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-
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

$$BH = (b_0, N_1), \ldots, (b_m, N_m),$$

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, $BH = (b_0, N_1), \ldots, (b_m, N_m)$, the **hybrid slice** with respect to a slicing criteria $SC = (V, b_m)$ is defined as follows:

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

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 $BH = (b_0, N_1), \ldots, (b_m, N_m)$.

We can replace this history by the following shortened breakpoint history:

$$SBH = (b_0, N_1), \ldots, (b_{m-\Delta}, N_{m-\Delta}), (b_m, N_m).$$

The shortened history only considers the most recent $\Delta$ 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 $X$
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 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, $BH = \langle (b_0, -), (b_1, N_1), \ldots (b_m, N_m) \rangle$, a path $P$ from $b_0$ to $b_m$ is feasible if and only if path $P$ is composed of subpaths as follows:

$$P = PATH(b_0 \sim b_1).$$

$$\vdots PATH(b_{m-1} \sim b_m)$$

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

$$NODES(PATH(b_{i-1} \sim b_i)) \cap N_i = \emptyset,$$

where $NODES(PATH(b_{i-1} \sim b_i))$ 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 $j < i$, if and only if there exists a feasible path $P$ such that:

- the subpath $PATH(b_{j-1} \sim b_j)$ 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

![Diagram](image-url)
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 \((b_j, N_j), (b_{j+1}, N_{j+1}), \ldots, (b_i, N_i)\). 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 repeatedly 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 \(VAR^{ent}_{\{n\}}\) and \(VAR^{exit}_{\{n\}}\). 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 \(VAR^{ent}_{\{n\}}\) is the feasible subset of \(VAR^{ent}_{\{n\}}\)). Finally when variable v reaches a definition of v, we examine the associated breakpoint. If the associated breakpoint is \(b_i\), then the statement becomes a potential candidate for inclusion in subslice \(HSLICE(b_{i-1}, b_i)\).

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 \(THSLICE^{ent}_{\{n\}}\) and \(THSLICE^{exit}_{\{n\}}\). 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, \(THSLICE^{ent}_{\{n\}}\) is the feasible subset of \(THSLICE^{exit}_{\{n\}}\)). 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 dependencies 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 \(CD^{ent}_{\{n\}}\) and \(CD^{exit}_{\{n\}}\). 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. \textbf{algorithm} ComputeHybridSlice \((SC = (V, b_m), BH =< (b_0, \neg), \ldots, (b_m, N_m) >)\)
2. \textbf{repeat}
   \begin{align*}
   &HSLICE(b_0, b_1) \leftarrow HSLICE(b_1, b_2) \leftarrow \ldots \leftarrow HSLICE(b_{m-1}, b_m) \leftarrow \phi \\
   &\text{triples} \leftarrow \{(v, s, b_m) : v \in V, b_m \text{ is at point immediately preceding } s\}
   \end{align*}
3. \textbf{repeat}
   \begin{enumerate}
   \item \textbf{Detection Step}
   \begin{enumerate}
   \item \textbf{for each triple} \((v, s, b_i) \in \text{triples}\) \textbf{do}
   \begin{enumerate}
   \item \(VAR_{en}^{alt}[s] \leftarrow VAR_{en}^{alt}[s] \cup \{(v, b_i)\}\)
   \item \(VAR_{ex}^{alt}[n] \leftarrow \phi\)
   \end{enumerate}
   \end{enumerate}
   \item \textbf{Verification Step}
   \begin{enumerate}
   \item \textbf{for each pair} \((s, b_i) \in \text{pairs}\) \textbf{do}
   \begin{enumerate}
   \item \(THSLICE_{en}^{alt}[s] \leftarrow THSLICE_{en}^{alt}[s] \cup \{(s, b_i)\}\)
   \item \(Tslice = Tslice \cup \{s\}\)
   \end{enumerate}
   \end{enumerate}
   \end{enumerate}
4. \textbf{until} \(\text{triples} = \phi\)
5. \textbf{end ComputeHybridSlice}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2}
\caption{Overview of Hybrid Slicing Algorithm.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3}
\caption{Identification of Potential Statements in the Slice.}
\end{figure}
1. **algorithm** ComputeActualSlice ( )

2. \[\text{while } \text{Tslicelist} \neq \phi \text{ do} \]

3. \[\text{get } n \text{ from the Tslicelist} \]

4. \[\text{for each } (s, b_i) \in \text{THSLICE}_{en}^{all}[n] \text{ and } b_{i-1} = n \text{ do} \]

5. \[\text{HSLICE}(b_{i-1}, b_i) = \text{HSLICE}(b_{i-1}, b_i) \cup \{s\} \]

6. \[\text{triples} \leftarrow \text{triples} \cup \{(v, s, b_i) : v \in \text{Ref}(s)\} \]

7. \[\text{if executable slice required then ComputeControlTriples}(s \in \text{HSLICE}(b_{i-1}, b_i)) \text{ endif} \]

8. \[\text{THSLICE}_{en}^{all}[n] = \text{THSLICE}_{en}^{all}[n] \setminus \{(s, b_i)\} \]

9. \[\text{endfor} \]

10. \[\text{for each} \ p \in \text{Pred}(n) \text{ do} \]

11. \[\text{NEWTHSLICE} \leftarrow \bigcup_{w \in \text{Succ}(p)} \text{THSLICE}_{en}^{all}[w] \]

12. \[\text{if NEWTHSLICE} \not\subseteq \text{THSLICE}_{ex}^{all}[p] \text{ then} \]

13. \[\text{THSLICE}_{ex}^{all}[p] \leftarrow \text{THSLICE}_{ex}^{all}[p] \cup \text{NEWTHSLICE} \]

14. \[\text{THSLICE}_{ex}^{all}[p] \leftarrow \text{THSLICE}_{ex}^{all}[p] \setminus \{(s, b_i) : s \in N_i\} \]

15. \[\text{THSLICE}_{en}^{all}[p] \leftarrow \text{THSLICE}_{en}^{all}[p] \cup \text{THSLICE}_{ex}^{all}[p] \]

16. \[\text{Tslicelist} \leftarrow \text{Tslicelist} \cup \{p\} \]

17. \[\text{endif} \]

18. \[\text{endfor} \]

19. \[\text{endwhile} \]

20. **end ComputeActualSlice**

---

1. **algorithm** ComputeControlTriples ( s \in \text{HSLICE}(b_{i-1}, b_i) )

2. \[\forall n, \text{CD}_{ex}^{all}[n] \leftarrow \text{CD}_{en}^{all}[n] \leftarrow \phi \]

3. \[\text{for each} \ \text{immediate control predecessor } c \ \text{of } s \ \text{do} \]

4. \[\text{CD}_{ex}^{all}[s] \leftarrow \text{CD}_{ex}^{all}[s] \cup \{(c, b_i)\} \]

5. \[\text{endfor} \]

6. \[\text{Cdlist} \leftarrow \{s\} \]

7. \[\text{while } \text{Cdlist} \neq \phi \text{ do} \]

8. \[\text{get } n \text{ from Cdlist} \]

9. \[\text{NEWCD} \leftarrow \{(c, b_{max}) : j \leq \text{max}, \forall(c, b_j) \in \text{CD}_{en}^{all}[w], \text{ where } w \in \text{Succ}(n)\} \]

10. \[\text{if NEWCD} \not\subseteq \text{CD}_{ex}^{all}[n] \text{ then} \]

11. \[\text{CD}_{ex}^{all}[n] \leftarrow \text{CD}_{ex}^{all}[n] \cup \text{NEWCD} \]

12. \[\text{CD}_{ex}^{all}[n] \leftarrow \text{CD}_{ex}^{all}[n] \setminus \{(c, b_i) : n \in N_i\} \]

13. \[\text{CD}_{en}^{all}[n] \leftarrow \text{CD}_{en}^{all} \cup \text{CD}_{ex}^{all}[n] \]

14. \[\text{for each} \ (c, b_i) \in \text{CD}_{en}^{all}[n] \ \text{st} \ b_{i-1} = n \ \text{do} \]

15. \[\text{CD}_{en}^{all}[n] \leftarrow (\text{CD}_{en}^{all}[n] \setminus \{(c, b_i)\}) \cup \{(c, b_{i-1})\} \]

16. \[\text{endif} \]

17. \[\text{Cdlist} \leftarrow \text{Cdlist} \cup \text{Pred}(n) \]

18. \[\text{endif} \]

19. **endwhile**

20. \[\text{Let } (c, b_k) \in \text{CD}_{en}^{all}[c]: \]

21. \[\text{HSLICE}(b_{k-1}, b_k) \leftarrow \text{HSLICE}(b_{k-1}, b_k) \cup \{c\} \]

22. \[\text{triples} \leftarrow \text{triples} \cup \{(v, c, b_k) : v \in \text{Ref}(c)\} \]

23. **ComputeControlTriples**(c \in \text{HSLICE}(b_{k-1}, b_k))

24. **end ComputeControlTriples**

---

Figure 4: Identification of Actual Statements in the Slice.

Figure 5: Considering Control Dependences.
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 ∈ HSLICE(b4, b5)). Based upon these results predicate 8, which is the immediate control predecessor of 9, is included in subslice HSLICE(b3, b4). The predicate 8 is included in subslice HSLICE(b3, b4) 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 dependencies. 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 x V. Thus, a single operation on the data flow set requires at most O(B x V) time. Each node may be examined MAX(d + 1, B) times until the data flow stabilizes. Thus the cost of the detection step is bounded by O(MAX(d + 1, B) x N x B x V). The size of a data flow set during the verification step is bounded by N x 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) x N x B). The total cost of the two steps is therefore given by O(MAX(d + 1, B) x N x 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(B2 x N2). Since the size of a hybrid slice is bounded by B x 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(B2 x N2). 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 \( CH = \langle CRO, CR_1, ..., CR_m \rangle \), 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:

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

**Definition 6:** For a calling history, \( CH = \langle CRO, CR_1, ..., CR_m \rangle \), the overall hybrid slice with respect to a slicing criteria \( S = (V, CR_m) \) is defined as follows:

\[
\text{HSLICE}(CR_m) = \bigcup_{i=1}^{m} \text{HSSLICE}(CR_{i-1}, CR_i)
\]

where subslice \( \text{HSSLICE}(CR_{i-1}, CR_i) \) contains those statements that were possibly executed after \( CR_i \) 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, \( CH = \langle CRO, CR_1, ..., CR_m \rangle \), a path from \( CR_0 \) to \( CR_m \) is feasible if and only if path \( P \) is composed of the following subpaths:
\[
P = \text{PATH}(CR_0 \leadsto CR_i).
\]
\[
\text{PATH}(CR_1 \leadsto CR_2).
\]
\[
\ldots \text{PATH}(CR_{m-1} \leadsto CR_m),
\]
such that for \(1 \leq i \leq m,\)

- if \(CR_{i-1}\) is a call and \(CR_i\) is a call then
  \[p_{\text{call}}^{i-1} = p_{\text{call}}^i;\]

- if \(CR_{i-1}\) is a call and \(CR_i\) is a return then
  \[p_{\text{call}}^{i-1} = p_{\text{call}}^i \quad \text{and} \quad p_{\text{callee}}^{i} = p_{\text{callee}}^{i-1};\]

- if \(CR_{i-1}\) is a return and \(CR_i\) is a call then
  \[p_{\text{callee}}^{i-1} = p_{\text{callee}}^{i};\]

- if \(CR_{i-1}\) is a return and \(CR_i\) is a return then
  \[p_{\text{callee}}^{i-1} = p_{\text{callee}}^{i}.\]

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 is 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

---

**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 \( P_2 \), then the slicing is not done on \( P_2 \). 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 ComputeInterTentativeSlice, 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 and the definition are 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 algorithum continues 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 \( H = \langle H_0, H_1, \ldots, H_m \rangle \), where \( H_i \) is one of the following: Breakpoint: \( (b_i, N_i) \); Procedure Call: \( CALL[P_{\text{caller}} \rightarrow P_{\text{callee}} \text{ at } s] \); or Procedure Return: \( RET[P_{\text{callee}} \rightarrow P_{\text{caller}} \text{ at } s] \).

Definition 9: For a given combined history, \( H = \langle H_0, H_1, \ldots, H_m \rangle \), the overall hybrid slice with respect to a slicing criteria \( SC = (V, CR_m) \) is defined as follows:

\[
HSLICE(H_m) = \bigcup_{i=0}^{m-1} HSLICE(H_i, H_{i+1})
\]

where subslice \( HSLICE(H_i, H_{i+1}) \) 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


