

Reuse-Driven Interprocedural Slicing

Mary Jean Harrold

Computer and Information Science
The Ohio State University
395 Dreese Lab
Columbus, OH 43210-1227 USA
+1 614 292 2568
harrold@cis.ohio-state.edu

Ning Ci

Computer and Information Science
The Ohio State University
395 Dreese Lab
Columbus, OH 43210-1227 USA
+1 614 292 1152
ci@cis.ohio-state.edu

ABSTRACT

To manage the evolution of software systems effectively, software developers must understand software systems, identify and evaluate alternative modification strategies, implement appropriate modifications, and validate the correctness of the modifications. One analysis technique that assists in many of these activities is program slicing. To facilitate the application of slicing to large software systems, we adapted a control-flow-based interprocedural slicing algorithm so that it accounts for interprocedural control dependencies not recognized by other slicing algorithms, and reuses slicing information for improved efficiency. Our initial studies suggest that additional slice accuracy and slicing efficiency may be achieved with our algorithm.

KEYWORDS

Program slicing, interprocedural analysis, data-flow analysis, demand analysis.

1 INTRODUCTION

Throughout their lifetimes, software systems continue to evolve. To manage this evolution effectively, software developers need to understand software systems, identify and evaluate alternative modification strategies, implement appropriate modifications, and validate the correctness of the modifications. One analysis technique that assists in many of these activities is program slicing (e.g., [3, 9, 14]). A *program slice* is defined with respect to a slicing criterion $\langle S, V \rangle$ in which S is a program point and V is a subset of program variables. A slice consists of a subset of program statements, possibly executable, that affect, or are affected by, the values of variables in V at S [15]. To be applicable to large software systems, slicing algorithms must perform the slicing across procedure boundaries (*interprocedural slicing*), and must be sufficiently precise and efficient.

The imprecision of early interprocedural slicing algorithms [15] was corrected by the system-dependence-graph approach, which guarantees that slices are computed only along realizable paths in the program [9].¹ With additional expense, more precise slices can be obtained if the computation uses a distinct calling context for each procedure at each call site. The context-graph approach lets a user select the number of distinct calling contexts, and thus, control the precision of slices [2]. The efficiency of interprocedural slicing may be improved by using database techniques [13] or demand approaches [2, 4, 6, 10], or by efficient management of program representations [2]. Our attempts to use existing interprocedural slicing algorithms for our applications uncovered several areas for improvement in the accuracy and efficiency of existing approaches.

First, none of the existing interprocedural slicing algorithms accounts for interprocedural effects on control dependence. Thus, given a slicing criterion $\langle S, V \rangle$, the slices computed by these algorithms may omit statements from the slice that affect the values of variables in V at program point S . The reason for this omission is that these algorithms use control-dependence information based on control-flow graphs for single procedures (*intraprocedural control dependence*). Recent work [8] describes several ways in which this intraprocedural control-dependence computation inaccurately models the semantic dependencies that exist between program statements.

One way in which intraprocedural control-dependence computation inaccurately models semantic dependence is seen in the effects of embedded halts.² For example, consider the alternative version of program Sum with statement 18', shown in Figure 1. The explicit halt at statement 18' affects the control dependencies of statements in procedures M and B. If control dependence is

¹A path is *realizable* if it respects the fact that procedures always return to the site of the most recent call [10].

²For languages that do not support embedded halts, the embedded-halt effect is of no concern. However, a study of a variety of non-trivial C programs showed that over 63% of the programs contained `exit()` statements that can affect control dependencies, and thus, potentially the outcome of slicing [8].

```

    procedure M
1  begin M
2    read i,j
3    sum = 0
4    while i < 10 do
5a     call B
5b     return B
    endwhile
6a    call B
6b    return B
7    print sum
8  end M

    procedure B
9  begin B
10a  call C
10b  return C
11  if j >= 0 then
12    sum = sum + j
13    read j
    endif
14    i = i + 1
15  end B

    procedure C
16  begin C
17  if sum > 100 then
18  print("error")
    endif
19  end C

```

alternative version of 18:
18' halt

Figure 1: Sample program Sum.

computed intraprocedurally in the usual way, without considering the effects of the embedded halt, statement 7, for example, will be control dependent on entering M. On closer inspection, however, we see that statement 7 depends most immediately for its execution on statement 17, because of the explicit halt in statement 18'. Thus, according to the definition of semantic dependence [12], statement 7 is semantically dependent on statement 17, and statement 17 should be included in any slice that includes statement 7. However, because the control dependence information that existing interprocedural slicing algorithms use is computed without considering the effects of embedded halts, none of slices computed by existing algorithms contains statement 17.

Second, although existing interprocedural slicing algorithms reuse slicing information, none of them is reported to reuse the slices for subsequent slicing. This reuse may be especially important when single or multiple users perform repeated slices on the same program [2]. For example, suppose that we slice with criterion $\langle 5b, \text{sum} \rangle$ on program Sum, shown in Figure 1, and save the slicing information. If we later compute a slice for $\langle 6b, \text{sum} \rangle$, we can reuse the partial slice computed at $\langle 15, \text{sum} \rangle$ (which was used to compute the slice for $\langle 5b, \text{sum} \rangle$) to compute the slice quickly because we avoid reslicing into procedures B and C.

Finally, none of the existing data-flow-based interprocedural slicing algorithms [2, 6] presents a technique for slicing into procedures that call the procedure in which the slice is initiated. For example, suppose that we wish to begin the slice in procedure B at statement 11. After slicing procedure C and reaching the entry to B, the computation of the slice can (1) terminate, (2) slice into all calling procedures at all call sites, or (3) slice into specific calling procedures selected by the user or by the program. Existing data-flow-based interprocedural slic-

ing algorithms use method (1). There are situations, however, in which methods (2) or (3) are desirable.

In this paper, we present an interprocedural slicing algorithm that addresses these problems of accuracy and efficiency in interprocedural slicing. Our algorithm uses data-flow analysis to compute the slices. At each control-flow graph node encountered, the algorithm determines whether the data-flow sets have changed since the last time the node was visited, and, if so, continues processing along control-flow and control-dependence predecessors of the node. When the propagation reaches a node whose control flow predecessor(s) is in another procedure, the algorithm uses cached slicing information, if possible. If no cached slicing information exists, the algorithm computes the partial slicing information for that procedure, and stores it for reuse. When processing these procedure-boundary nodes, the algorithm uses a call stack to ensure that the slice is computed only along realizable paths in the program. Finally, the algorithm uses control dependence information that lets it correctly account for interprocedural control dependencies in the computation of a slice, and thus, avoid omitting statements from the slice.

Our approach has several advantages over previous approaches. First, our algorithm accounts for interprocedural control dependencies not accounted for by any existing interprocedural slicing algorithms. A study of a small set of C programs showed that three of the eight programs in the suite had embedded halts, and that the slices computed for two of these three programs using other algorithms omit statements that should be included in the slice. Second, our algorithm reuses previously computed slicing information, including partial slices, when a criterion is encountered in the computation of interprocedural slices. Our initial studies suggest that caching may provide savings in processing. Finally, our algorithm computes slices into procedures that call the procedure in which the slice is initiated either by slicing into all called procedures or by slicing into procedures selected by the user.

2 SLICE COMPUTATION

In this section, we give the details of our algorithm. First, we discuss the input and output for the main component of the algorithm, `ComputePSlice`, which is shown in Figure 2.³ Then, we discuss the global and local data structures used by `ComputePSlice`. Finally, we discuss the way in which `ComputePSlice` propagates the data-flow sets to compute the slice.

³Our algorithm can handle programs with recursion by overestimating the statements in the slice. For brevity, we omit the discussion of this overestimation from this paper.

```

function ComputePSlice
input      ( $s, v$ ) : slicing criterion for CFG node  $s$  for procedure
               $P$  and variable  $v$ 
output    PSlice : set of nodes in  $\mathcal{P}$ , initially empty
              CalleeVars : variables relevant
globals   CallGraph : program  $\mathcal{P}$ 's call graph
              Callstack : records calling context during slicing
              SliceList[crit] : list of (entryVars, pslice), stores
              previously computed slice information
declare   WorkList : ordered list of nodes in  $P$ , initially empty
              CFG : control flow graph for  $P$ 
               $N$  : node in CFG for  $P$ 
              CD[ $N$ ] : control dependencies for nodes in  $P$ 
              RelIn[ $N$ ], RelOut[ $N$ ] : sets of relevant variables used
              for propagation, initially empty
              RelVarTo, DummyVars : temporary sets of relevant
              variables, initially empty
              Def[ $N$ ], Ref[ $N$ ] : variables defined, referenced at  $N$ 

begin ComputePSlice
1. if SliceList[ $s, v$ ]  $\neq$  NULL then
2.   PropagateToCallers(SliceList[ $s, v$ ].entryVars,
                       SliceList[ $s, v$ ].pslice)
3. else
4.   Initialize data structures
5.   while WorkList  $\neq$   $\phi$  do
6.     Remove  $N$  from WorkList
7.     Update RelOut[ $N$ ], RelIn[ $N$ ], and PSlice
8.     case  $N$  is an Entry node :
9.       SliceList[ $s, v$ ] = (RelOut[ $N$ ], PSlice)
10.      PropagateToCallers(RelOut[ $N$ ], PSlice)
11.      case  $N$  is a Return node associated with node  $C$ 
          that represents a Call to procedure  $P_n$ 
12.        Add  $C$  to WorkList
13.        Modify RelIn[ $N$ ] according to Def[ $C$ ] and Ref[ $C$ ]
14.        RelVarTo = bind(RelIn[ $N$ ], globals, parameters)
15.        if RelVarTo  $\neq$   $\phi$ 
16.          foreach  $u \in$  RelVarTo do
17.            Push  $C$  onto CallStack
18.            (PSlice, RelIn[ $N$ ]) = (PSlice, RelIn[ $N$ ])
               $\cup$  ComputePSlice(Exit $P_n$ ,  $v$ )
19.          endfor
20.        endif
21.        default : // any other type of node
22.          Add control flow, control dependence predecessors
23.        endcase
24.      endwhile
25.    endif
26.  return PSlice, CalleeVars
end ComputePSlice

function PropagateToCallers
input      VarSet, NodeSet

begin PropagateToCallers
27. if CallStack  $\neq$  NULL then
28.    $C$  = PopCallStack
29.   CalleeVars = backbind(VarSet,  $C$ )
30.   PSlice = NodeSet
else
31.   foreach call site  $C_i$  of  $P$  do
32.     RelVarTo = backbind(VarSet,  $C_i$ )
33.     foreach  $u \in$  RelVarTo do
34.       (NodeSet, DummyVars) = (NodeSet, DummyVars)
          $\cup$  ComputePSlice( $C_i$ ,  $u$ )
35.     endfor
36.   endfor
37.   PSlice = NodeSet
38. endif
end PropagateToCallers

```

Figure 2: Function to compute a slice for $\langle s, v \rangle$.

2.1 Input and Output

Given slicing criterion $\langle S, V \rangle$ for program \mathcal{P} , our algorithm identifies the procedure P containing S and the node s in P 's control flow graph that corresponds to S , and creates slicing criteria $\langle s, v \rangle$ for all $v \in V$. Our algorithm then calls **ComputePSlice** for each criterion. **ComputePSlice** inputs a slicing criterion, and outputs *PSlice* and *CalleeVars*. *PSlice* contains nodes that represent the partial slice at the entry point of P , and *CalleeVars* contains the variables (*relevant variables*) that are used to continue slicing in the calling procedure after **ComputePSlice** returns. When the traversal reaches a procedure boundary, **ComputePSlice** performs the required parameter binding to create a new set of relevant variables, and creates a new criterion for each relevant variable.⁴ **ComputePSlice** is called for each newly created criterion.

Example 1. Suppose that we wish to compute a slice at point 6b of program *Sum* with $V = \{i, \text{sum}\}$. At this return point, **ComputePSlice** determines that both i and *sum* are global variables and are thus, relevant at B 's exit point, and creates two new slicing criteria: $\langle 15, i \rangle$ and $\langle 15, \text{sum} \rangle$. For $\langle 15, i \rangle$, **ComputePSlice** returns $\{14\}$ for *PSlice* and $\{i\}$ for *CalleeVars*; for $\langle 15, \text{sum} \rangle$, **ComputePSlice** returns $\{11, 12\}$ for *PSlice* and $\{\text{sum}, j\}$ for *CalleeVars*. \square

2.2 Global Data Structures

ComputePSlice uses program \mathcal{P} 's *CallGraph* to access information about call sites and locations of halt statements, and keeps a *CallStack* to record the calling history as it performs the slicing. To facilitate reuse, **ComputePSlice** caches slicing information using *SliceList*, which contains one entry for each slicing criterion, $\langle s, v \rangle$, that was previously sliced. Each entry in *SliceList* contains two fields: *SliceList*[s, v].*entryVars*, the variables that are relevant at the entry to P after slicing with $\langle s, v \rangle$; and *SliceList*[s, v].*pslice*, the partial slice that was computed for $\langle s, v \rangle$; this partial slice contains the results of slicing into procedures called by P but not into the procedures that call P . This information can be reused during computation of a particular slice or subsequent slices. In either case, repeated slicing with the same criterion is avoided.

Example 2. After slicing procedure B with $\langle 15, i \rangle$ and $\langle 15, \text{sum} \rangle$, *SliceList* contains cached information: for $[15, i]$, *entryVars* is $\{i\}$ and *pslice* is $\{14\}$; for $[15, \text{sum}]$, *entryVars* is $\{\text{sum}, j\}$ and *pslice* is $\{11, 12\}$; for $[19, i]$, $[19, i]$, $[19, \text{sum}]$, $[19, \text{sum}]$, $[19, j]$, and $[19, j]$, both *pslice* and *entryVars* are ϕ . Subsequent calls to

⁴**ComputePSlice** may also create a new criterion to account for interprocedural control dependencies – we discuss this case in subsequent sections.

`ComputePSlice` with either $\langle 15, i \rangle$ or $\langle 15, \text{sum} \rangle$, for example, will use the cached information instead of slicing B again with the criterion. \square

The difference between the information returned by `ComputePSlice` and the information stored in `SliceList` bears further discussion. When `ComputePSlice` is called with a criterion $\langle s, v \rangle$, it computes a new partial slice, `PSlice`, for P with respect to $\langle s, v \rangle$. When the P 's entry is reached, `ComputePSlice` saves `PSlice` in `SliceList[s, v].pslice`. If `CallStack` is not empty, `ComputePSlice` returns `SliceList[s, v].pslice` to the calling procedure; otherwise, instead of returning `SliceList[s, v].pslice` to callers, `ComputePSlice` invokes a series of calls to `ComputePSlice` for all (or some subset of) calling procedures. At P 's entry, `ComputePSlice` also saves the current set of relevant variables, `RelOut[Entry]`, in `SliceList[s, v].entryVars`. Then `ComputePSlice` uses `RelOut[Entry]` along with parameter information at the call site to create `CalleeVars`, which is also returned by `ComputePSlice`. If a subsequent call to `ComputePSlice` is made with $\langle s, v \rangle$, `SliceList[s, v].entryVars` is used to create `CalleeVars`, and both `CalleeVars` and `SliceList[s, v].pslice` are returned by `ComputePSlice`, without reslicing into the called procedures. Because program `Sum` contains only global variables, `SliceList[s, v].entryVars` and `CalleeVars` are identical. In programs with parameters, however, these sets may differ, depending on the scope and type of variables and parameters.

2.3 Local Data Structures

To compute `PSlice`, `ComputePSlice` uses the control flow graph, `CFG`, for P . Figure 3 shows the control flow graphs for program `Sum`. In each control flow graph, nodes represent statements and edges represent the flow of control between statements. Each procedure has unique Entry and Exit nodes, and each call site has unique Call and Return nodes. Dashed edges show the interprocedural flow of control between procedures.

`ComputePSlice` also uses control dependencies, `CD[N]`, for slice computation. Our algorithm differs from other existing interprocedural slicing algorithms in that the slice computation accounts for the effects of embedded halts [8]. Before control-dependence information is computed, our algorithm uses the `CallGraph` to determine whether any procedures reachable from P contain embedded halts.⁵ If there are no reachable embedded halts, the control dependencies for statements in P are correctly computed using P 's control flow graph. If there are reachable embedded halts, then

⁵If there is no statically unreachable code in the program, then inspection of the call graph is sufficient to determine whether P can call procedures with embedded halts; otherwise, and an algorithm that uses the control flow graphs can be used for this identification [8].

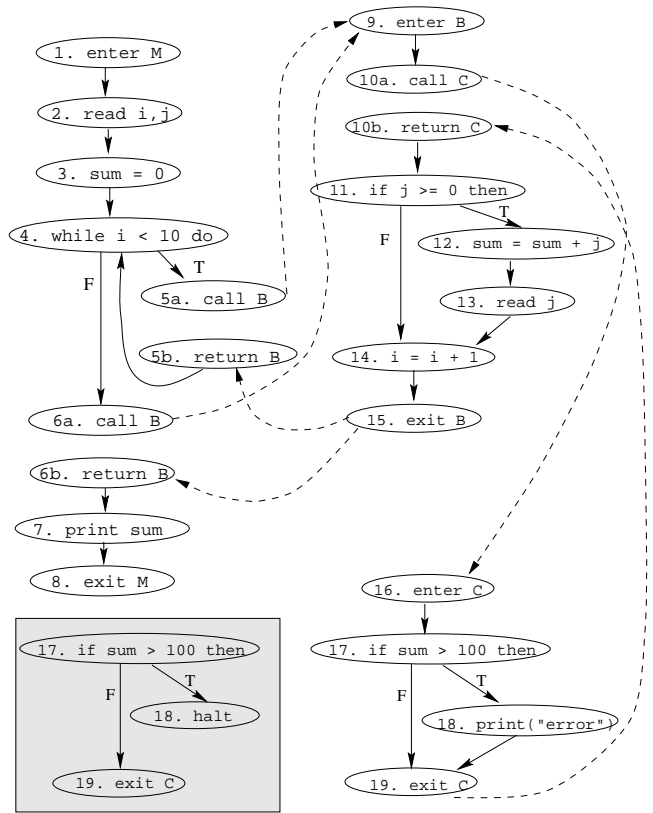


Figure 3: Control flow graphs for `Sum`.

there exist call sites in P whose return may be control dependent on conditional statement(s) (conditional statements controlling embedded halts) in other procedures. Although intraprocedural control-dependence analysis cannot identify these conditional statements, it can, using an *augmented control flow graph* (ACFG) [8], compute partial control-dependence information that `ComputePSlice` uses to compute correct interprocedural slices.

An ACFG for a procedure P is a control flow graph that is augmented with placeholder nodes that represent interprocedural control dependencies. An ACFG has a unique node, Super Entry, that acts as a placeholder for entry to P . For each Return node associated with a call site to a procedure P_i that contains a halt statement, an ACFG contains a unique conditional node, Return Predicate, that acts as a placeholder for the conditional statements on which return from P_i is control dependent. An ACFG also contains a unique node, Super Exit, that represents all exits from P . Finally, an ACFG has additional control flow edges that connect CFG nodes with these new nodes.

When the ACFG is used to compute intraprocedural control dependencies using a control-dependence computation algorithm, such as [5], the control dependen-

cies, $CD[N]$, for all nodes N that are control dependent on returning from called procedures and thus, control dependent on predicates in other procedures, will contain a Return Predicate node. When such an N is added to $PSlice$, $ComputePSlice$ forces the slicing to continue into called procedures until the node on which the embedded halt depends is found.

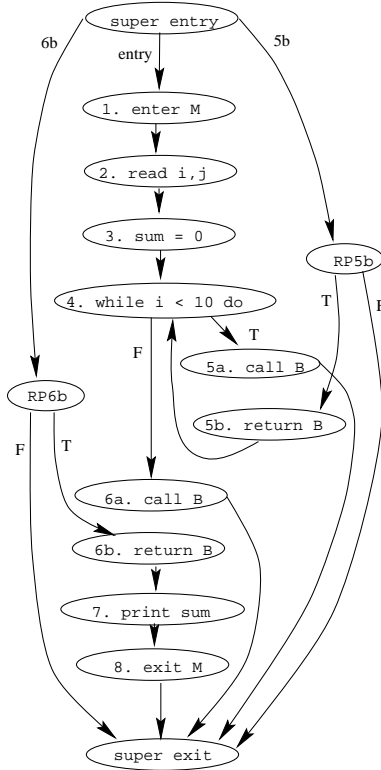


Figure 4: Augmented control flow graph for M.

Example 3. Figure 4 depicts the ACFG for M. In addition to the Super Entry and Super Exit nodes, there are two Return Predicate nodes: RP5b, representing the conditional statement on which the return from the call at 5a depends; and RP6b, representing the conditional statement on which return from the call at 6a depends. There are also edges connecting CFG nodes with these new nodes. When the ACFG is used in control-dependence computation, it finds that nodes 5b and 4 are control dependent on RP5b, node 6a is control dependent on node 4, and nodes 6b and 7 are control dependent on RP6b. \square

$ComputePSlice$ uses an ordered *WorkList* to record nodes in P that remain to be processed, and uses data-flow sets, $RelIn[N]$ and $RelOut[N]$, to record the variables that are relevant to the slice at points before and after nodes in the control flow graph, respectively. $ComputePSlice$ uses $Def[N]$ and $Ref[N]$ sets to store definitions and references, respectively, for N .

2.4 Propagation Steps

Each time $ComputePSlice$ is invoked, it first searches *SliceList* for $\langle s, v \rangle$ (line 1). If $\langle s, v \rangle$ is found, $ComputePSlice$ calls procedure $PropagateToCallers$ (line 2) to propagate the slicing information back to the caller(s) of P . If $PropagateToCallers$, shown in Figure 2, finds C , the caller of P , on *CallStack* (lines 27–30), it uses *backbind* to determine *CalleeVars* using *SliceList[s, v].entryVars* together with the actual parameters at C ; this computation is similar to the algorithm presented in [6].

If $PropagateToCallers$ finds that the *CallStack* is empty (line 31), there are two possibilities: (1) the slicing has reached the main procedure, or (2) for procedures other than main, the slicing has reached a caller of the procedure in which the slice was initiated or the procedure itself. In case (1), this invocation of $ComputePSlice$ terminates. In case (2), there are two choices: the user can select those calling procedures in which to continue the slicing; or the algorithm can select those (possibly all) calling procedures in which to continue the slicing. In either case, the algorithm calls $ComputePSlice$ to compute and return the slices for all selected calling procedures (lines 31–38); in this case, *CalleeVars* is empty. The algorithm then computes the union of those returned slices. After returning from $PropagateToCallers$, the function returns *PSlice* and *CalleeVars* (line 26).

Example 4. Consider a slice with criterion $\langle 7, sum \rangle$. The slicing proceeds through procedure B, and then to C. When $PropagateToCallers$ is invoked at the entry to C, it finds B on the *CallStack* and continues slicing in B. Subsequently, when $PropagateToCallers$ is invoked at the entry to B, it finds M on *CallStack*, and continues slicing in M. Now consider a slice with criterion $\langle 17, sum \rangle$. When $PropagateToCallers$ reaches the entry to C, it finds that *CallStack* is empty, and slices into B, C’s only caller. When $PropagateToCallers$ subsequently reaches the entry to B, and finds *CallStack* empty, it either continues slicing at both call sites to B, statements 5b and 6b, or continues slicing at the call sites selected by the user. \square

If $\langle s, v \rangle$ is not found in *SliceList* (line 3), then $ComputePSlice$ computes the slice for the criterion. It first performs initialization tasks (line 4). $ComputePSlice$ requests the required analysis information for P from the underlying analysis system. This information can be computed at this time, accessed from a location in memory, or retrieved from the database. $ComputePSlice$ adds s to *PSlice* and to *WorkList*.

$ComputePSlice$ then begins to process nodes on *WorkList*; the **while** loop (lines 5 – 24) handles this processing. After removing N from the *WorkList*,

the algorithm updates the $RelOut[N]$ and $RelIn[N]$ sets using $Def[N]$ and $Ref[N]$ sets. It first computes $RelOut[N]$ as the union of $RelIn[Succ]$, where $Succ$ is a successor of N in the CFG and then assigns $RelOut[N]$ to $RelIn[N]$. If $Def[N] \cap RelOut[N]$ is not empty, then `ComputePSlice` removes the variable in $Def[N]$ from $RelIn[N]$, adds variables in $Ref[N]$ to $RelIn[N]$, and adds N to $PSlice$.

`ComputePSlice` uses N 's node type to determine the actions it takes. If N is an Entry node (lines 8–10) in the CFG , then the partial slice is complete for this criterion. The algorithm saves the current $PSlice$ and $RelOut[N]$ in $SliceList$, and calls `PropagateToCallers`, passing $RelOut[N]$ and $PSlice$. `PropagateToCallers` performs actions that are similar to those described in Section 2.2, except that it uses the $RelOut[N]$ and the computed slice, $PSlice$, instead of the stored information in $SliceList$.

If N is a Return node in the CFG , `ComputePSlice` adds the corresponding Call node C to $WorkList$ (line 12), updates $RelIn[N]$ using $Def[C]$ and $Ref[C]$ sets (line 13), and computes $RelVarTo$, the variables that are relevant at the exit of the called procedure P_n (line 14). At this point, `ComputePSlice` also determines whether the return node's $CD[N]$ is a Return Predicate node, and if so, the algorithm adds a dummy variable to $RelVarTo$. This dummy variable forces the slicing to continue into P_n even if there are no other relevant variables at the exit node of P_n . If $RelVarTo$ is not empty, `ComputePSlice` creates a new slicing criterion for each variable in $RelVarTo$ (line 17) and calls `ComputePSlice` for each of them. The results of these calls are added to the existing $PSlice$ and $RelIn[N]$ sets.

Example 5. For the alternative version of program `Sum` with statement 18', consider the computation of a slice that reaches node 10b. Suppose that $RelOut[10b]$ contains a variable that is not in scope in procedure `C` (not illustrated in the example). In this case, the $RelVarTo$ set created by `ComputePSlice` would be empty, and using other algorithms, the slicing would not continue into `C`. However, under our approach, we know that there is a conditional statement in a called procedure that must be located and added to $PSlice$ – the fact that $CD[10b]$ is a Return Predicate tells us this. Thus, `ComputePSlice` creates criterion $\langle 19, dummy \rangle$, and continues slicing into `C`. In this case, it adds node 17 to $PSlice$ and $\{sum\}$ is passed back as $CalleeVars$ from `C`, and added to $RelIn[10b]$. \square

If N is any other type of node in the CFG , `ComputePSlice` performs two important functions. First, `ComputePSlice` determines if $RelIn[N]$ has changed during this iteration of the `while` loop, and, if so, the algorithm adds the control flow predeces-

sors of N to $WorkList$. Second, if N is in $PSlice$, then `ComputePSlice` considers each $d \in CD[N]$. If d is a Return Predicate, then `ComputePSlice` adds the Return node in the CFG for P , corresponding to d , to $WorkList$. Otherwise, `ComputePSlice` adds d to $WorkList$ and to $PSlice$.

3 EMPIRICAL RESULTS

We implemented a prototype of our algorithm and performed a number of studies on a set of C programs. This section reports on both the implementation and the studies.

3.1 Implementation

We used the `Aristotle` analysis system [7], which provides analysis information, such as control flow, data flow, and control dependence, to provide analysis information for our prototype slicer. To provide easier access to `Aristotle`'s analysis information, we implemented a wrapper around the `Aristotle` analysis system. The wrapper is written in C++ using the Standard Template Library (STL) and contains 2605 lines of C++ code, not including the STL code. We implemented a prototype of our slicing algorithm in C++ on top of this C++ wrapper on an HP 715 running HP-UX 9.05. The prototype slicer, which contains 821 lines of C++ code, performs interprocedural slicing using our algorithm.

At present, our prototype slicer has several limitations. First, the current version of our slicer does not handle programs with recursion; our algorithm does, however, handle recursion, and we are adding this feature to our prototype. For the studies reported in this paper, we used subject programs that contain no recursion. Second, the current version of our slicer does not use data-flow information that was computed using precise alias analysis. Instead, when performing slicing, we assume that every memory location v , accessed using a pointer, could be aliases to every other memory location accessed using a pointer of the same type as v . Third, the current version of our slicer does not perform the additional analysis required to compute executable slices in the presence of unstructured transfers of control such as `goto`, `break`, or `continue`; we are currently implementing Agrawal's technique [1] to include such statements in the slices. For a given slicing criterion, $\langle s, v \rangle$, our prototype does, however, include all statements that could effect the value of v at s .

3.2 Studies

Table 1 lists the C programs we used as subjects for our studies. One of the programs, `calc`, is a component of the `Aristotle` analysis system; the remaining seven programs were used for a study of control-flow- and data-flow-based testing [11].

Table 1: Subject programs for the studies.

Program	Executable Statements	Number of Functions	Description
tcas	138	8	altitude separation
calc	194	5	postfix calculator
schedule2	297	16	priority scheduler
schedule	299	18	priority scheduler
totinfo	346	16	information measure
printtok	402	21	lexical analyzer
printtok2	483	20	lexical analyzer
replace	516	21	pattern replacement

We performed three studies using our prototype slicer. The first study considered one aspect of slicing itself: the sizes of the slices relative to the size of the program. The next two studies considered aspects of our slicing algorithm: effectiveness and efficiency. The second study compared the slices computed by our algorithm with the slices computed by existing interprocedural slicing algorithms. The third study calculated an estimate of the number of slicing criteria that did not have to be processed by our algorithm because it caches slicing information.

3.2.1 Study 1: Slice size relative to program size. Our objective in this first size study (Study 1a) was to investigate the relationship between slice size and program size for slices computed using our technique. To do this, we considered each subject program \mathcal{P} , each statement S in \mathcal{P} , the *CFG* node s associated with S , and each variable v that is referenced at S ,⁶ and we computed a slice using $\langle s, v \rangle$ as the slicing criterion. For each slice, we computed the percentage of statements in the slice relative to the number of statements in \mathcal{P} . We then determined, for each \mathcal{P} , the number of slices whose percentages lie in the ranges $[0\% - 20\%]$, $(20\% - 40\%)$, $(40\% - 60\%)$, $(60\% - 80\%)$, $(80\% - 100\%)$, and the percentage of slices in each range. Finally, we computed the average of the percentages over all \mathcal{P} .

Figure 5 shows the results of the study as a segmented bar graph. The graph contains one segmented bar for each subject program. Each segmented bar shows, for that program, the percentage of slices that lie in the different ranges. For example, for `calc`, our algorithm computed 65 slices. Of those slices, 43% contain between 0% and 20% of the program statements, 23% contain between 20% and 40% of the program statements, and 34% contain between 60% and 80% of the program statements; there are no slices whose percentage of program statements is between 40% and 60% or between 80% and 100%.

The results show that, for our subject programs, the

⁶Our algorithm can find slices at S for any set of variables that are in scope at S , but for this study, we used only the set of variables referenced at S .

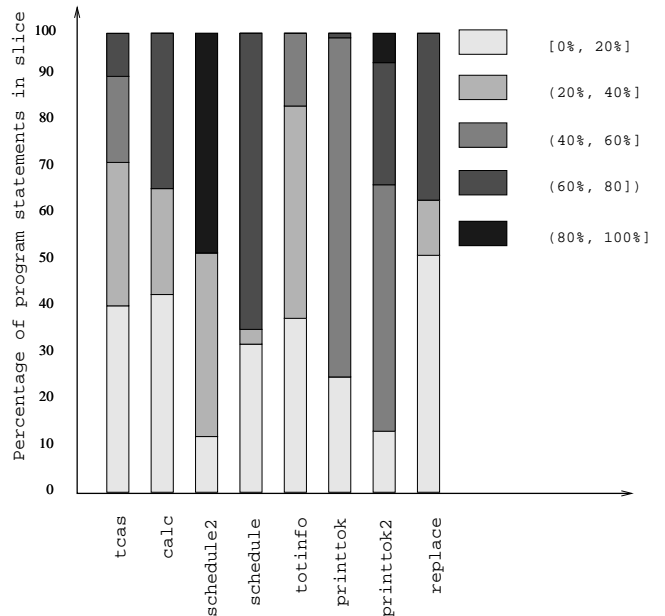


Figure 5: Study 1a – Relative sizes of slices per program.

sizes of the slices relative to the size of the program vary considerably. For example, more than half of the slices for `replace`, our largest subject program, contained less than 40% of the program’s statements, and no slice in `replace` contained more than 80% of the program’s statements, whereas many of the slices for `schedule2` contained between 80% and 100% of the program’s statements.

As we were gathering the statistics for Study 1a, we noticed that many of the slices contained few statements relative to the number of statements in the program. Thus, in our second size study (Study 1b), we wanted to determine the distribution of the relative sizes of the slices across procedures in the program. To do this, we used the results that we obtained in Study 1a, except that we considered the percentages per procedure in \mathcal{P} instead of considering the percentages over all of \mathcal{P} . We report the results for two representative subject programs: `schedule2` and `printtok2`.

Figure 6 shows the results of this study as a segmented bar graph. Each group of bars represents one subject program, and each bar in a group represents a procedure in that program.⁷ We observe that, for `schedule2`, the percentages of slices in the different ranges are unevenly distributed across procedures: in approximately 50% of the procedures, the slices contain over 80% of the program’s statements, and most of the slices in the remaining 50% of the procedures contain less than 40% of the program’s statements. For `printtok2` are also

⁷Some functions had no lines of code containing variable uses; we performed no slices in these functions.

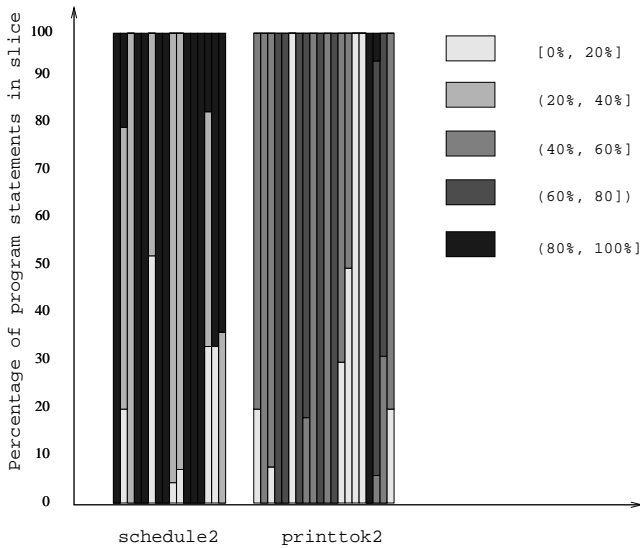


Figure 6: Study 1b – Relative sizes of slices per procedure.

unevenly distributed across procedures, but the difference in the ranges is less than it is for `schedule2`.

These results illustrate that for the procedures in our subject programs, the slice can be small relative to the size of the program, and thus, useful for applications. The results also show, that, even for subject programs whose average slice is large relative to the program being sliced, such as `schedule2`, the slices computed for many of the procedures can be small relative to the size of the program. These results suggest that program metrics other than averages might be useful for predicting relative sizes of slices, and that except for “core” procedures for which slices contain most of the program statements, the size of slices may be small compared to the size of the program. Additional empirical studies will address these issues.

3.2.2 Study 2: Slice inaccuracies due to embedded halts.

Our objective in this study was to compare the slices computed by our algorithm, which considers interprocedural control dependencies in its computation, with slices computed by existing interprocedural slicing algorithms, which do not consider such dependencies. To do this, we considered each subject program \mathcal{P} that contains embedded halts, each statement S in \mathcal{P} , the *CFG* node s associated with S , and each variable v that is referenced at S , and, for each slicing criterion $\langle S, v \rangle$, we computed two slices. We computed the first slice using the prototype implementation of our algorithm. We computed the second slice by simulating the results that would be obtained by existing algorithms – we used our prototype with control dependence information computed in the usual way. We then compared these two slices to determine the percentage of statements that

were in the slice computed with our algorithm but not in the slice computed using the simulation of existing algorithms. We can view this percentage as the percentage of statements “missed” by interprocedural slicing algorithms that do not consider interprocedural control dependencies.

Five of our subject programs, `tcas`, `calc`, `schedule`, `printtok2`, and `replace`, contained no embedded halts. For these subjects, slices computed using our algorithm and slices computed using the simulation of existing algorithms would be identical, and thus, we did not consider them in our study. For one of the programs that contains an embedded halt, `printtok`, there were no differences in the slices computed using our algorithm and the simulation of existing algorithms. For the other two programs, our results show that there can be a significant percentage of missed statements in the slices.

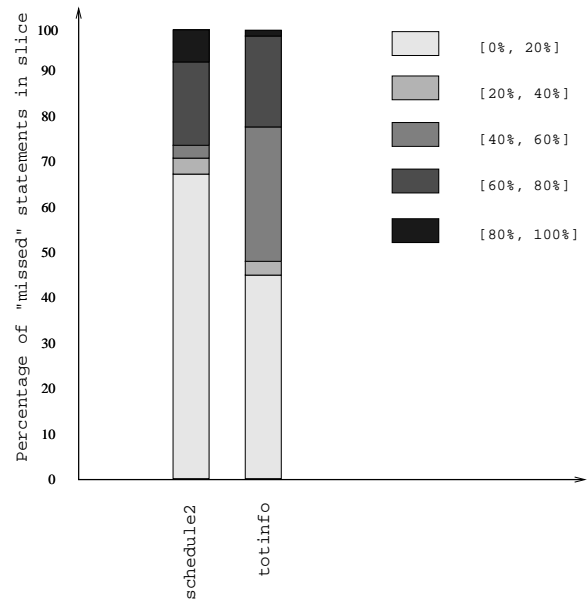


Figure 7: Study 2 – Statements missed by existing interprocedural slicing algorithms.

Figure 7 shows the differences in slices that we observed for these two programs as a segmented bar graph. In both cases, there were at least 30% of the slices that missed at least 40% of the statements. These studies suggest that, for languages, such as C and C++ that have embedded halts, slices computed without considering these embedded halts (i.e., all interprocedural slicing algorithms except ours) can omit statements or entire procedures. Future work will investigate this problem further.

3.2.3 Study 3: Potential savings due to caching. Our objective in this study was to investigate the potential for savings due to caching of slicing information. For this study, we considered the same two subject programs

that we used for Study 1b. For our first study of reuse (Study 3a), we measured the reuse of the slicing information for individual slices. For each subject program \mathcal{P} , statement S , CFG node s , and each variable v referenced at S , we performed a slice with criterion $\langle s, v \rangle$ using our prototype slicer. After each slice, we restarted the slicer; thus, the reuse is measured with respect to single slices from the start-up state of the slicer, and does not include reuse of slicing information computed for previous slices.

As the slice was being computed, we recorded the number of times that `ComputePSlice` was called with a criterion, $crit_{new}$, and the number of times that `ComputePSlice` had to process the criterion that it received, $crit_{processed}$. The $crit_{new}$ represents the total number of times the algorithm would have performed the slicing without reuse, and $crit_{processed}$ represents the number of times the algorithm actually performed the slice. The ratio of $crit_{new} - crit_{processed}$ to $crit_{new}$ gives us a lower bound on the percentage of criterion that we avoid processing. $crit_{processed}$ represents a lower bound on the number of criteria that we avoid processing because our counts do not include the criterion reused in called procedures. To illustrate, consider program `Sum`. Suppose that during slice computation, `ComputePSlice` is called a second time with criterion $\langle 15, sum \rangle$. To compute $PSlice$ for this criterion requires processing criteria, $\langle 19, sum \rangle$ and $\langle 19, j \rangle$, because both `sum` and `j` are relevant at Return node 10b. We count the invocation of `ComputePSlice` with $\langle 15, sum \rangle$ in $crit_{new}$ but not in $crit_{processed}$ because the algorithm uses the cached information. However, we do not count the invocation of `ComputePSlice` with $\langle 19, sum \rangle$ and $\langle 19, j \rangle$, whose computation is also avoided, in any of the counts. Thus, we get a lower bound on the reuse achieved.

Figure 8 represents, as a segmented bar graph, the percentages of criteria that we avoid processing. The graph shows that, for both programs, there is a high percentage of reuse of the slicing information.

For our second study of reuse (Study 3b), we wanted to investigate the reuse that might be possible in an environment where one user is performing multiple slices on the same program or where multiple users are performing slices on the same program. To do this, we considered the same two subject programs that we used for Study 3a. For each subject program \mathcal{P} , each procedure P_i with statements S_1, \dots, S_n , and the variables v referenced at S , we formed three sequences of slicing criteria: (1) slicing criteria for S_1, \dots, S_n ; (2) slicing criteria for S_n, \dots, S_1 ; and (3) slicing criteria for $S_{n/2}, \dots, S_n$, then $S_1, \dots, S_{n/2-1}$. For each sequence, we restarted the slicer, performed the slices in the sequence order, and recorded the reuse, as we did for the previous study,

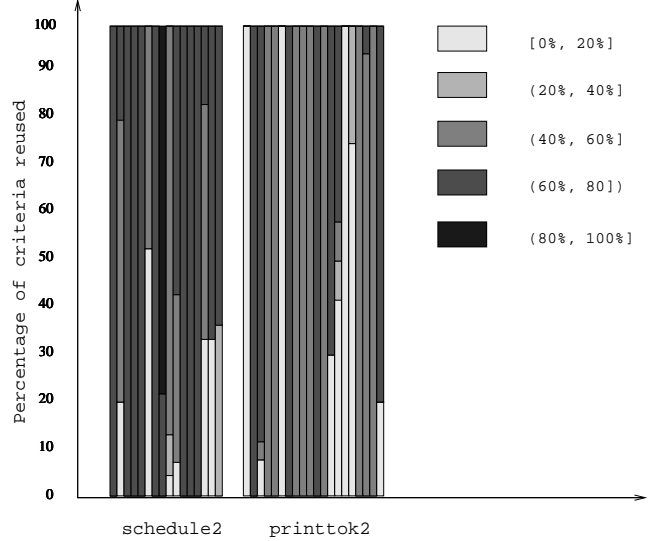


Figure 8: Study 3a – Potential reuse savings per slice.

using $crit_{new}$ and $crit_{processed}$. We also computed the percentages of reuse of slicing information.

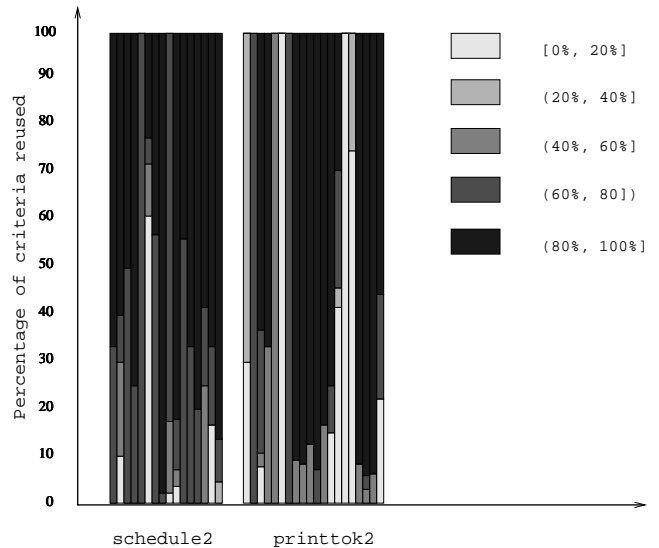


Figure 9: Study 3b – Potential reuse savings per procedure.

Figure 9 represents these percentages as a segmented bar graph. Our results here show an even higher percentage of reuse in most of the procedures in the subject programs that we studied than in Study 3a. These results suggest that providing efficient ways to cache and reuse slicing information may provide significant savings. Future work will investigate ways to maintain this data for both single-user and multi-user environments.

4 CONCLUSIONS AND FUTURE WORK

We presented a control-flow based interprocedural slicing algorithm that has the advantages that it (1) accounts for interprocedural control dependencies in the computation of the slices, and (2) stores and reuses slicing information, including slices, in subsequent slicing. We also presented the results of several studies that considered slice size, slice accuracy, and reuse of slicing information for our slicing technique. Although our subject programs are small, they let us draw some conclusions about our approach, and consider areas for future work.

- Our results illustrate that there exist real programs for which many slices contain a relatively small number of statements, compared to the number of statements in the program. Future work could consider whether this result generalizes to large programs. Future work could also identify those programs and for which applications, slicing will be useful.
- Our results illustrate that there exist real programs for which slices computed with existing interprocedural slicing algorithms miss a significant number of statements that should be included in the slice. However, we do not know to what extent the problem that our algorithm solves is widespread in practice. Future work will address this issue.
- We have incorporated interprocedural control dependence information into our slicing algorithm, but the system dependence graph could be modified to provide similar information. If modified, the two-pass slicing algorithm of [9] would produce the same slices as our algorithm computes.
- Our results show that, for our subject programs, reuse of slicing information could be significant. We did not measure the storage requirements or time saved because of the reuse that our approach provides. An improved prototype will let us continue to experiment.
- We presented reuse studies for our algorithm only. We did not compare our results with results using a system-dependence-graph approach. We believe there are programs and slicing criteria for which our approach will be more efficient than a systems dependence graph approach, and vice versa. Future studies will compare the two approaches, and attempt to determine under what conditions the approaches will be efficient.

5 ACKNOWLEDGMENTS

This work was supported in part by a grant from Microsoft Inc., by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University, and by an Ohio State University Research Foundation Seed Grant. Jim Jones and Tom Jelen helped gather some of the data for the experiments. Bill Griswold and the anonymous reviewers provided helpful comments on an earlier version of the paper.

REFERENCES

- [1] H. Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 302–12, June 1994.
- [2] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proc. of the 18th Int'l. Conf. on Softw. Eng.*, pages 16–27, Mar. 1996.
- [3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. *Conf. Record of the Twentieth ACM Symp. on Prin. of Prog. Lang.*, pages 384–396, Jan. 1993.
- [4] E. Duesterwald and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of 22nd ACM Symposium on Principles of Programming Languages*, pages 37–48, January 1995.
- [5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Sys.*, 9(3):319–349, July 1987.
- [6] R. Gupta and M. L. Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proc. of the 3rd Intl. Symp. on the Found. of Softw. Eng.*, pages 29–40, Oct. 1995.
- [7] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Mar. 1997.
- [8] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis*, Mar. 1998.
- [9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Prog. Lang. and Sys.*, 12(1):26–60, Jan. 1990.
- [10] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. of the 3rd Intl. Symp. on the Found. of Softw. Eng.*, pages 104–115, Oct/Oct. 1995.
- [11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.
- [12] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–79, Sept. 1990.
- [13] T. Reps. *Demand interprocedural program analysis using logic databases*, pages 163–196. Kluwer Academic Publishers, Boston, MA.
- [14] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. *Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis*, pages 169–184, Aug. 1994.
- [15] M. Weiser. Program slicing. *IEEE Trans. on Softw. Eng.*, 10(4):352–357, July 1984.