Hybrid Slicing: Integrating Dynamic Information with Static Analysis *

Rajiv Gupta, Mary Lou Soffa, and John Howard
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 possible executions that reach a given program point rather than the specific execution under which the program is being debugged. Dynamic slices, based on the specific exeuction being debugged, are precise but incur high run time overhead due to the tracing information that is collected during the program's execution. We present a hybrid slicing technique that integrates dynamic information from a specific execution into a static slice analysis. The *hybrid slice* produced is more precise than the static slice and less costly than the dynamic slice. The technique exploits dynamic information that is readily available during debugging, namely breakpoint information and the dynamic call graph. This information is integrated into a static slicing analysis to more accurately estimate 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 provides information as to when a statement was encountered during execution. Another attractive feature of our approach is that it allows the user to control the cost of hybrid slicing by limiting the amount of dynamic information used in computing the slice. We implemented the hybrid slicing technique to demonstrate the feasibility of our approach.

Keywords - static slice, dynamic slice, hybrid slice, breakpoint, dynamic call graph.

* Supported in part by the National Science Foundation Presidential Young Investigator Award CCR-9157371, Grant CCR-9109089 and Grant CCR-9402226 to the University of Pittsburgh. Preliminary version of this paper will appear in the Third Symposium on the Foundations of Software Engineering, October 1995.

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 [1, 5, 12, 17, 23]. 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 static slice are of significant interest.

One approach for improving the precision of static slices is to employ dynamic slicing [2, 9, 14]. A dynamic slice is constructed for a fixed input (i.e., for a specific program execution) in contrast to a static slice which makes no assumptions about the input. However, the construction of a dynamic slice is expensive since it requires tracing of the program's execution. Therefore techniques that reduce the expense of dyamic slices without sacrificing too much of the precision are desirable.

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 [16, 6, 9, 4, 19]. Information about the program 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.

Our approach in this paper is the development of a *hybrid slicing* technique that integrates a limited amount of dynamic information into a static slicing analysis. The hybrid slice is computed both intraprocedurally and interprocedurally by exploiting information readily available during debugging. 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 some of the paths that could not have been involved in the specific execution. The particular dynamic information exploited 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 code position of the call sites.

The hybrid slicing approach improves on the precision of static slicing by computing more accurate and therefore smaller slices. The hybrid slice decomposes the overall slice into subslices 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 static slice, which is the usual approach. Another attractive feature of our approach is that it allows the user to control the cost of slicing by limiting the amount of dynamic information provided to the hybrid slicing algorithm.

The static slicing analysis technique 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), 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. In order to do so, instead of simply propagating a variable name of interest, the breakpoint or call/return point relative to an execution point is also propagated. The dynamic information and analysis technique also enable 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.

We implemented the hybrid slicing technique at both the intraprocedural and interprocedural levels to demonstrate its practicality and effectiveness. We found that the hybrid slice produced is smaller than the static slice and does reduce in size as more breakpoints are added. We also found that there is no measurable difference in execution time between computing static slices and hybrid slices.

Our technique uses the control flow graph as the program representation. Another representation that has been used in slicing is the program dependence graph [10, 13]. 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.

Dynamic slicing techniques [2, 9, 14] can be used to compute precise slices since accurate control and data flow information can be saved during program execution. Relevant data dependences among statements involving only scalar variables can be accurately determined if the exact program path taken during the execution is known. On the other hand accurate identification of data dependences in the presence of pointers and arrays requires tracing all read and write operations. Since the breakpoint history used during hybrid slicing only enables us to predict the execution path taken during execution with greater accuracy, its main impact is the improved precision of static slicing for statements involving scalars. In contrast dynamic slicing can also accurately slice programs in presence of pointers and arrays.

Other approaches to hybrid slicing have been proposed [6, 8, 9, 16, 19]. These approaches use static information to improve the execution time performance of dynamic slicing while maintaining the precision of dynamic slicing. In one approach, dependencies that can be computed statically are computed before dynamic slicing and utilized at run time. Dependencies involving array elements or pointers that must be computed at run time are computed at execution time and recorded in a trace [8, 9, 16]. Another approach records only a small amount of trace information during program execution. During debugging, these coarse grained traces are supplemented with static information to generate fine-grained traces [6, 19].

The notion of a constrained/quasi-static slice is an approach to reduce the size of static slices [22]. An algorithm for computing constrained slices appears in [11]. 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 integrates dynamic information for improving the precision and cost of slicing during debugging. The usefulness of constrained slicing for understanding legacy codes has been demonstrated in [20]. While constrained slicing is useful for program understanding, hybrid slicing is more appropriate for program debugging.

A related operation to slicing is chopping [15, 21], which computes the elements that cause a program point to have an effect on another program point. It thus provides a more focused approach than slicing for determining the effects of a program point on another. Our approach of using subslices also provides more focused slices according to the program's execution history.

In section 2 of this paper, we describe the hybrid slicing technique for a single program 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. An implementation is briefly described in section 4 and concluding remarks are given in section 5.

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 the 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 in computing 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.

In order to define the form of the breakpoint history, we view the program execution as consisting of a series of intervals where each interval represents the execution between two successive breakpoints encountered during the execution. Consider an execution interval during which a set of breakpoints B was active (i.e., set). Further assume that the interval was terminated when a breakpoint $b \in B$ was encountered. The history associated with this execution interval is a pair of the form $(b, N = B - \{b\})$, where b is the breakpoint encountered at the end of the execution interval and N is the set of breakpoints that were active but not encountered during the interval. The overall breakpoint history is composed of the breakpoint histories of individual execution intervals. The overall hybrid slice is composed of the hybrid subslices corresponding to the execution intervals. The computation of a hybrid slice is initiated with respect to values of variables at the most recent breakpoint. A more precise definition of the breakpoint history and hybrid slices follows.

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

$$BH = < (b_0, \phi), (b_1, N_1 = B_1 - \{b_1\}), ...(b_m, N_m = B_m - \{b_m\}) >,$$

where b_0 is the start node of the program, b_m is the latest breakpoint encountered, B_i is the set of breakpoints that were active during the execution interval from breakpoint b_{i-1} to b_i, and N_i is the set of breakpoints that were active but not encountered during the execution interval from b_{i-1} to 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, $BH =< (b_0, \phi), (b_1, N_1), ...(b_m, N_m) >$, 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. The statements that influence the computation of a variable are computed by taking the transitive closure over both data and control dependencies. Data slices are computed by taking the transitive closure over only data dependences.

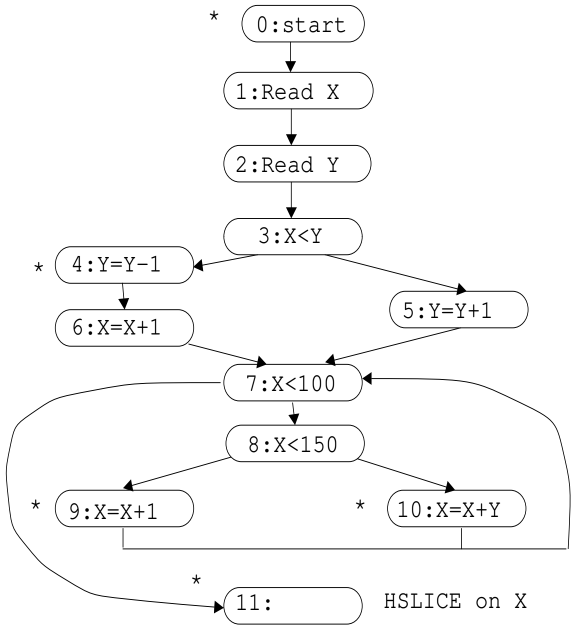
As the breakpoint history for a given program execution grows, so does the number of subslices computed by the hybrid slicing algorithm. The complexity of computing hybrid slices can be limited by restricting the size of the breakpoint history. It is reasonable to expect that the subslices corresponding to the recent breakpoints are of more interest to the user than the earlier ones. 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 $BH =< (b_0, \phi), (b_1, N_1), ...(b_m, N_m) >$. We can replace this history by the following shortened breakpoint history:

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

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 breakpoints in set $N_{m-\Delta-1}$ of the shortened history include all those breakpoints that were set and never encountered for the program's execution up to breakpoint $b_{m-\Delta-1}$.

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 can 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



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
0,11	HSLICE(0,11)={1}	HSLICE(0,11) {1}
0, 4,11	HSLICE(0,4)={1} HSLICE(4,11)={6}	HSLICE(0,11) {1,6}
0, 4,10,11	HSLICE(0,4)={1,2} HSLICE(4,10)={4,6} HSLICE(10,11)={10}	HSLICE(0,11) {1,2,4,6,10}
0, 4, 9, 10, 9,11	HSLICE(0,4)={1,2} HSLICE(4,9)={4,6} HSLICE(9,10)={9} HSLICE(10,9)={10} HSLICE(9,11)={9}	HSLICE(0,11) {1,2,4,6,9,10}

(b)

Figure 1: Examples of Hybrid Data Slices.

value of X 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 2 subslices. As we can see from this example, by providing subslices we allow the user to understand the flow of values including those values computed during the execution of loops. This flow of values could not have been inferred by simply examining an overall slice for a single module.

Let us now illustrate the effect of shortening the histories on the quality of slicing information. Let us assume that the last history in Figure 1(b) has been shortened to only include the two latest breakpoints, that is, Δ is set to two and the breakpoints 4, 9 and 10 are no longer part of the history. Using the shortened history we obtain the subslices $HSLICE(9,11) = \{9\}$ and $HSLICE(0,9) = \{1,2,4,5,6,9,10\}$. As we can see, the subslice $HSLICE(9,11)$ is the same as it was for the complete history. However, the slice $HSLICE(0,9)$ has increased in size since an additional statement, statement 5, has been included in the subslice. Thus, the precision of the subslice corresponding to the period of execution over which the history is not maintained is reduced, while the precision of the subslice for the period of execution over which the history is maintained is not affected.

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. Intuitively, a path is said to be feasible with respect to a breakpoint history if it visits the encountered breakpoints in the appropriate order and it does not visit the breakpoints that were not encountered in the corresponding execution intervals. In the computation of static slices all paths to a point are considered feasible. In contrast, during hybrid slicing only a subset of such paths are considered feasible. Once an algorithm for identifying immediate dependences has been developed, it can be repeatedly applied for computing the transitive closure over the data dependencies.

Definition 4: Given a breakpoint history, $BH = < (b_0, \phi), (b_1, N_1), ...(b_m, N_m) >$, 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 during execution. 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', and
- 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 initial input triples of the form (v, s, b_m), indicating an interest in the value of variable v immediately preceeding statement s, which is where the latest breakpoint b_m occurred. In the process of taking the closure over data dependencies, a triple (v, s, b_i) is considered if variable v is used by statement s and statement s has been included in the hybrid subslice $HSLICE(b_{i-1}, b_i)$. 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, this 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.

algorithm ComputeHybridSlice ($SC = (V, b_m)$, $BH =< (b_0, \phi), ...(b_m, N_m) >$)
  $HSLICE(b_0, b_1) \leftarrow HSLICE(b_1, b_2) \leftarrow ... \leftarrow HSLICE(b_{m-1}, b_m) \leftarrow \phi$
  $triples \leftarrow \{(v, s, b_m) : v \in V, b_m \textit{ is at point immediately preceding } s\}$
  repeat
    $Detection\ \ Step:$
    Initialize data flow sets with appropriate (variable,breakpoint) pairs.
    $\forall n,\ VAR^{all}_{en}[n] \leftarrow VAR^{all}_{ex}[n] \leftarrow \phi$
    for each triple $(v, s, b_i) \in triples$ **do**
      $VAR^{all}_{en}[s] \leftarrow VAR^{all}_{en}[s] \bigcup \{(v, b_i)\}$
      $Varlist \leftarrow Varlist \bigcup Pred(s)$
    endfor
    Identify statements that may potentially belong to the hybrid slice.
    $pairs \leftarrow \phi$
    ComputeTentativeSlice()
    $Verification\ \ Step:$
    Initialize data flow sets with appropriate (statement,breakpoint) pairs.
    $\forall n,\ THSLICE^{all}_{en}[n] \leftarrow THSLICE^{all}_{ex}[n] \leftarrow \phi$
    for each pair $(s, b_i) \in pairs$ **do**
      $THSLICE^{all}_{en}[s] \leftarrow THSLICE^{all}_{en}[s] \bigcup \{(s, b_i)\}$
      $Tslicelist \leftarrow Tslicelist \bigcup \{s\}$
    endfor
    Identify statements for inclusion in the hybrid slice.
    $triples \leftarrow \phi$
    ComputeActualSlice()
  until $triples = \phi$
end ComputeHybridSlice

Figure 2: Overview of Hybrid Slicing Algorithm.

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 the path is infeasible (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 $HSLICE(b_{j-1}, b_j)$.

1.**algorithm** ComputeTentativeSlice ( )
2. **while** $Varlist \neq \phi$ **do**
3. get n from the $Varlist$
 Propagate (variable,breakpoint) pairs backward through n.
4. $change \leftarrow false$
5. $NEWVAR \leftarrow \bigcup_{w \in Succ(n)} VAR_{en}^{all}[w]$
6. **if** $NEWVAR \not\subseteq VAR_{ex}^{all}[n]$ **then**
7. $change \leftarrow true$
8. $VAR_{ex}^{all}[n] \leftarrow VAR_{ex}^{all}[n] \bigcup NEWVAR$
  Remove pairs (v, b_i) st n could not have been encountered after b_{i-1} and
   before b_i according to the breakpoint history.
9. $VAR_{ex}^{f}[n] \leftarrow VAR_{ex}^{all}[n] - \{(v, b_i) : n \in N_i - \{b_{i-1}\}\}$
10.   **for each** $(v, b_i) \in VAR_{ex}^{f}[n]$ st $v \in Def(n)$ **do**
   Immediate data dependency found;
   create (statement,breakpoint) pair for the verification step.
11. $pairs \leftarrow pairs \bigcup \{(n, b_i)\}$
12. $VAR_{ex}^{f}[n] \leftarrow VAR_{ex}^{f}[n] - \{(v, b_i)\}$
13.   **endfor**
14. $VAR_{en}^{all}[n] \leftarrow VAR_{en}^{all}[n] \bigcup VAR_{ex}^{f}[n]$
15.   **for each** $(v, b_i) \in VAR_{en}^{all}[n]$ st $b_{i-1} = n$ **do**
   Modify (variable,breakpoint) pairs;
   replace (v, b_i) by (v, b_{i-1}) if n is the b_{i-1}-th breakpoint.
16. $VAR_{en}^{all}[n] \leftarrow (VAR_{en}^{all}[n] - \{(v, b_i)\}) \bigcup \{(v, b_{i-1})\}$
17.   **endfor**
18. **endif**
19. **if** $change$ **then** $Varlist \leftarrow Varlist \bigcup Pred(n)$ **endif**
20. **endwhile**
21.**end ComputeTentativeSlice**

Figure 3: Identification of Potential Statements in the Slice.

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, $THSLICE^{f}_{ex}[n]$ is the feasible subset of $THSLICE^{all}_{ex}[n]$). 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).

1.**algorithm** ComputeActualSlice ( )
2. **while** $Tslicelist \neq \phi$ **do**
3. get n from the $Tslicelist$
Propagate (statement,breakpoint) pairs backward through n.
4. **for each** $(s, b_i) \in THSLICE^{all}_{en}[n]$ and $b_{i-1} = n$ **do**
Since b_{i-1} has been reached, include s in the hybrid subslice $Hybrid(b_{i-1}, b_i)$.
5. $HSLICE(b_{i-1}, b_i) \leftarrow HSLICE(b_{i-1}, b_i) \bigcup \{s\}$
Generate new triples for taking closure over data dependencies.
6. **if** $s = b_{i-1}$ **then** $triples \leftarrow triples \bigcup \{(v, s, b_{i-1}) : v \in Ref(s)\}$
7. **else** $triples \leftarrow triples \bigcup \{(v, s, b_i) : v \in Ref(s)\}$ **endif**
8. **if** closure over control dependencies is also required **then**
ComputeControlTriples($s \in HSLICE(b_{i-1}, b_i)$)
endif
Discontinue propagation of processed (statement,breakpoint) pairs.
9. $THSLICE^{all}_{en}[n] \leftarrow THSLICE^{all}_{en}[n] - \{(s, b_i)\}$
10. **endfor**
11. **for each** predecessor $p \in Pred(n)$ **do**
12. **if** $THSLICE^{all}_{en}[n] \not\subseteq THSLICE^{all}_{ex}[p]$ **then**
13. $THSLICE^{all}_{ex}[p] \leftarrow THSLICE^{all}_{ex}[p] \bigcup THSLICE^{all}_{en}[n]$
Eliminate (statement,breakpoint) pairs inconsistent with breakpoint history.
14. $THSLICE^{f}_{ex}[p] \leftarrow THSLICE^{all}_{ex}[p] - \{(s, b_i) : n \in N_i\}$
15. $THSLICE^{all}_{en}[p] \leftarrow THSLICE^{all}_{en}[p] \bigcup THSLICE^{f}_{ex}[p]$
16. $Tslicelist \leftarrow Tslicelist \bigcup \{p\}$
17. **endif**
18. **endfor**
19. **endwhile**
20.**end ComputeActualSlice**

Figure 4: Identification of Actual Statements in the Slice.

Next let us consider the control dependencies in the computation of 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; that is, s is control dependent on c. 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 $CD^{all}_{en}[n]$ and $CD^{all}_{ex}[n]$. 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).

1.**algorithm** ComputeControlTriples ($s \in HSLICE(b_{i-1}, b_i)$)
2. $\forall n,\ CD^{all}_{ex}[n] \leftarrow CD^{all}_{en}[n] \leftarrow \phi$
3. **for each** immediate control predecessor c of s **do**
4. $\quad CD^{all}_{ex}[s] \leftarrow CD^{all}_{ex}[s] \bigcup \{(c, b_i)\}$
5. **endfor**
6. $Cdlist \leftarrow \{s\}$
7. **while** $Cdlist \neq \phi$ **do**
8. $\quad$ get n from $Cdlist$
$\quad$ Consider only the latest execution of c during propagation.
9. $\quad NEWCD \leftarrow \{(c, b_{max}) : j \leq max, \forall (c, b_j) \in CD^{all}_{en}[w],\ where\ w \in Succ(n)\}$
10. $\quad$ **if** $NEWCD \not\subseteq CD^{all}_{ex}[n]$ **then**
11. $\quad\quad CD^{all}_{ex}[n] \leftarrow CD^{all}_{ex}[n] \bigcup NEWCD$
$\quad\quad$ Eliminate (predicate,breakpoint) pairs inconsistent with breakpoint history.
12. $\quad\quad CD^{f}_{ex}[n] \leftarrow CD^{all}_{ex}[n] - \{(c, b_i) : n \in N_i\}$
13. $\quad\quad CD^{all}_{en}[n] \leftarrow CD^{all}_{en} \bigcup CD^{f}_{ex}[n]$
14. $\quad\quad$ **for each** $(c, b_i) \in CD^{all}_{en}[n]\ st\ b_{i-1} = n$ **do**
$\quad\quad\quad$ Modify (predicate,breakpoint) pairs;
$\quad\quad\quad$ replace (c, b_i) by (c, b_{i-1}) if n is the b_{i-1}-th breakpoint.
15. $\quad\quad\quad CD^{all}_{en}[n] \leftarrow (CD^{all}_{en}[n] - \{(c, b_i)\}) \bigcup \{(c, b_{i-1})\}$
16. $\quad\quad$ **endfor**
17. $\quad\quad Cdlist \leftarrow Cdlist \bigcup Pred(n)$
18. $\quad$ **endif**
19. **endwhile**
Include predictes in appropriate subslices.
20. Let $(c, b_k) \in CD^{all}_{en}[c]$:
21. $HSLICE(b_{k-1}, b_k) \leftarrow HSLICE(b_{k-1}, b_k) \bigcup \{c\}$
Generate new triples for taking closure over data dependencies.
22. $triples \leftarrow triples \bigcup \{(v, c, b_k) : v \in Ref(c)\}$
Take closure over control dependencies.
23. ComputeControlTriples($c \in HSLICE(b_{k-1}, b_k)$)
24.**end ComputeControlTriples**

Figure 5: Considering Control Dependencies.

Figures 6 and 7 illustrate 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 $HSLICE(b_3, b_4)$ (see Figure 8). The predicate 8 is included in subslice $HSLICE(b_3, b_4)$ be-cause it was executed during the period of execution that corresponds 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 $O(MAX(d+1, B) \times N \times B \times V)$. Since the number of source variables is typically smaller than the number of source statements, the time for the detection step is bounded by $O((d+1) \times N^2 \times B)$. 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 $O(B^3 \times N^3)$.

From the above analysis it is clear that the complexity of hybrid slicing increases with the length of breakpoint history. Thus, the ability of our approach to allow the use of shortened histories is significant. On the other hand the complexity of dynamic slicing cannot be controlled in the above fashion. In order to obtain precise dynamic slices and subslices for programs with only scalar references, the dynamic trace of all conditional branches must be kept so that program's execution path is completely known. In contrast, during hybrid slicing, the breakpoint history is likely to contain a small subset of recently executed conditional branches. The computation of precise dynamic slices in presence of pointers and arrays would require a significantly larger overhead for tracing read and write operations by all statements. Thus, while dynamic slicing enables computation of precise slices, the run-time complexity of computing dynamic slices is significantly higher than hybrid slicing.

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, reference parameters and aliases.

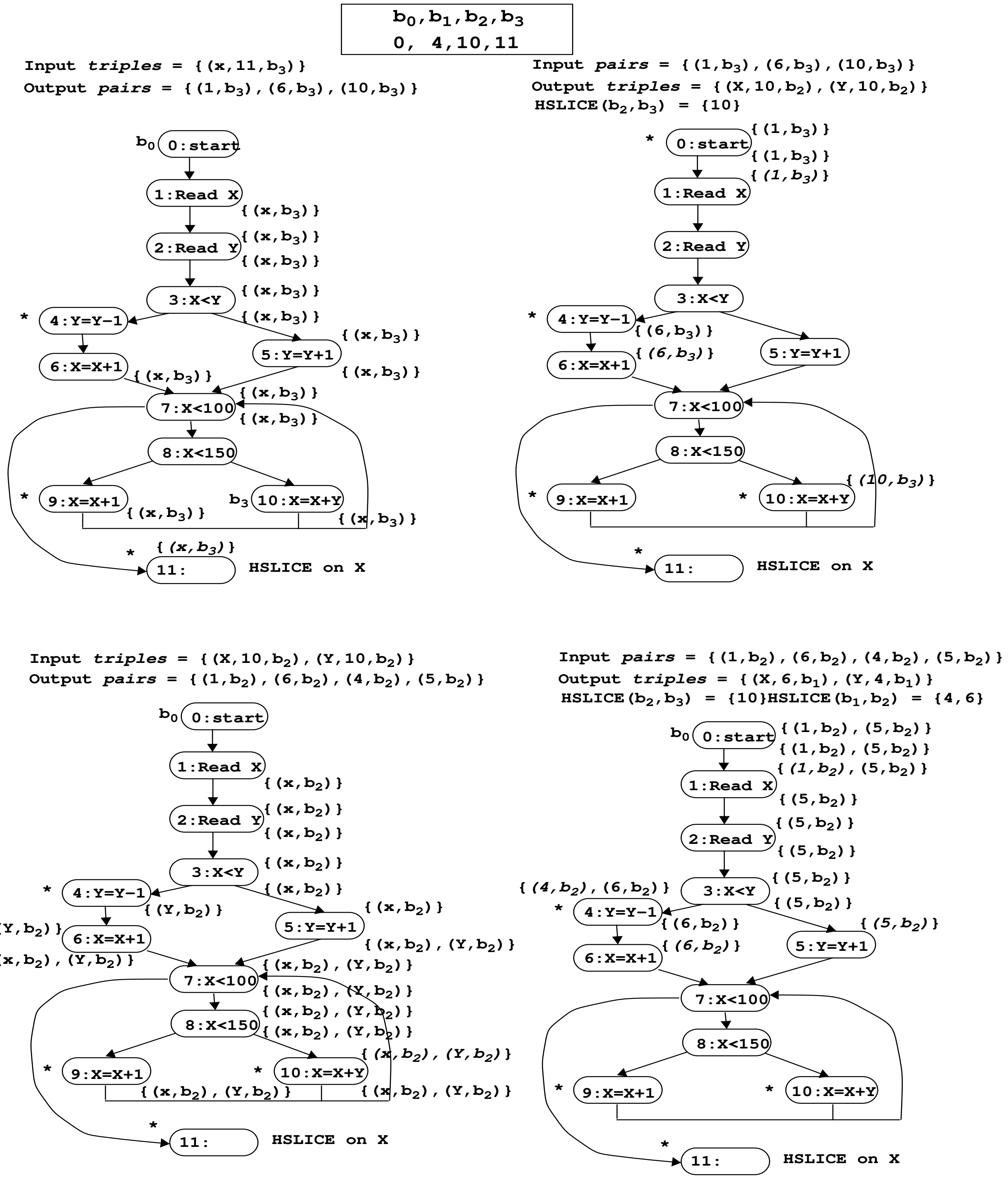


Figure 6: Detailed Example.

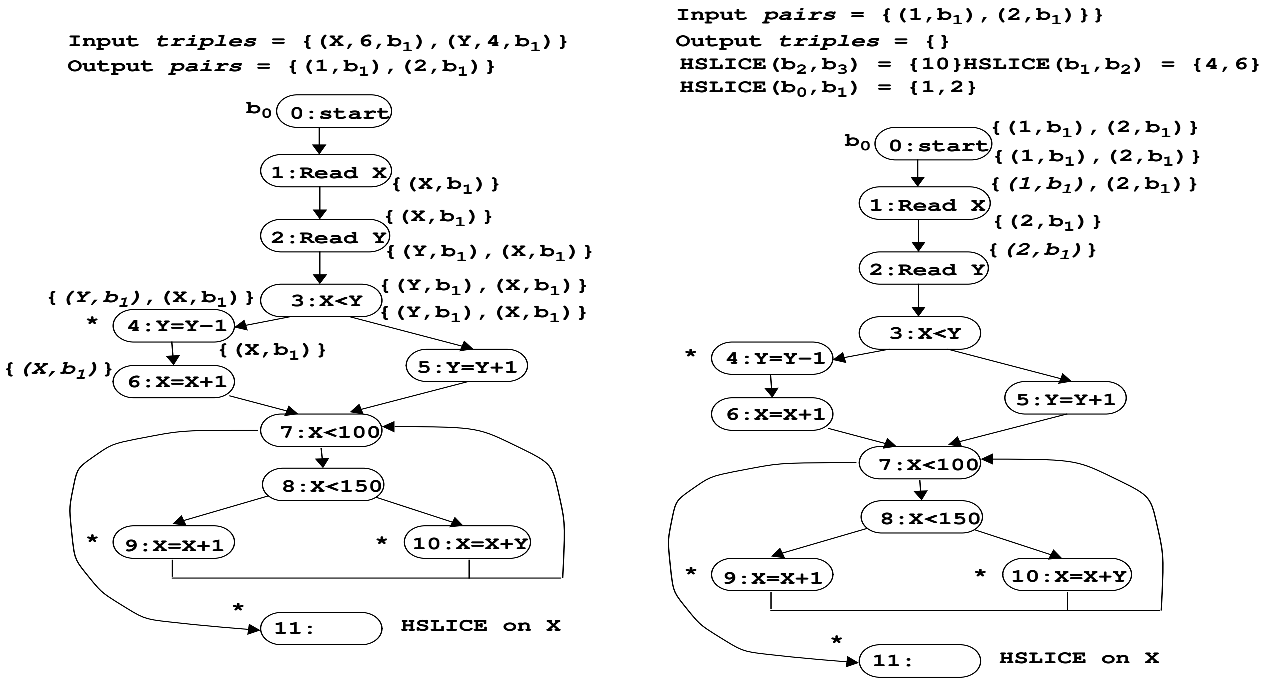


Figure 7: Detailed Example Continued.

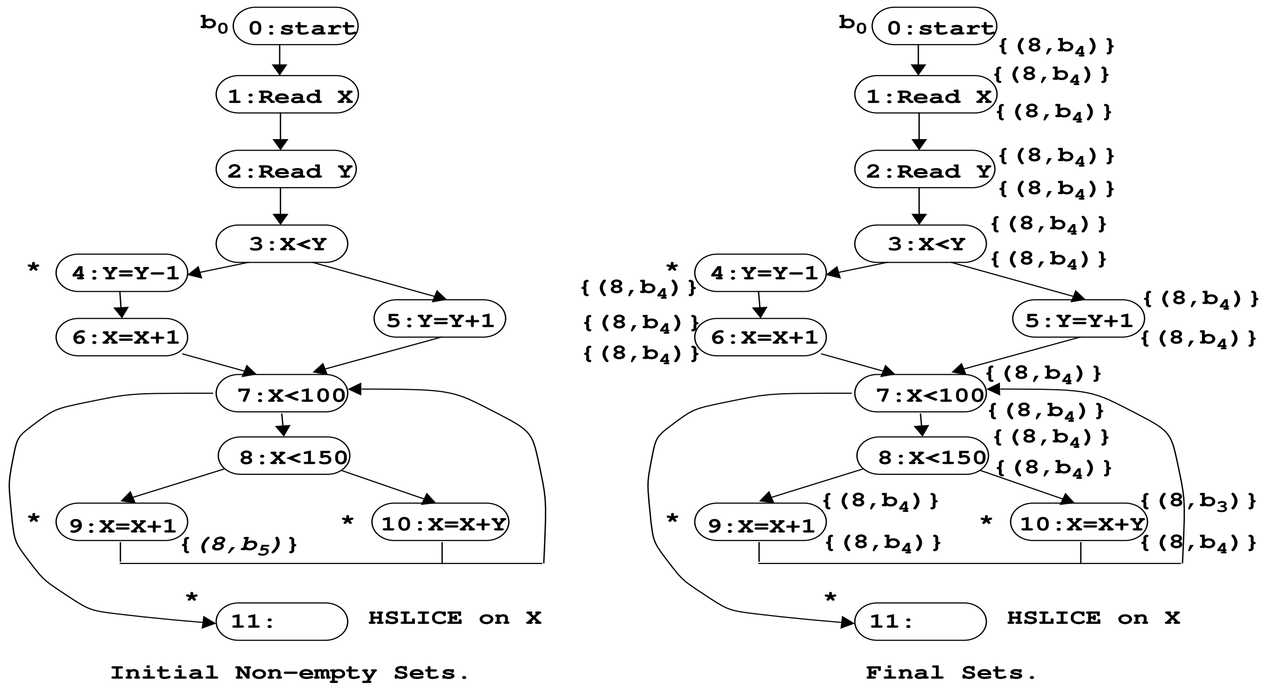


Figure 8: Example of Processing Control Dependencies.

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:

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

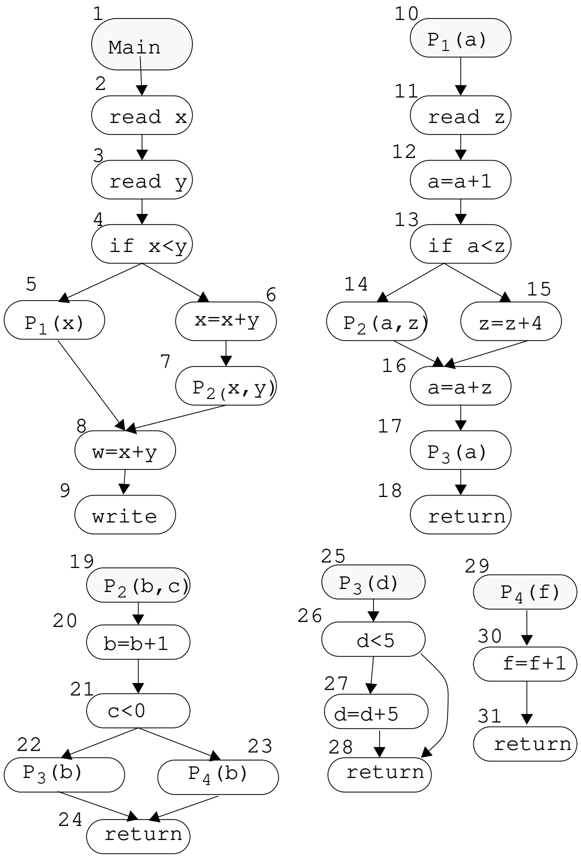
where subslice $HSSLICE(CR_{i-1}, CR_i)$ contains those statements that were possibly executed after CR_{i-1} and before CR_i and their execution, directly or indirectly, influenced the computation of the value of some variable in V at CR_m.

During the 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. If $bind$ or $bind^{-1}$ is called with a variable that is not an actual argument or formal parameter, the function returns that variable (used for globals).

Again, we concentrate on data slices in the description of the call/return hybrid slice. Consider the example program given in Figure 9, 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 9. If static slices are obtained using Weiser's interprocedural slicing technique [23], 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 [13, 16], 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 a 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 9. 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 during debugging.



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 9: Examples of Interprocedural Hybrid Data Slices.

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. Initially we do not consider parameter aliasing, but later we show how it can be incorporated.

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:

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

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* which are 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. The detailed algorithms are presented in Figures 10, 11 and 12.

The algorithm, *ComputeInterHybridSlice*, given in Figure 10, iteratively calls algorithms *ComputeInterTentativeSlice* and *ComputeInterActualSlice* to compute the hybrid slice. The algorithm, *ComputeInterTentativeSlice*, given in Figure 11 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 for slicing. 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 $CR_i \rightarrow CR_{i-1}$. *Findhistory* on line 13 in the algorithm is a routine that checks the call/return history 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 program point is found.

algorithm ComputeInterHybridSlice $(SC(V, CR_m), CH =< CR_0, CR_1, ..., CR_m >)$
  $HSLICE(CR_0, CR_1) \leftarrow HSLICE(CR_1, CR_2) \leftarrow ... \leftarrow HSLICE(CR_{m-1}, CR_m) \leftarrow \phi$
  $triples \leftarrow \{(v, s, CR_m) : v \in V, CR_m \; is \; at \; point \; immediately \; preceding \; s\}$
  repeat
    $Detection \;\; Step :$
    Initialize data flow sets with appropriate (variable,call-return) pairs.
    $\forall \; n \; \in \; P_i, VAR_{en}^{all}[n] \leftarrow VAR_{ex}^{all}[n] \leftarrow \phi$
    for each triple $(v, s, CR_i) \in triples$ **do**
      $VAR_{en}^{all}[s] \leftarrow VAR_{en}^{all}[s] \bigcup \{(v, CR_i)\}$
      $Varlist \leftarrow Varlist \bigcup Pred(s)$
    endfor
    Identify statements that may potentially belong to the hybrid slice.
    $pairs \leftarrow \phi$
    ComputeTentativeSlice()
    $Verification \;\; Step :$
    Initialize data flow sets with appropriate (statement,call-return) pairs.
    $\forall \; n \; \in \; P_i, THSLICE_{en}^{all}[n] \leftarrow THSLICE_{ex}^{all}[n] \leftarrow \phi$
    for each pair $(s, CR_i) \in pairs$ **do**
      $THSLICE_{en}^{all}[s] \leftarrow THSLICE_{en}^{all}[s] \bigcup \{(s, CR_i)\}$
      $Tslicelist \leftarrow Tslicelist \bigcup \{s\}$
    endfor
    Identify statements for inclusion in the hybrid slice.
    $triples \leftarrow \phi$
    ComputeActualSlice()
  until $triples = \phi$
end ComputeInterHybridSlice

Figure 10: Overview of the Interprocedural Hybrid Slicing Algorithm.

algorithm ComputeInterTentativeSlice ()
while $Varlist \neq \phi$ **do**
get n from $Varlist$
$NEWVAR \leftarrow \bigcup_{w \in Succ(n)} VAR_{en}^{all}[w]$
if $NEWVAR \not\subseteq VAR_{ex}^{all}[n]$ **then**
$VAR_{ex}^{all}[n] \leftarrow VAR_{ex}^{all}[n] \bigcup NEWVAR$
if n is a call statement to P_{fr} **then**
for each $(v, CR_i) \in VAR_{ex}^{all}[n]$ st $CR_{i-1} = Return\ from\ P_{fr}\ to\ P_{current}\ at\ n$ **do**
Propagate (variable,call-return) pairs to the exit of the callee,
if callee is consistent with call-return history.
if $v \in$ actuals of P_{fr} or global variables **then**
$VAR_{en}^{all}[exit\ of\ P_{fr}] \leftarrow VAR_{en}^{all}[exit\ of\ P_{fr}] \bigcup \{(bind.v, CR_{i-1})\}$
$Varlist \leftarrow Varlist \bigcup Pred(exit\ of\ P_{fr})$
else
$VAR_{en}^{all}[n] \leftarrow VAR_{en}^{all}[n] \bigcup \{(v, findhistory(CR_i)\}$
$Varlist \leftarrow Varlist \bigcup Pred(n)$
endif
endfor
elseif n is the *entry* node of the procedure $P_{current}$ **then**
for each $(v, CR_i) \in VAR_{ex}^{all}[n]$ st $CR_{i-1} = Call\ P_{fr}\ to\ P_{current}\ at\ s$ **do**
if $v \in$ formals of $P_{current}$ or global variables **then**
Propagate (variable,call-return) pairs to the call site in the caller,
if the caller is consistent with call-return history.
$VAR_{en}^{all}[s] \leftarrow VAR_{en}^{all}[s] \bigcup \{(bind^{-1}.v, CR_{i-1})\}$
$Varlist \leftarrow Varlist \bigcup Pred(s)$
endif
endfor
else
$VAR_{en}^{all}[n] \leftarrow VAR_{en}^{all}[n] \bigcup VAR_{ex}^{all}[n]$
for each $(v, CR_i) \in VAR_{ex}^{all}[n]\ st\ v \in Def(n)$ **do**
Immediate data dependency found;
create (statement,call-return) pair for the verification step.
$pairs \leftarrow pairs \bigcup \{(n, CR_i)\}$
$VAR_{en}^{all}[n] \leftarrow VAR_{en}^{all}[n] - \{(n, CR_i)\}$
endfor
$Varlist \leftarrow Varlist \bigcup Pred(n)$
endif
endif
endwhile
endComputeInterTentativeSlice

Figure 11: Identification of Potential Statements in the Interprocedural Slice.

In the algorithm *ComputeInterActualSlice*, entry and call nodes are again treated as nodes of interest (see Figure 12). 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 feasible 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.

algorithm ComputeInterActualSlice ( )
while $Tslicelist \neq \phi$ **do**
  get n from the $Tslicelist$
  if n is a call statement to P_{fr} **then**
    for each $(s, CR_i) \in THSLICE_{en}^{all}[n]$ st $CR_{i-1} = Return\ from\ P_{fr}\ to\ P_{current}\ at\ n$ **do**
      Since CR_{i-1} has been reached, include s in the hybrid subslice $HSLICE(CR_{i-1}, CR_i)$.
      $HSLICE(CR_{i-1}, CR_i) \leftarrow HSLICE(CR_{i-1}, CR_i) \bigcup \{s\}$
      Generate new triples for taking closure over data dependencies.
      $triples \leftarrow triples \bigcup \{(bind.v, end\ of\ P_{fr}, CR_{i-1}) : v \in Ref(s)\}$
    endfor
  elseif n is the *entry* node of $P_{current}$ **then**
    for each $(s, CR_i) \in THSLICE_{en}^{all}[n]$ st $CR_{i-1} = Call\ P_{fr}\ to\ P_{current}\ at\ q$ **do**
      Since CR_{i-1} has been reached, include s in the hybrid subslice $HSLICE(CR_{i-1}, CR_i)$.
      $HSLICE(CR_{i-1}, CR_i) \leftarrow HSLICE(CR_{i-1}, CR_i) \bigcup \{s\}$
      Generate new triples for taking closure over data dependencies.
      $triples \leftarrow triples \bigcup \{(bind^{-1}.v, q, CR_{i-1}) : v \in Ref(s)\}$
    endfor
  else
    for each predecessor $p \in Pred(n)$ **do**
    Propagate (statement,call-return) pairs backwards through p.
      $NEWTHSLICE \leftarrow \bigcup_{w \in Succ(p)} THSLICE_{en}^{all}[w]$
      if $NEWTHSLICE \not\subseteq THSLICE_{ex}^{all}[p]$ **then**
        $THSLICE_{ex}^{all}[p] \leftarrow THSLICE_{ex}^{all}[p] \bigcup NEWTHSLICE$
        $Tslicelist \leftarrow Tslicelist \bigcup \{p\}$
      endif
    endfor
  endif
endwhile
end ComputeInterActualSlice

Figure 12: Identification of Actual Statements in the Interprocedural Slice.

Consider the example given in Figure 9 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 with 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, CR_6, becomes the slicing criteria in the next phase. 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 that 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 9.

The hybrid interprocedural algorithm can be extended to include aliases caused by reference parameters and global variables. The computation of the alias sets is performed before slicing begins [7]. 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.

Through the utilization of interprocedural hybrid slicing algorithms in conjunction with static interprocedural slicing algorithms, we can also allow for shortened call histories. A call history can be shortened by maintaining only the part of the call graph that corresponds to a recent period of program execution which is of interest to the user. To illustrate this approach let us consider the history of the example in Figure 9. Let us assume that the call graph history is shortened to only include CR_6, that is, history from CR_1 through CR_5 is excluded. Using our hybrid slicing algorithm we compute the subslice $HSLICE(CR_6, CR_7) = \{27\}$. In addition, we determine that the value of d at the start of P_3 is of interest. For the slicing criteria of d at the start of P_3, we compute a static slice using an existing static slicing algorithm. For example, let us assume that we use Weiser's interprocedural static slicing algorithm [23] which does not consider the calling context. The result of this algorithm would be: $SLICE(CR_0, CR_6) = \{2, 3, 6, 11, 12, 15, 16, 20, 27, 30\}$. Thus, by sacrificing the precision and quality of slicing information for earlier parts of the execution we can reduce the cost of computing hybrid slices.

Complexity Analysis

The complexity of the interprocedural hybrid slicing algorithm is equivalent to computing an intraprocedural hybrid slice on an expanded flow graph which is obtained by replacing each procedure call in a caller by the callee's control flow graph. The breakpoints correspond to the beginning and ending points of each called procedure. Let N_{max} be the number of nodes in the control flow graph of the largest program module and let C be the length of the call graph history, that is, the number of calls and returns. The size of the expanded flow graph is bounded by $C \times N_{max}$ and the number of breakpoints is bounded by C. In the previous section, we showed that the complexity for intraprocedural hybrid slicing is bounded by $O(B^3 \times N^3)$. Thus, the complexity of interprocedural hybrid slicing is given by $O(C^3 \times (C \times N_{max})^3)$ or $O(C^6 \times N_{max}^3)$. The user can control the cost of slicing by limiting the size of the history and thus the value of C to a small number.

Combined Histories

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 integrated. The data flow sets as before carry either a breakpoint reference or a call/return reference.

4 Implementation

We have implemented both the intraprocedural and interprocedural hybrid slicing algorithms to illustrate the feasibility of computing hybrid slices. The implementation incorporates the hybrid slicing algorithms in the Unravel [18] static slicing tool. Unravel, developed at the National Institute of Standards and Technology, was designed to assist in the evaluation of software written in ANSI C. We extended the Unravel tool to accept both breakpoint and call-return histories. These histories are entered either interactively or through an input file containing the histories.

Figure 13 shows the interface between the tool and the user and highlights the statements in the hybrid slice. The breakpoint history is also displayed. During debugging, breakpoints at statements 6, 33, and 45 were set. During execution, breakpoints at statements 6 and 45 were encountered while the breakpoint at statement 33 was not encountered. At the breakpoint at statement 45 a hybrid slice on variable *bill* was requested. The computed hybrid slice contains statements 18, 24, 31, 40, 42, and 44 in the procedure *main*. If a static slice at statement 45 would have been requested, statements 34 and 36 would also have been included.

To illustrate the potential of hybrid slices in reducing the number of statements from the static slice, hybrid slices were computed for a number of small programs. Table 1 shows the results of computing intraprocedural hybrid slices using breakpoint history. The programs include *Heapsort*, procedures *Paintface* and *ClockShowDate* taken from the *Clock* program, procedures *ResolvedGet* and *SlicePass* taken from the static slicing program in Unravel, and procedure *ComputeTentativeSlice* taken from our implementation of the intraprocedural hybrid slicing algorithm. The second column gives the number of lines of code in the program modules. In the third column of Table 1, the range of the number of breakpoints that were set is given and column four contains the range of the number of statements included in the hybrid slice. In these experiments, all breakpoints except the first and last were placed on a statement following a conditional statement. Execution of the hybrid slicing algorithm with zero breakpoints corresponds to the static slice and resulted in the inclusion of the maximum number of statements in the slice. It can be observed from the table that as the number of breakpoints increased, the size of the hybrid slice decreased due to the greater accuracy with the path taken by the program is predicted. In general the decrease in the size of a hybrid slice depends upon the placement of breakpoints as well as the variable and the program point with respect to which the slice is requested. The results in Table 1 illustrate that at least for some slicing requests and breakpoint placements the decrease in the size of the hybrid slice can be significant, ranging from about 30% to over 50% for these requests.

Table 2 gives results after running the interprocedural hybrid slicing algorithm on complete programs. The programs include *Heapsort*, the *Hybrid* slicing algorithm, the *Queens* program, and the *Clinpack* routines. The

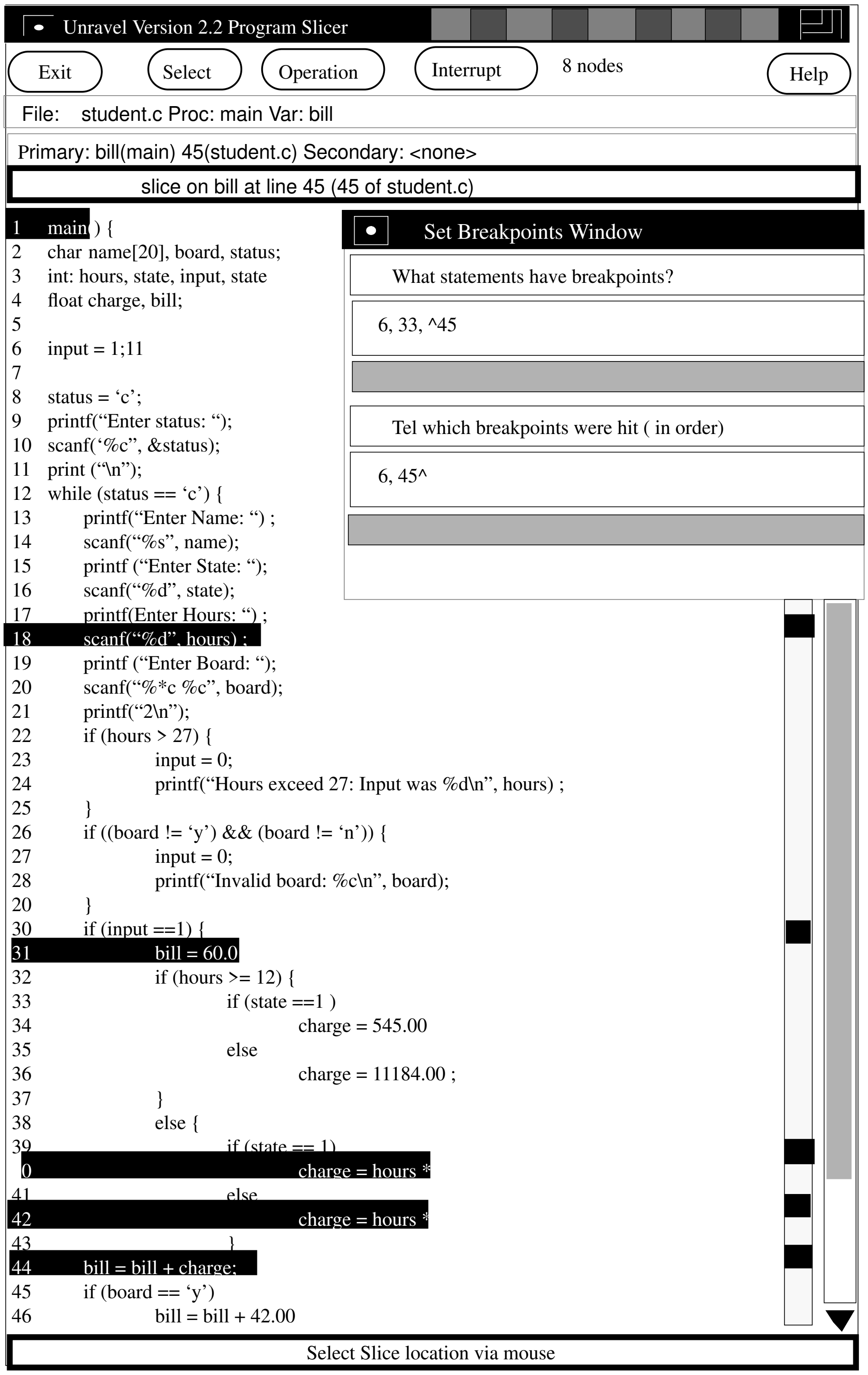


Figure 13: User Interface of the Hybrid Slicer.

Program	Lines of Code	Number of Breakpoints	Slice Size
Paintface	30	0-3	8-5
ClockShowDate	49	0-4	15-7
ResolvedGet	50	0-4	12-6
SlicePass	47	0-3	9-5
ComputeTentativeSlice	89	0-2	6-4

Figure 14: Intraprocedural Hybrid Slicing.

program sizes are given in the second column of the table. The slices were computed for various call-return histories and variables. The third column contains the number of statements that occur in the static slice and the last column gives the number of statements in the interprocedural hybrid slice. As was the case with the intraprocedural slices, the number of statements in the interprocedural hybrids was significantly less than the corresponding static slices. The reductions ranged from 0% to 60%. Thus these results illustrate that there are slicing requests that greatly benefit from the availability of call-return histories.

Program	Lines of Code	Static Slice	Hybrid Slice
Heapsort	225	58	37
		58	49
		58	40
Hybrid	688	10	4
Queens	89	17	12
		15	13
Clinpack	256	18	18
		50	24

Figure 15: Interprocedural Hybrid Slicing.

The above results illustrate that for some slicing requests the hybrid slices are significantly smaller than static slices and hence more precise. During the above experiments we observed that there was no measurable difference in the execution times of static and hybrid slicing algorithms. Our experience with hybrid slicing so far indicates that the proposed hybrid slicing algorithms can be implemented with a reasonable amount of effort and that hybrid slicing is a promising approach for improving the precision of static slices. However, to quantify the efficiency and precision of hybrid slicing more extensive experimentation is required.

5 Concluding Remarks

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 and the static slicing tool must have access to this history. 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. Hybrid slicing also allows the user to control the cost of hybrid slicing by controlling the size of history.

ACKNOWLEDGMENT

We thank the reviewers for helpful comments and suggestions that led to improvements in the paper's content and presentation.

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, JUNE 1990.

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

[4] T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 4, pages 1319-1360, July 1994.

[5] 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.

[6] 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.

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

[8] E. Duesterwald, R. Gupta and M.L. Soffa, "Rigorous Data Flow Testing through Output Influences," *Proc. 2nd Irvine Software Symposium*, pages 131-145, Irvine, CA, March 1992.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] D. Jackson and E.J. Rollins, "A New Model of Program Dependences for Reverse Engineering," *Proc. of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineer*, pg. 2-10, 1994.

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

[17] 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.

[18] J.R. Lyle, D.R. Wallace, J.R. Graham, K.B. Gallagher, J.P. Poole, and D.W. Binkley, "Unravel: A CASE Tool to Assist Evaluation of High Integrity Software," *Technical Report NISTIR-5691*, National Institute of Standards and Technology, 1995.

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

[20] J. Ning, A. Engberts, and W. Kozaczynski, "Automated Support for Legacy Code Understanding," *Communications of the ACM*, Vol. 37, No. 5, pages 50-57, 1994.

[21] T. Reps and G. Rosay, "Precise Interprocedural Chopping," *Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41-52, Oct. 1995.

[22] G.A. Venkatesh, "The Semantic Approach to Program Slicing," *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 107-119, June 1991.

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