{ex}^{all}[n] \leftarrow CD_{en}^{all}[n] \leftarrow \phi$
  for each immediate control predecessor c of s **do**
    $CD_{ex}^{all}[s] \leftarrow CD_{ex}^{all}[s] \bigcup \{(c, b_i)\}$
  endfor
  $Cdlist \leftarrow \{s\}$
  while $Cdlist \neq \phi$ **do**
    get n from $Cdlist$
    $NEWCD \leftarrow \{(c, b_{max}) : j \leq max, \forall (c, b_j) \in CD_{en}^{all}[w],\ where\ w \in Succ(n)\}$
    if $NEWCD \not\subseteq CD_{ex}^{all}[n]$ **then**
      $CD_{ex}^{all}[n] \leftarrow CD_{ex}^{all}[n] \bigcup NEWCD$
      $CD_{ex}^{f}[n] \leftarrow CD_{ex}^{all}[n] - \{(c, b_i) : n \in N_i\}$
      $CD_{en}^{all}[n] \leftarrow CD_{en}^{all} \bigcup CD_{ex}^{f}[n]$
      for each $(c, b_i) \in CD_{en}^{all}[n]\ st\ b_{i-1} = n$ **do**
        $CD_{en}^{all}[n] \leftarrow (CD_{en}^{all}[n] - \{(c, b_i)\}) \bigcup \{(c, b_{i-1})\}$
      endfor
      $Cdlist \leftarrow Cdlist \bigcup Pred(n)$
    endif
  endwhile
  Let $(c, b_k) \in CD_{en}^{all}[c]$:
  $HSLICE(b_{k-1}, b_k) \leftarrow HSLICE(b_{k-1}, b_k) \bigcup \{c\}$
  $triples \leftarrow triples \bigcup \{(v, c, b_k) : v \in Ref(c)\}$
  ComputeControlTriples($c \in HSLICE(b_{k-1}, b_k)$)
end ComputeControlTriples

Figure 5: Considering Control Dependences.

```
b0,b1,b2,b3
0, 4,10,11
```

```
Input triples = {(x,11,b3)}
Output pairs = {(1,b3),(6,b3),(10,b3)}

Input pairs = {(1,b3),(6,b3),(10,b3)}
HSLICE(b2,b3) = {10}
Output triples = {(X,10,b2),(Y,10,b2)}

Input triples = {(X,10,b2),(Y,10,b2)}
Output pairs = {(1,b2),(6,b2),(4,b2),(5,b2)}

Input pairs = {(1,b2),(6,b2),(4,b2),(5,b2)}
HSLICE(b1,b2) = {4,6}
Output triples = {(X,6,b1),(Y,4,b1)}

Input triples = {(X,6,b1),(Y,4,b1)}
Output pairs = {(1,b1),(2,b1)}

Input pairs = {(1,b1),(2,b1)}
Output triples = {}
HSLICE(b0,b1) = {1,2}
```

Figure 6: An Example Illustrating Hybrid Slicing Algorithms.

Figure 6 illustrates the computation of a hybrid data slice for the third breakpoint history of the example in Figure 1. The computation of the data slice takes three iterations. The pairs/triples resulting from each invocation of the detection/verification step are shown in Figure 6. In the first detection step starting from statement 11 definitions of X in statements 1, 6 and 10 are identified and during the verification step statement 10 is included in the slice while statements 1 and 6 are discarded. Statement 10 uses variables X and Y whose definitions are sought in the next iteration. Definitions in statements 1, 4, 5 and 6 are detected and during the verification step statements 4 and 6 are included in the slice. The statements 6 and 4 reference variables X and Y respectively and in the final iteration the definitions in statements 1 and 2 are included in the slice.

Next we illustrate the treatment of control dependencies. Let us consider the last breakpoint history for which the hybrid data slice is shown in Figure 1b. Consider the invocation $ComputeControlTriples(9 \in HSLICE(b_4, b_5))$. Based upon these results predicate 8, which is the immediate control predecessor of 9, is included in subslice $HSLICE(b_3, b_4)$. The predicate 8 is included in subslice $HSLICE(b_3, b_4)$ because it was executed during the period of execution that correspond to this subslice. Once predicate 8 has been included, the algorithm is recursively invoked to compute the transitive closure over control dependences. Also triples are generated for later processing for variables referenced in 8 to take the closure over data dependencies.

Complexity Analysis of Data Slicing: For the purpose of this analysis we assume that the program contains V variables, N statements (and the number of statements is of the same order as the number of edges), and d is maximum depth of loop nests. We further assume that B is the number of breakpoints in the history. During the detection step the maximum size of a data flow set is $B \times V$. Thus, a single operation on the data flow set requires at most $O(B \times V)$ time. Each node may be examined $MAX(d+1, B)$ times until the data flow stablizes. Thus the cost of the detection step is bounded by $O(MAX(d+1, B) \times N \times B \times V)$. The size of a data flow set during the verification step is bounded by $N \times B$ and the number of times each node may be examined is bounded by $d+1$. Thus the cost of the verification step is bounded by $O((d+1) \times N^2 \times B)$. The total cost of the two steps is therefore given by $O(MAX(d+1, B) \times N^2 \times B)$. Since typically d is a small number it is reasonable to assume that $MAX(d+1, B)$ is equal to B. Thus, the worst case cost of the two steps can be considered to be $O(B^2 \times N^2)$. Since the size of a hybrid slice is bounded by $B \times N$, the maximum number of times the detection and verification steps are repeated is also bounded by this value. Therefore the total cost of computing data hybrid slices is $O(B^3 \times N^3)$. The user is able to limit B to a small number by constructing a shortened history.

3 Interprocedural Hybrid Slicing

Conceptually the hybrid slicing algorithm using breakpoint history can be extended to interprocedural hybrid slicing. However, this straightforward extension could not adequately handle the calling contexts of procedures and thus would introduce a new source of imprecision in the slice. The interprocedural hybrid slicing that we present uses the dynamic call and return information to correctly handle the calling context problem. A combination of breakpoints and call/return information can also be used in the definition of the interprocedural hybrid slice. However to simplify the presentation of the algorithms in this section, we focus on the use of only call/return information to define the interprocedural hybrid slice. In this case, construction of the hybrid slice is guided by the procedure calls and returns. As was the case for breakpoint hybrid slices, the call/return hybrid slices provide more precise information than static slices and provide subslices that give more detailed information as to where in the call/return sequence a particular statement may have been executed.

We first present definitions based on the use of call/return information and then give the algorithms, which are similar to those for breakpoint hybrid slicing. Our algorithms handle local and global variables and reference parameters.

Definition 5: The **call history** of a program execution is of the form $CH =< CR_0, CR_1, ..., CR_m >$, where CR_i is either a procedure call or a return from a procedure and CR_0 is assumed to be a call to start execution of the main program. The forms of call and return are:

- $CALL[P_{caller} \rightarrow P_{callee}\ at\ s]$: indicates that procedure P_{caller} calls procedure P_{callee} at statement s.
- $RET[P_{callee} \rightarrow P_{caller}\ at\ s]$: indicates a return from P_{callee} to P_{caller} at statement s (call site of P_{callee} in P_{caller}).

Definition 6: For a calling history, $CH =< CR_0, CR_1, ..., CR_m >$, the overall **hybrid slice** with respect to a slicing criteria $SC = (V, CR_m)$ is defined as follows:

$$HSLICE(CR_m) \leftarrow \bigcup_{i=1}^{m} HSSLICE(CR_{i-1}, CR_i)$$

where subslice $HSSLICE(CR_{i-1}, CR_i)$ contains those statements that were possibly executed after CR_{i-1} and before CR_i and their execution, directly or indirectly, influenced the computation of the value of some variable in V at CR_m.

During propagation of data flow information across procedure boundaries the mapping from actuals to formals across a call site and the reverse mapping from formals to actuals at procedure entry are needed. The bindings are obtained by examining the appropriate call site.

Again, we concentrate on data slices in the description of the call/return hybrid slice. Consider the example program given in Figure 7, which consists of a main program and four procedures. Assume the call/return history and the slicing criteria of variable d at statement 28 as given in Figure 7. If static slices are obtained using Weiser's interprocedural slicing technique [19], which does not take into account the calling contexts of the calls, the following slice is computed: $\{2, 3, 6, 11, 12, 15, 16, 20, 27, 30\}$. If a static slice is computed using the calling context [12, 14], more precise information can be found. Using the calling context, only the call site related to the call is processed. Thus, statement 6 would not be included as part of the slice, as P_2 in statement 7 is not a possible call site. Thus, statement 3 defining y would not be included in the slice. Using the calling context, the slice consists of the following statements: $\{2, 11, 12, 15, 16, 20, 27, 30\}$.

Using our technique, we are able to compute the precise slice: $\{2, 11, 12, 16, 20, 27\}$. In the hybrid slice, statement 15 is not part of the slice as this statement is not executed in the above calling sequence for the example program. The path containing the call to P_2 at statement 14 is the path that is executed. Likewise, all of the statements in procedure P_4 would not be included, as the path where the call to P_4 is found is not executed (statement 23). Both of these conditions can be determined by using the dynamic call/return information.

As was the case in the hybrid slicing using breakpoints, we can also construct subslices that indicate where in the call/return history a particular statement could be executed. The subslices computed by our algorithm for the example are also given in Figure 7. Using the subslices, we see that statement 16 appeared on the slice between CR_5 and CR_6. Thus, the assignment statement is executed after P_2 returns to P_1 at call site 14 but before P_1 calls P_3 at statement 17. Statement 27 appears in two subslices. It was executed after P_2 called P_3 at call site 22 but before P_3 returned to P_2, that is, between CR_3 and CR_4. It was also on the execution path after P_1 called P_3 at call site 17 but before the breakpoint in P_3, that is, between CR_6 and CR_7. This type of detailed information could help the user pinpoint the occurrence of definitions of variables when debugging.

The technique to compute the hybrid slice using call/return history closely follows that of the breakpoint hybrid slices. However, instead of breakpoints, calls and returns, including the call sites, are used.

Given a call/return history, we define a feasible path as follows:

Definition 7: Given a call/return history, CH=$< CR_0, CR_1, ..., CR_m >$, a **path** from CR_0 to CR_m is **feasible** if and only if path P is composed of the following subpaths:

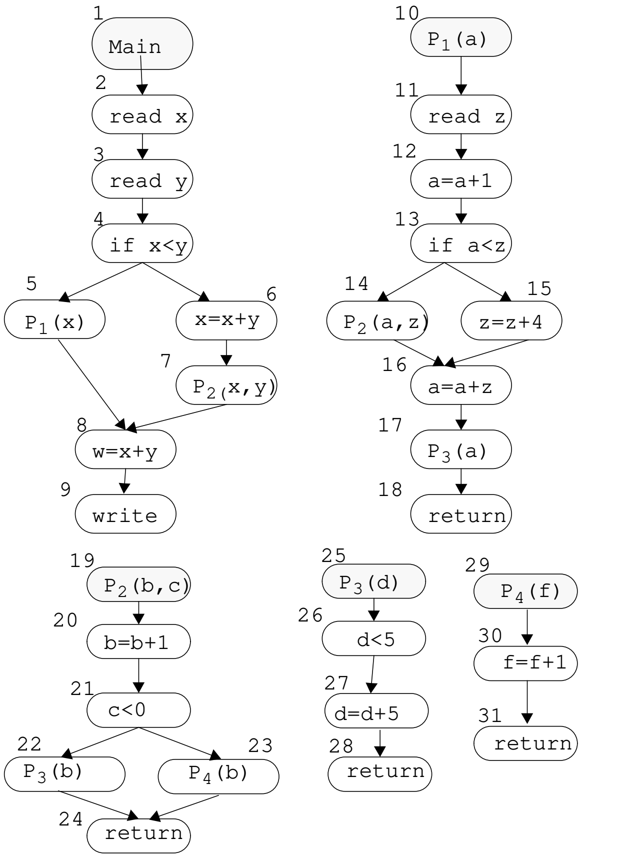
$$P = PATH(CR_0 \rightsquigarrow CR_1).$$
$$PATH(CR_1 \rightsquigarrow CR_2).$$
$$\cdots PATH(CR_{m-1} \rightsquigarrow CR_m),$$

such that for $1 \leq i \leq m$,

- if CR_{i-1} is a call and CR_i is a call then $P^{i-1}_{callee} = P^i_{caller}$;
- if CR_{i-1} is a call and CR_i is a return then $P^{i-1}_{caller} = P^i_{callee}$ and $P^{i-1}_{callee} = P^i_{caller}$ and call site $s_{i-1} = s_i$;
- if CR_{i-1} is a return and CR_i is a call then $P^{i-1}_{caller} = P^i_{caller}$; and
- if CR_{i-1} is a return and CR_i is a return then $P^{i-1}_{callee} = P^i_{caller}$.

The overall algorithm for interprocedural slicing, *ComputeInterHybridSlice*, uses the phases *ComputeInterTentativeSlice* and *ComputeInterActualSlice* similar to the phases used in the computation of the hybrid slice using breakpoint history hybrid slice, as are the same data flow sets used. However, in the hybrid interprocedural slicing algorithms, the points of interest are procedure call, entry and exit points where propagation starts or terminates. In the remainder of the section we provide a brief overview of the algorithms. More detailed algorithms can be found in [11].

The algorithm, *ComputeInterTentativeSlice* propagates variables and call/return history points backwards until either an entry of the current procedure P is reached, a call to a procedure is reached, or an assignment to a variable being sliced is encountered. If a call statement is reached, then the call/return history is checked to see if this call matches a return in the history point. If so, variables are bound to the formals in the called procedure, and if the variable being sliced is involved, then the VAR set at the end of the procedure being called is updated and the appropriate nodes from the procedure are placed on the list to slice. If the node encountered is an entry node of a procedure, the call/return history is checked to determine if this procedure was called at the appropriate call history point. If so, the variables being sliced are bound by a reverse binding of formals to actuals and propagation continues at the call point, as determined by the call site information in the call/return history. When a statement is encountered that defines a variable being sliced, a pair is generated and propagation for this variable terminates. A pair consists of a statement and the call/return history point. If a variable in the criteria in not defined in a procedure, propagation continues in the calling procedure and the history point is updated to include a previous call return point, that is $CR_i \rightarrow CR_{i-1}$. The



Call/Return History:

<CR_0: The main program M is called>
<CR_1: M calls P_1 at 5 >
<CR_2: P_1 calls P_2 at 14 >
<CR_3: P_2 calls P_3 at 22 >
<CR_4: P_3 returns P_2 at 22 >
<CR_5: P_2 returns P_1 at 14 >
<CR_6: P_1 calls P_3 at 17>
<CR_7: P_3 calls breakpoint at 28>

Slicing Criteria:variable d at statement 28

Hybrid Slice,HSLICE(CR_7)={2,11,12,16,20,27}

HSLICE (CR_0,CR_1) = {2},
HSLICE (CR_1,CR_2) = {11,12},
HSLICE (CR_2,CR_3) = {20},
HSLICE (CR_3,CR_4) = {27},
HSLICE (CR_5,CR_6) = {16},
HSLICE (CR_6,CR_7) = {27}

Static Slice (without calling context)
={2,3,6,11,12,15,16,20,27,30}

Static Slice (with calling context)
= {2,11,12,15,16,20,27,30}

Figure 7: Examples of Interprocedural Hybrid Data Slices.

call/return history is checked to determine the appropriate point whenever a pair is propagated over a call site. That is, if the slicing variable is reference parameter that is not included in a particular call, say to P_x, then the slicing is not done on P_x. The CR point in the pair has to be updated to reflect the number of procedure calls and returns that are being skipped. The history can be checked to match the calls and returns that are found in the history until the correct point is found.

In the procedure *ComputeInterActualSlice*, entry and call nodes are again treated as nodes of interest. The pairs, each consisting of a statement and the call/return history point, which were found in the *ComputeInterTentativeSlice* algorithm, are propagated until it can be determined whether the statement reaches the next valid call/return point and should be included in the slice. When the statement reaches a call or entry point, the call/return history is checked to determine if the node reached is on a valid path. If so, the statement is added to the slice and triples are generated for any variables used in the statement being added. Included in the slice is the call/return point that the statement was encountered.

Consider the example given in Figure 7 with the criteria (d at statement 28) and the call and return history given in the example. The definition of d in statement 27 is identified as being a potential statement in the slice associated with the call to P_3 from P_1 at statement 17. In the algorithm *ComputeInterActualSlice*, statement 27 along CR_7 is propagated until the entry of procedure P_3 is reached. The call/return history is consulted and the call is identified as being on a valid path so the statement is put in the slice. The reference to d in the statement is bound to the actual parameter a at the call site (statement 17) and this a, along with the previous call/return point becomes the slicing criteria in the next phase with CR_6 being the history point. Slicing continues from statement 17 using *ComputeInterTentativeSlice*, finding statement 16, to be a potential statement to be added to the slice. The *ComputeInterActualslice* determines if statement 16 is on a valid path. Finding a call to P_2 in node 14, which matches the history, it puts statement 16 in the slice. Slicing continues with *ComputeInterTentativeSlice*, with the transitive closure of variables in this statement, which are a and z. When *ComputeInterTentativeSlice* takes the path through node 15 and propagates the statement 15 defining z, it finds the entry node. A check of the call/return history finds this is an invalid path and statement 15 is not included in the slice. The algorithms continue in this manner, finally finding the subslices given in Figure 7.

The hybrid interprocedural algorithm can be extended to include aliases caused by reference parameters. The computation of the alias sets is performed before slicing begins [6]. The alias sets for a procedure are placed on the program nodes for that procedure. As a variable name is propagated in search of its definition, we include in the slice any definition of an alias as well as any definition of the variable itself.

Another extension to this algorithm that can be considered is the utilization of interprocedural hybrid slicing algorithms in conjunction with static interprocedural slicing algorithms to allow the consideration of shortened call histories. The call histories can be shortened by eliminating subgraphs corresponding to periods of program execution that are of little interest to the user.

In the interprocedural hybrid slicing algorithm presented the only dynamic information that was used was the call/return history. However, the breakpoint history can also be used in conjunction with call/return history to further improve the precision of the interprocedural slice. The form of the combined history and slice follows.

Definition 8: The **combined history** of a program execution is of the form $\mathcal{H} =< H_0, H_1, ...H_m >$, where H_i is one of the following: Breakpoint: (b_i, N_i); Procedure Call: $CALL[P_{caller} \rightarrow P_{callee}\ at\ s]$; or Procedure Return: $RET[P_{callee} \rightarrow P_{caller}\ at\ s]$.

Definition 9: For a given combined history, $\mathcal{H} =< H_0, H_1, ...H_m >$, the overall **hybrid slice** with respect to a slicing criteria $SC = (V, CR_m)$ is defined as follows:

$$HSLICE(H_m) \leftarrow \bigcup_{i=0}^{m-1} HSSLICE(H_i, H_{i+1})$$

where subslice $HSSLICE(H_i, H_{i+1})$ contains those statements that were possibly executed after H_i and before H_{i+1} and their execution, directly or indirectly, influenced the computation of the value of some variable in V at CR_m.

With this combined history the breakpoint and call/return algorithms can be easily integrated. The data flow sets as before carry either a breakpoint reference or a call/return reference.

4 Conclusions

We have presented the notion of a hybrid slice that utilizes dynamic information readily available during debugging to improve the effectiveness of static slices. Typically a debugger provides a facility to do breakpointing and to trace procedure calls and returns. In order to implement the hybrid slicing, the breakpoint and caller/return history must be saved. This is the only extra run time cost associated with the hybrid slice that is not typically incurred with debugging. Thus, our technique requires little additional tracing at run time.

The value of the hybrid slice is that it improves the accuracy of the static slice and also provides a more detailed analysis about the statements in the slice. This information enables the user to better identify where values are computed. An important aspect of the hybrid slice is that the detailed information is guided by the user's breakpoints. Thus, the information presented to the user more clearly focuses on the areas of the program that are of interest to the user. Purely static slices can be produced if no breakpoints are placed and control flow is precisely predicted if breakpoints are placed at every control predicate.

References

[1] H. Agrawal, "On Slicing Programs with Jump Statements," *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 302-312, June 1994.

[2] H. Agrawal and B. Horgan, "Dynamic Program Slicing," *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 246-256, 1990.

[3] A. Aho, R. Sethi, and J. Ullman, *Compiler Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, MA, 1986.

[4] J-D. Choi and J. Ferrante, "Static Slicing in the Presence of Goto Statements," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 4, pages 1097-1113, July 1994.

[5] J-D. Choi, B. Miller, and R. Netzer, "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 13, N. 4, pages 491-530, 1991.

[6] K. Cooper, "Analyzing Aliases of Reference Formal Parameters," *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, pages 281-290, 1985.

[7] E. Duesterwald, R. Gupta, and M.L. Soffa, "Distributed Slicing and Partial Re-execution for Distributed Programs," *Fifth Workshop on Languages and Compilers for Parallel Computing, LNCS 757 Springer Verlag*, pages 497-511, Yale University, New Haven, Connecticut, August 1992.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and its use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pages 319-349, July 1987.

[9] J. Field, G. Ramalingam, and F. Tip, "Parametric Program Slicing," *Proc. ACM Symposium on Principles of Programming Languages*, pages 379-392, January 1995.

[10] R. Gupta and M.L. Soffa, "A Framework for Partial Data Flow Analysis," *International Conference on Software Maintenance*, Victoria, British Columbia, pages 4-13, September 1994.

[11] R. Gupta and M.L. Soffa, "Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information," *Technical Report TR-95-30*, University of Pittsburgh, Pittsburgh, June 1995.

[12] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pages 26-60, January 1990.

[13] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, Vol. 29, pages 155-163, Oct. 1988.

[14] M. Kamkar, O. Fritzson, and N. Shahmehri, "Three Approaches to Interprocedural Dynamic Slicing," *Microprocessing and Microprogramming 38*, pages 625-636, 1993.

[15] J.R. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing," *Proc. Second IEEE Symposium on Computers and Applications*, pages 877-883, June 1987.

[16] R. Netzer and M. Weaver, "Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs," *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 313-325, June 1994.

[17] J. Ning, A. Engberts, and W. Kozaczynski, "Automated Support for Legacy Code Understanding," *Communications of the ACM*, Vol. 37, No. 3, pages 50-57, 1994.

[18] A. Venkatesh and C.N. Fischer, "SPARE: A Development Environment for Program Analysis Algorithms," *IEEE Transactions on Software Engineering*, Vol. 18, No. 4, April 1992.

[19] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pages 352-357, July 1984.