Proceedings of the International Conference On Software Maintanence'99 Aug. 1999

Reuse-Driven Interprocedural Slicing in the Presence of Pointers and Recursi on

Donglin Liang and Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210 USA
{dliang,harrold}@cis.ohio-state.edu

Abstract

Program slicing, a technique to compute the subset of program statements that can affect the value of a program variable at a specific program point, is widely used in tools to support maintenance activities. To be useful for supporting these activities, a slicing technique must be sufficiently precise and efficient. Harrold and Ci propose a method for improving the efficiency of slicing by reusing slicing information for subsequent slicing. This paper presents an interprocedural slicing algorithm that improves the efficiency and precision of Harrold and Ci's algorithm for programs with pointer variables and recursion. Our empirical results show that our improvements can effectively achieve more reuse in slice computation, for programs with pointers, and can significantly reduce the sizes of slices, for programs with recursion.

Keywords: Slicing, demand-driven, equivalence.

1 Introduction

Maintenance activities, such as program understanding, regression testing, and reverse engineering, require extensive tool support. Program *slicing* [14], a technique to compute the subset of program statements that can affect the value of variable v at point p ($\langle p, v\rangle$ is the *slicing criterion*), is widely used in tools to support these activities. Researchers have proposed many approaches for the computation of program slices (e.g., [3, 5, 6, 14]).

To be useful for supporting maintenance activities, a slicing technique must be sufficiently precise and efficient. Harrold and Ci (HC) propose a method to improve the efficiency of slicing [5]. This method caches and reuses information previously computed by the slicer; thus, it should speed up the computation of subsequent slices. This method is also demand-driven in nature in that it computes only the information that is needed for the computation of a slice; thus, it should be more space and time efficient than exhaustive techniques. However, for programs with pointer variables, the HC algorithm can be inefficient, and for programs with recursion, the HC algorithm can be imprecise.

First, for programs with pointer variables, the HC algorithm can miss opportunities to reuse slicing information. In a procedure, variables pointed to by a pointer variable typically share the same set of data facts. The HC algorithm derives slicing information from this set of data facts. Thus, it repeatedly computes similar slicing information for each of these variables.

Second, for programs with recursion, the HC algorithm can produce imprecise slices. When the algorithm requests slicing information for a non-local variable at a recursive call, it may find that the slicing information is currently being computed and thus, unavailable. In this case, the algorithm computes an overestimate of the statements that affect the slicing criterion. However, without performing analysis in the recursively called procedure, an overestimate must include, in the slice, almost all statements in procedures that are reachable from the recursive call. Thus, in many cases, overestimation can force a large portion of program statements to be included in a slice.

This paper presents approaches for improving the efficiency and precision of the HC algorithm for programs with pointer variables and recursion. Our approach for handling programs with pointer variables uses *equivalence analysis* [10], a technique that partitions the non-local variables accessed in a procedure P into equivalence classes. Equivalence analysis ensures that, at each statement in P and at each statement in procedures directly or indirectly called by P, variables in an equivalence class share the same set of data facts. Therefore, if v_1 and v_2 are in the same equivalence class in P, at any point s in P, v_1 and v_2 are affected by the same set of statements in P and in the procedures directly or indirectly called by P. The slicer can reuse the information computed for $\langle s, v_1\rangle$ when it requests information for $\langle s, v_2\rangle$. To further improve efficiency, our approach also puts variables not accessed in P into another equivalence class. For each of these variables, the slicer immedi-

ately returns an empty slice.

Our approach for handling programs with recursion computes a minimal fixed point for the procedures involved in the recursion. When the slicer requests slicing information that is currently being computed at a recursive call, it records the request, takes the information available at that point in the analysis, and continues processing. Later, when that information is updated, the slicer reprocesses the requests that depend on it. Note that, in the absence of recursion, the slicer works the same as the HC algorithm.

This paper also presents empirical studies in which we investigate the effectiveness of our improvements to the HC algorithm. The studies show that our approach for handling programs with pointer variables can effectively achieve more reuse in slice computation; thus, it can significantly reduce the cost of computing a slice in the presence of pointer variables. The studies also show that, for the programs where the overestimation approach for handling recursion is too conservative, our approach for handling recursion can significantly reduce the size of slices.

2 Slicing in the Absence of Pointers

This section gives an overview of the HC interprocedural slicing algorithm in the absence of pointer variables.

Figure 1 shows `Slicer`, the algorithm that uses cached information to compute interprocedural slices over control flow graphs (CFGs). `Slicer` inputs a slicing criterion, $\langle s, v\rangle$, and outputs the program slice with respect to $\langle s, v\rangle$. `Slicer` first checks $SliceList$ to see whether the slice for criterion $\langle s, v\rangle$ has already been computed (line 1). If no such slice exists, `Slicer` calls `ComputePSlice` to compute the *partial slice*, $PSlice$, and an *interprocedural relevant set* (IRSet), $EVars$, with respect to $\langle s, v\rangle$ (line 2). $PSlice$ contains the nodes representing statements that can affect $\langle s, v\rangle$ in procedure P_s — the procedure that contains node s — or in the procedures that are directly or indirectly called by P_s. $EVars$ contains the global variables and formal parameters of P_s that can affect $\langle s, v\rangle$ from outside P_s. `Slicer` adds $PSlice$ to $Slice$ (line 3). Then, for each call c to P_s, `Slicer` calls `BackBind` to bind $EVars$ back to c, and puts the result in $RelVarTo$ (line 5). For each variable u in $RelVarTo$, `Slicer` creates a new criterion $\langle c, u\rangle$ and invokes itself to compute the slice for $\langle c, u\rangle$ (line 7). `Slicer` combines these slices and stores the result in $SliceList[s, v]$ (line 10). Finally, `Slicer` returns $SliceList[s, v]$ (line 12).

`ComputePSlice` uses a worklist W to compute $PSlice$, the partial slice with respect to $\langle s, u\rangle$. `ComputePSlice` computes two variable sets, $RelIn[N]$ and $RelOut[N]$, for each node N that can reach s in P_s's CFG. `ComputePSlice` first initializes $RelIn[s]$ with u and adds s's CFG predecessors to W (line 13). Then,

```
algorithm   Slicer(s, v)
input       s: a CFG node
            v: a variable
output      program slice w.r.t. ⟨s, v⟩
globals     SliceList[s, v]: slice w.r.t. ⟨s, v⟩
declare     slice, PSlice: sets of CFG nodes
            RelVarTo, CalleeVars: sets of variables
 begin Slicer
 1.   if SliceList[s, v] == NULL then
 2.     (PSlice, EVars) = ComputePSlice(s, v)
 3.     slice = slice ∪ PSlice
 4.     foreach callsite c that calls P_s do
 5.       RelVarTo = BackBind(EVars, c, P_s)
 6.       foreach u in RelVarTo do
 7.         slice = slice∪ Slicer(c, u)
 8.       endfor
 9.     endfor
 10.    SliceList[s, v] = slice
 11.  endif
 12.  return SliceList[s, v]
 end Slicer

function    ComputePSlice(s, u)
input       s: a CFG node
            u: a variable
output      PSlice: partial slice
            CalleeVars: a set of variables
declare     W : a list of CFG nodes
            RelIn[N], RelOut[N]: set of relevant variables
                initially empty
 begin ComputePSlice
 13.  initialize data structures
 14.  while W ≠ φ do
 15.    remove N from W
 16.    update RelOut[N]
 17.    Case N is a call node calling to Procedure P_n:
 18.      foreach v in RelOut[N] do
 19.          (PSlice, RelIn[N]) = (PSlice, RelIn[N])
                  ∪ FindSummary(N, P_n, v)
 20.      endfor
 21.    default:
 22.      update RelIn[N] with Def[N], Ref[N] and Kill[N]
 23.    endcase
 24.    add control-flow, control-dependence predecessors
 25.  endwhile
 26.  return (PSlice, RelIn[P_s.entry])
 end ComputePSlice

function    FindSummary(c, P, v)
input       P: a procedure
            c: a call node that calls P
            v: a variable
output      a pair (partial slice, CalleeVars)
globals     cache[P, v]: pair of (pslice, entryVars)
                previously computed by ComputePSlice
 begin FindSummary
 27.  u = Bind(v, c, P)
 28.  if( cache[P, u]==NULL) then
 29.    cache[P, u] = ComputePSlice(P.exit, u)
 30.  endif
 31.  CalleeVars = BackBind(cache[p, u].entryVars, c,P)
 32.  return (cache[p, u].pslice, CalleeVars)
 end FindSummary
```

Figure 1. `Slicer`: Interprocedural slicing algorithm in the absence of pointers.

for each CFG node N in W, `ComputePSlice` updates $RelOut[N]$ by copying the variables in the $RelIn$ sets of N's CFG successors (lines 15,16). If $RelOut[N]$ changes, `ComputePSlice` propagates the new variables in $RelOut[N]$ to $RelIn[N]$ according to N's type (lines 17-23). If N is a call node to procedure P_n, then for each variable v in $RelOut[N]$, `ComputePSlice` calls `FindSummary` to compute the summary information for

v in P_n (lines 18,19). The summary information contains the partial slice with respect to $\langle P_n.exit, v\rangle$ and the set of variables that can affect v across the call site represented by N. `ComputePSlice` adds the partial slice returned by `FindSummary` to $PSlice$ and adds the set of variables returned by `FindSummary` to $RelIn[N]$. If the partial slice returned by `FindSummary` is not empty, `ComputePSlice` adds N to $PSlice$. If N is a node other than a call node, `ComputePSlice` updates $RelIn[N]$ according to $Def[N]$,$Kill[N]$ and $Ref[N]$ (line 22), where $Def[N]$, $Kill[N]$, and $Ref[N]$ are variables defined, killed and referenced at N. `ComputePSlice` first assigns $RelOut[N]$ to $RelIn[N]$. If $RelOut[N] \cap Def[N] \neq \phi$, `ComputePSlice` removes the variables in $Kill[N]$ from $RelIn[N]$, and adds the variables in $Ref[N]$ in $RelIn[N]$. If $RelOut[N] \cap Def[N] \neq \phi$, `ComputePSlice` also adds N to $PSlice$. After these actions, if $RelIn[N]$ changes, `ComputePSlice` adds N's CFG predecessors in the worklist (line 24). If N is added to $PSlice$, `ComputePSlice` also adds N's control-dependence predecessors to $PSlice$. For each such predecessor, N', `ComputePSlice` adds $Ref[N']$ to $RelIn[N']$. `ComputePSlice` then adds the CFG and control-dependence predecessors of N' accordingly.

`FindSummary` uses a cache to compute summary information for a variable v at a call node c that calls P. `FindSummary` first calls `Bind` to bind v from c to a variable u in P (line 27) and then checks the cache against $\langle P, u\rangle$ (line 28). If the algorithm finds that the cache for $\langle P, u\rangle$ is empty, it calls `ComputePSlice` to compute the partial slice and the IRSet with respect to $\langle P.exit, u\rangle$, and stores the result in the cache (line 29). Then, `FindSummary` calls `BackBind` to bind $cache[P, v].entryVars$ back to call node c and return it with the partial slice $cache[P, v].pslice$ (lines 31-32).

3 Slicing in the Presence of Pointers

This section first discusses the inefficiencies that pointers cause, then gives an overview of equivalence analysis, and finally presents improvements to the HC algorithm.

3.1 Program slicing in the presence of pointers

Program slicing, like other data-flow analyses, requires alias information to ensure safety. An *alias* occurs between two names that can access one memory location at a program point. Aliases can be resolved using *flow-sensitive* (e.g., [8]) or *flow-insensitive* (e.g., [1, 9, 13]) algorithms. Flow-sensitive algorithms compute more precise alias information at a higher cost than flow-insensitive algorithms.

After the alias information is computed, many slicing techniques (and other data-flow analyses) can be modified

```
Program 1                        15. f(int *p) {
 1. int j,sum;                   16.    if( j<0 )
 2. main() {                     17.      incr(&j);
 3.    int sum1, i1,i2;          18.    sum = sum + j;
 4.    reset(&sum);              19.    incr(p);
 5.    reset(&i1);               20.    read(&j);
 6.    read(&j);                 21. }
 7.    while(i1<10)
 8.      f(&i1);                 22. incr(int *q){
 9.    sum1 = sum;               23.    *q = *q+1;
10.    reset(&i2);               24. }
11.    while(i2<20)
12.      f(&i2);                 25. reset( int *s){
13.    write(sum1);              26.    *s = 0;
14. }                            27. }
```

procedure	equivalence class for non-local
main()	{sum}, {j}
f()	{i1,i2}, {sum}, {j}
incr()	{i1,i2,j}
reset()	{i1,i2,sum}

Figure 2. Example Program 1 (top) and the equivalence classes for Program 1 (bottom).

[2, 12] to handle pointers in a program using a method similar to the following: before the algorithm computes data facts at a statement, it replaces every dereference of pointers with the set of memory locations that may be aliased to the dereference. For example, to compute the DEF set (the set of variables defined at a statement) for statement 26 in Figure 2, a slicer first replaces the pointer dereference `*s` with {`i1, i2, sum`}, the set of memory locations that may be aliased to `*s` at statement 26. Because statement 26 modifies `*s`, the slicer concludes that statement 26 modifies memory locations `i1`, `i2`, and `sum`, and includes these memory locations in the DEF set for statement 26.

The above method of handling pointers, although safe, can increase the cost of interprocedural data-flow analyses. Because most memory locations accessed through dereferences of parameter pointers or global pointers are non-local to the procedure in which the access occurs, pointer variables can greatly increase the data-flow information computed for the procedure. For example, we found [9] that a procedure in a C program with 25,002 lines of code can modify, on average, 199 non-local memory locations when the aliases are computed with Steensgaard's algorithm. This means that interprocedural data-flow analyses, such as the reuse-driven slicer, must compute large amounts of summary information at call statements, which could significantly degrade the performance of the analysis.

3.2 Equivalence analysis

Observing that, at a statement, memory locations accessed through a dereference of a pointer typically share the same data facts, we introduce an equivalence relation [10], induced by the access patterns of the memory locations.

Memory locations are accessed by *object names* [8], which consist of a program variable and a sequence of

dereferences and field accesses. An object name obj is *indirect* if there is at least one dereference in obj; otherwise, obj is *direct*. For example, in Figure 2, `*s` is an indirect object name, and `s` is a direct object name.

A memory location is always *equivalent* to itself. A memory location l_1 is *equivalent* to another memory location l_2 in procedure P if (1) if l_1 is accessed by an object name obj at a statement S in P, then l_2 is also accessed by obj at S, and vice versa; and (2) l_1 and l_2 are equivalent in the procedures called by P if l_1 and l_2 are accessed by those procedures. For example, `i1` is equivalent to `i2` in `reset()` in Figure 2 according to these conditions.

We have developed an algorithm to compute a set of equivalence classes for each procedure [10] in a program. The table on the bottom of Figure 2 shows the equivalence classes computed by our algorithm for the procedures in `Program 1`. We use equivalence classes to improve the performance of many data-flow analyses, including reuse-driven program slicing.

3.3 Using equivalence classes to improve program slicing in the presence of pointers

We can use equivalence classes to improve HC algorithm so that we can achieve more reuse. `ComputePSlice` computes similar partial slices and IRSets for memory locations in an equivalence class. Assume that l_1 and l_2 are equivalent in procedure P. If l_1 is in $DEF[N]$, $REF[N]$, or $KILL[N]$ of a CFG node N in P, then l_2 is also in the same set, or vice versa. Thus, `ComputePSlice` computes the same set of CFG nodes for criterion $\langle s, l_1\rangle$ and for criterion $\langle s, l_2\rangle$, where s is a CFG node in P. When we need to compute the partial slice for $\langle s, l_2\rangle$, we can directly reuse the partial slice that is computed for $\langle s, l_1\rangle$. However, `ComputePSlice` can compute two different IRSets for these two criteria. For example, although `i1` and `i2` are equivalent in `reset()` in Figure 2, `ComputePSlice` computes {`i1`} as the IRSet for ⟨27,`i1`⟩ and {`i2`} as the IRSet for ⟨27,`i2`⟩. Let S_1 be the IRSet with respect to $\langle s, l_1\rangle$ and S_2 be the IRSet with respect to $\langle s, l_2\rangle$, we can derive S_2 from S_1: if $l_2 \notin S_1$ and $l_1 \in S_1$, then $S_2 = (S_1 - \{l_1\}) \cup \{l_2\}$; otherwise, $S_2 = S_1$. Thus, we can reuse the IRSet computed for $\langle s, l_1\rangle$ when we compute the IRSet for $\langle s, l_2\rangle$.

Figure 3 shows `FindSummaryPT`, an enhanced version of `FindSummary` (Figure 1) that uses equivalence classes to achieve more reuse in the computation of summary information for variable v at call node c. `FindSummaryPT` binds v from c to variable u in the called procedure P, and then calls `FindRep` to compute a representative memory location r for the equivalence class of P that contains u (lines 1-2). `FindSummaryPT` then checks the cache with $\langle P, r\rangle$ (line 3). If no cached information is available for $\langle P, r\rangle$, `FindSummaryPT` calls `ComputePSlice` (Figure 1) to compute the partial slice and the IRSet for $\langle P, r\rangle$ (line 4). Then, `FindSummaryPT` calls `Derive` to derive the IRSet for u from that of r (line 6). Finally, `FindSummaryPT` binds u's IRSet back to c and returns (lines 7-8).

```
function  FindSummaryPT(c, P, v)
input     P: a procedure
          c: a call node that calls P
          v: a variable
output    a pair (partial slice, CalleeVars)
globals   cache[P, v]: pair of (pslice, entryVars)
              previously computed by ComputePSlice
declare   r: the representative location of u
begin FindSummaryPT
1.   u = Bind(v, c, P)
2.   r = FindRep(P, u)
3.   if( cache[P, r]==NULL) then
4.      cache[P, r] = ComputePSlice(P.exit, r)
5.   endif
6.   EVars = Derive(P, cache[P, r].entryVars, u, r)
7.   CalleeVars = BackBind(EVars, c,P)
8.   return (cache[p, r].pslice, CalleeVars)
end FindSummaryPT

function  Derive(P, entryVars, u, r)
input     P: a procedure
          entryVars: IRSet for ⟨P, r⟩
          u, r: two variables
output    IRSet for ⟨P, u⟩
begin Derive
9.   if u in entryVars or r not in entryVars then
10.     EVars = entryVars
11.  else
12.     EVars = entryVars − {r} ∪ {u}
13.  endif
14.  return EVars
end Derive
```

Figure 3. `FindSummaryPT`:Replaces `FindSummary` of Figure 1 to gain more reuse.

To gain more reuse using equivalence classes, we must change the reuse strategy used in `Slicer` (Figure 1). Given that two memory locations l_1 and l_2 are equivalent in P, `Slicer` can compute different slices for criteria $\langle s, l_1\rangle$ and $\langle s, l_2\rangle$, where s is a CFG node in P. These two slices can be different because l_1 and l_2 can be nonequivalent in the procedures that call P. However, these two slices share the same set of CFG nodes within P. If we change the reuse strategy in `Slicer` so that it reuses partial slices instead of complete slices, then we can use equivalence classes to achieve more reuse.

Figure 4 shows `SlicerPT`, an enhanced version of `Slicer` (Figure 1) that reuses partial slices in the presence of pointers. `SlicerPT` computes the slice with respect to the input slicing criterion. `SlicerPT` also computes a variable set, $RelSet[c]$, for each call c that can reach the slicing criterion in the program. $RelSet[c]$ is used to avoid propagating a memory location from c twice.

`SlicerPT` uses a worklist W_{inter} to iterate over the groups of slicing criteria. `SlicerPT` first initializes W_{inter} with the original slicing criterion (line 1). `SlicerPT` then removes a group of slicing criteria (s, V), where s is a CFG node and V is a set of memory locations,

```
algorithm   SlicerPT(s, v)
input       s: a CFG node
            v: a variable
output      slice: program slice w.r.t. ⟨s, v⟩
global      SliceList[s, v]: a list containing pairs
              of (partial slice, IRSet) for criterion ⟨s, v⟩
declare     W_inter: a list containing pairs
              of (CFG node, varialbes)
            RelSet[c]: the set of memory locations that have
              been propagated from callnode c
begin SlicerPT
1.    add (s, {v}) to W_inter
2.    while W_inter ≠ φ do
3.      remove (s, V) from W_inter
4.      foreach u in V do
5.        r = FindRep(P, u)
6.        if SliceList[s, r]==NULL then
7.          SliceList[s, r] = ComputePSlice(s, r)
8.        endif
9.        EVars = Derive(P, SliceList[s, r].entryVars, u, r)
10.       slice = slice ∪ SliceList[s, r].PSlice
11.       for each callsite c that calls P_s do
12.         RelVarTo = BackBind(EVars, c)
13.         add (c, RelVarTo − RelSet[c]) to W_inter
14.         add RelVarTo to RelSet[c]
15.       endfor
16.     endfor
17.   endwhile
18.   return slice
end SlicerPT
```

Figure 4. SlicerPT: An interprocedural slicing algorithm in the presence of pointers.

from the worklist (line 3). For each u in V, SlicerPT finds a representative memory location, r, which represents the equivalence class that contains u (line 5). SlicerPT checks $SliceList$ using $\langle s, r\rangle$ (line 6). If the partial slice and the IRSet for $\langle s, r\rangle$ are not yet computed, SlicerPT calls ComputePSlice to compute the partial slice and the IRSet (line 7). SlicerPT then calls Derive to derive the IRSet for $\langle s, u\rangle$ from that of $\langle s, r\rangle$ (line 9). SlicerPT adds the partial slice to the slice and binds the IRSet back to the call nodes that call the procedure containing s (lines 10-15). At each call node c, SlicerPT checks to see whether there are memory locations that are not in $RelSet[c]$. If there are such memory locations, SlicerPT creates a new criterion group for the call node, adds the criterion group to W_{inter}, and updates $RelSet[c]$. SlicerPT returns the slice if W_{inter} is empty (line 17).

4 Slicing in the Presence of Recursion

This section first gives an overview of our approach for handling recursion in the reuse-driven slicer, and it then gives a detailed description of the approach.

4.1 Overview

The reuse-driven slicing algorithm described in Figure 1 might not terminate in the presence of recursion. First, the computation of $SliceList[s, v]$ depends on the computations of other entries in $SliceList$. In the presence of recursion, the computation of $SliceList[s, v]$ can depend on the computation itself. In this case, the computation of $SliceList[s, v]$ cannot be completed. Second, similar to the computation of $SliceList[s, v]$, the computation of $cache[P, v]$ depends on the computations of other entries in $cache$. In the presence of recursion, the computation of $cache[P, v]$ can depend on the computation itself. In this case, the computation of $cache[P, v]$ cannot be completed.

```
Program 2
1. int g1, g2, g, e;
2. main() {
3.    g1=0;
4.    g2=0;
5.    g=0;
6.    e=0;
7.    f(0);
8.    g1++;
9. }
10. f(int a) {
11.    if( a>0 ) {
12.      e = e+1;
13.      g1 = g2;
14.    }
15.    else {
16.      g2 = g1+g;
17.      f(a++);
18.    }
19. }
```

Figure 5. Example Program 2.

The HC algorithm uses overestimation to solve the non-termination problem: when the slicer encounters a recursive call, it includes in the slice, all statements that can affect any global variable in the procedures that the recursive call can reach, and it includes in the IRSet, all formal parameters and all global memory locations that are referenced by those procedures. For example, when the HC algorithm processes statement 7 in Program 2 (Figure 5), it includes all statements in f() in the slice and g1,g2,g,e,a in the IRSet. This overestimate can be very imprecise because it could include most of the statements in the procedures that can be reached from the recursive call.

SlicerPT, shown in Figure 4, solves Slicer's non-termination problem by computing a fixed point. Once $RelSet[c]$ stabilizes at each call node c, the worklist becomes empty and SlicerPT terminates.

The intuition for our approach that guarantees the termination of FindSummary and ComputePSlice is that, once FindSummary finds that, to compute $cache[P, u]$, it is requesting information in $cache[Q, v]$, which is being computed, and thus, only partially available, it uses underestimation: it includes the information available so far for $cache[Q, v]$ and continues to compute $cache[P, u]$. To ensure the safety of the algorithm, FindSummary puts $\langle P, u\rangle$ in $cache[Q, v]$'s dependence list. Later, when $cache[Q, v]$ is updated, the new information in $cache[Q, u]$ is used to recompute $cache[P, u]$.

For example, when ComputePSlice computes $cache$[f,g1] for the program in Figure 5, it invokes FindSummary to find the summary information for ⟨17,g1⟩. Because statement 17 calls f(), FindSummary checks the cache with ⟨f,g1⟩ and finds that $cache$[f,g1] is being computed; thus, it returns the information available so far for $cache$[f,g1] and put $cache$[f,g1] in the dependence list of $cache$[f,g1]. ComputePSlice con-

tinues processing the other statements in function `f()` and updates $cache$[`f,g1`] with partial slice {10,11,13} and IRSet {`g2,a`}. Because $cache$[`f,g1`] depends on itself at statement 17, the slicer processes statement 17 again and updates $cache$[`f,g1`] by adding statements 16 and 17 to $cache$[`f,g1`]$.PSlice$ and adding `g1` and `g` to $cache$[`f,g1`]$.entryVars$.

4.2 Detailed description

Figures 6 and 7 show `FindSummaryRecur` and `ComputePSliceRecur`, the enhanced versions of `FindSummary` and `ComputePSlice` (Figure 1), respectively, that handle programs with recursion. Before we discuss these functions, we describe some data structures that these functions use.

During the computation, $cache[P, u]$ can be in several states. By default, $cache[P, u]$ is in the `notcomputing` state. When `ComputePSliceRecur` is invoked to compute $cache[P, u]$, $cache[P, u]$ moves to the `computing` state. After `ComputePSliceRecur` returns from computing $cache[P, u]$, if $cache[P, u]$ directly or indirectly depends on some entries in $cache$ that are in the `computing` state, then $cache[P, u]$ moves to the `pending` state. Otherwise, $cache[P, u]$ moves to the `complete` state. `pending` state indicates that $cache[P, u]$ could be an underestimate. Therefore, in the absence of recursion, $cache[P, u]$ is never in the `pending` state.

We introduce $dependence[P, u]$, a list of tuples ($proc$,var,$call$,ins,$outs$) to indicate that the computation of $cache[proc, var]$ directly depends on $cache[P, u]$. The tuple means that, when `ComputePSliceRecur` is invoked to compute $cache[proc, var]$, it uses an underestimate at call node $call$ because $cache[P, u]$ is not totally computed. When this occurs, `ComputePSliceRecur` stores $RelIn[call]$ in ins and $RelOut[call]$ in $outs$ so that the summary information for $call$ can be updated later. For example, when the algorithm finds that $cache$[`f,g1`] depends on itself at statement 17, a tuple (`f,g1,17,`ϕ,{`g1`}) is added to $dependence$[`f,g1`].

We add another field, $actives$, to $cache[P, u]$ to store the pairs of $\langle proc, var\rangle$. Each of these pairs represents that the computation of $cache[P, u]$ cannot be completed because it directly or indirectly depends on $cache[proc, var]$, which is being computed by `ComputePSliceRecur`. When $cache[proc, var]$ moves from `computing` to another state, $\langle proc, var\rangle$ is removed from the $actives$ fields of any entry in $cache$. The $actives$ field is used to detect whether $cache[P, u]$ should move to the `complete` state; when $cache[P, u].actives$ becomes empty, $cache[P, u]$ moves from the `pending` state to the `complete` state.

`FindSummaryRecur` (Figure 6) binds v to u at call node c in procedure P and then checks $status[P, u]$, the state of $cache[P, u]$ (lines 19-20). If $status[P, u]$ is `notcomputing`, `FindSummaryRecur` invokes `ComputeSummary` to compute $cache[P, u]$ (line 21). `FindSummaryRecur` then rechecks $status[P, u]$ (line 23). If $status[P, u]$ is `computing` or `pending`, then `FindSummaryRecur` creates $waiting$, a tuple (c,P,u), to indicate that, because $cache[P, u]$ is not complete, summary information for call node c is an underestimate (line 24).

```
function  FindSummaryRecur(c, P, v)
input     c: callnode that calls P
          v: a variable
output    a tuple of (partial slice, CalleeVars,waiting, actives)
global    status[P, u]: status of computing cache[P, u]
          cache[P, u]: a tuple of (pslice,entryVars, actives)
 begin FindSummaryRecur
19.   u = Bind(v, c, P)
20.   if status[P, u] == notcomputing then
21.     ComputeSummary(P, u)
22.   endif
23.   if status[P, u] == computing or pending then
24.     waiting = (c, P, u)
25.     actives=(status[P, u]==computing)?
            {P, u}:cache[P, u].actives
26.   endif
27.   calleevars = BackBind(cache[P, u].entryVars, c, P)
28.   return (cache[P, u].pslice, calleevars, waiting, actives)
 end FindSummaryRecur

procedure  ComputeSummary(P, u)
input      P: a procedure
           u: a variable
output     updated status[P, u] and cache[P, u]
 begin ComputeSummary
29.   status[P, u] = computing
30.   cache[P, u].pslice, cache[P, u].entryVars, actives,
          pendings = ComputePSliceRecur(P.exit, u)
31.   CreateDependences(P, u, pendings)
32.   if ⟨P, u⟩ in actives then
33.     actives ∪ = ResolvePending(P, u)
34.     remove ⟨P, u⟩ from actives
35.   endif
36.   cache[P, u].actives =actives
37.   status[P, u]=(actives is empty)?complete:pending
38.   PropagateActives(cache[P, u].actives)
39.   RemoveActives(⟨P, u⟩)
 end ComputeSummary

function  ResolvePendings(P, u)
input     P: a procedure
          u: a variable
output    actives: a set of pairs (procedure, variable)
global    dependence[P, u]: a set of tuples
          (proc, variable, call, variable set, variable set)
 begin ResolvePendings
40.   W = dependence[P, u]
41.   while W ≠ NULL do
42.     remove (Q, var, c, outs, ins) from W;pendings = φ
43.     foreach v in outs do
44.       (cache[Q, var].PSlice, Vars, waitings, actives)
              ∪ = FindSummaryRecur(c,v)
45.     endfor
46.     foreach v in Vars − ins do
47.       cache[Q, var].PSlice, cache[Q, var].entryVars, actives,
              pendings ∪ = ComputePSliceRecur(call, v)
48.     endfor
49.     update ins in the tuple with Vars
50.     CreateDependences(Q, var, pendings)
51.     if cache[Q, var] changes then
52.       W = W ∪ dependence[Q, var]
53.     endif
54.   endwhile
55.   return actives
 end ResolvePendings
```

Figure 6. `FindSummaryRecur()`: Computes summary information in the presence of recursion.

```
function   ComputePSliceRecur(s, v)
input      s: a CFG node
           v: a variable
output     PSlice: partial slice
           CalleeVars: a set of variables
           actives: a set of pairs (procedure, variable)
           pendings: a set of tuples (call, RelOut[call]
                RelIn[call],proc, variable)
declare    W : a list of CFG nodes
           RelIn[N], RelOut[N]: sets of relevant variables
begin ComputePSliceRecur
56.   initialize data structures
57.   while W ≠ NULL do
58.     remove N from W
59.     update RelOut[N]
60.     case N is a call node to procedure P_n:
61.       foreach u in RelOut[N] do
62.         (PSlice, RelIn[N], waitings, actives) ∪ =
                FindSummaryRecur(N, Pn, u)
63.       endfor
64.     default:
65.       update RelIn[N] with Def[N], Ref[N] and Kill[N]
66.     endcase
67.     add control-flow, control-dependence predecessors
68.   endwhile
69.   foreach (c, P, u) in waitings do
70.     pendings = pendings ∪ (c, RelOut[c], RelIn[c], P, u)
71.   endfor
72.   return PSlice, RelIn[P_s.entry], actives, pendings
end ComputePSliceRecur
```

Figure 7. `ComputePSliceRecur`: Compute partial slice in the presence of recursion.

If $status[P, u]$ is `computing`, `FindSummaryRecur` puts $\{P, u\}$ in $actives$ to indicate that c must be processed again when $cache[P, u]$ is updated (line 25). Otherwise, `FindSummaryRecur` puts $cache[P, u].actives$ in $actives$. Finally, `FindSummaryRecur` binds $cache[P, u].entryVars$ back to variables $CalleeVars$ at c, and returns the partial slice, $CalleeVars$, $waiting$, and $actives$ (lines 27-28). For example, when `FindSummaryRecur` is invoked to find the summary information for `g1` at statement 17, it finds that $status$[`f`,`g1`] is `computing`. Therefore, `FindSummaryRecur` puts {`f`,`g1`} in $actives$, creates (17,`f`,`g1`) as $waiting$ and return $actives$ and $waiting$ together with the partial slice and IRSet, which are empty in this case.

`ComputeSummary` (Figure 6) sets $status[P, u]$ to `computing` and then invokes `ComputePSliceRecur` to compute the partial slice and the IRSet for $\langle P, u\rangle$ (lines 29-30). `ComputePSliceRecur` (Figure 7) enhances `ComputePSlicer` so that it can return the set of $actives$ that it collects from the return of `FindSummaryRecur` at the call nodes. `ComputePSliceRecur` also returns a list of $pendings$. Each pending is a tuple (c,$outs$,ins,$proc$,var) in which c is a call node to procedure $proc$, $outs$ is the $RelOut[c]$, and ins is the $RelIn[c]$ after c is processed by `ComputePSliceRecur` to compute $cache[proc, var]$. `ComputePSliceRecur` creates a pending by adding the $RelOut[c]$ and $RelIn[c]$ to the $waiting$ tuple $(c, proc, var)$ returned by `FindSummaryRecur` when c is processed. For example, `ComputeSummary` receives (17,`f`,`g1`) in $waiting$ when it invokes `FindSummaryRecur` to find the summary information for `g1` at statement 17. `ComputePSliceRecur` creates a pending (17,{`g1`},ϕ,`f`,`g1`).

After `ComputePSliceRecur` returns, `ComputeSummary` calls `CreateDependences` to create an entry (P,u,c,ins,$outs$) in $dependence[proc, var]$ for each pending (c,$outs$,ins,$proc$,var) to indicate that the computation of $cache[P, u]$ depends on $cache[proc, var]$ and requires further updating at c (line 31). For example, `ComputeSummary` creates (`f`,`g1`,`17`,ϕ,{`g1`}) in $dependence$[`f`,`g1`] after it receives (17,{`g1`},ϕ,`f`,`g1`) to indicate that $cache$[`f`,`g1`] depends on itself.

`ComputeSummary` also checks $actives$ (line 32). If $\langle P, u\rangle$ is in $actives$, `ComputeSummary` invokes `ResolvePendings` to update the entries of $cache$ that directly or indirectly depend on $cache[P, u]$ (line 33). For example, when `ComputeSummary` computes $cache$[`f`,`g1`], it finds ⟨`f`,`g1`⟩ in $actives$. Thus, it invokes `ResolvePendings` to update the entries. `ComputeSummary` also removes $\langle P, u\rangle$ from $actives$ (line 34). Then, `ComputeSummary` stores $actives$ in $cache[P, u].actives$ (line 36). If $cache[P, u].actives$ is empty, then $status[P, u]$ is set `complete`; otherwise, $status[P, u]$ is set to `pending` (line 37). If $cache[P, u].actives$ is not empty, `ComputeSummary` invokes `PropagateActives` to propagates the items in $cache[P, u].actives$ to the $actives$ fields of the entries in $cache$ that directly or indirectly depend on $cache[P, u]$ (line 38). `ComputeSummary` then invokes `RemoveActives` to remove $\langle P, u\rangle$ in the $actives$ fields of those entries (line 39). If the $actives$ field of an entry becomes empty after $\langle P, u\rangle$ is removed, the status of the entry is set to `complete`.

`ResolvePendings` uses a worklist to update entries in $cache$ that directly or indirectly depend on $cache[P, u]$. The worklist is initialized with entries in $dependence[P, u]$ (line 40). For each tuple (Q,var,c,ins,$outs$) in the worklist, `ResolvePendings` invokes `FindSummaryRecur` to get the summary information for each variable in $outs$ (lines 43-45). If `FindSummaryRecur` returns some variables that are not in the ins, `ResolvePendings` invokes `ComputePSliceRecur` to update $cache[Q, var]$ (line 47). `ResolvePendings` then updates ins in the tuple (line 49). If `ComputePSliceRecur` returns new pendings, `ResolvePendings` creates new dependences from the pendings (line 50). If $cache[Q, var]$ changes, `ResolvePendings` updates the worklist with the entries in $dependence[Q, var]$ for further updating (line 52). Finally, `ResolvePendings` returns the $actives$ it collects when it processes the call statements with `ComputePSliceRecur` (line 55).

For example, `ResolvePendings` initializes the work-

Program	Lines of Code	Number of CFG Nodes	Funcs in Recursions	Funcs Reached
loader	1132	819	2	3
ansitape	1596	1087	1	1
dixie	2100	1357	3	9
learn	1600	1596	0	0
unzip	4075	1892	2	4
lharc	3235	2539	3	58
flex	6902	3762	5	5
space	11474	5601	0	0
bison†	7893	6533	3	4
larn†	9966	11796	3	72
mpeg_play†	17263	11864	3	3

† Alias information for Landi and Ryder's algorithm is not available for the program because the time required for the analysis exceeded our limit.

Table 1. Information about the subject programs.

list with (f,g1,17,ϕ,{g1}) when it is invoked to update the entries in $cache$ that depend on $cache$[f,g1]. Then, ResolvePendings finds the summary information for g1 at statement 17 using FindSummaryRecur, which returns g1 and a. ResolvePendings then invokes ComputePSliceRecur to propagate g1 and a from statement 17, and obtains partial slice {11,16,17} and IRSet {g1,g,a}. Thus, ResolvePendings updates $cache$[f,g1] with the partial slice and the IRSet. Because $cache$[f,g1] changes, ResolvePendings adds (f,g1,17,{g1,a},{g1}) to the worklist and continues processing until there is no change to $cache$[f,g1].

5 Empirical Studies

To investigate the efficiency and effectiveness of our two improvements to the HC algorithm, we developed a prototype, using the PROLANGS Analysis Framework (PAF) [4], that implements our approaches. We conducted several studies and collected the data on a Sun Ultra 30 workstation with 640 MB of physical memory. Our prototype resolves pointer dereferences using the alias information provided by the following algorithms: Steensgaaard's algorithm (ST) [13], Liang and Harrold's algorithm (LH) [9], Andersen's algorithm (AND) [1], and Landi and Ryder's algorithm (LR) [8]. Steensgaard's, Liang and Harrold's, and Andersen's algorithms are flow-insensitive; Landi and Ryder's algorithm is flow-sensitive.[1] Our prototype stores cached information only during the computation of the current slice; after a slice is computed, the cached information is deleted. Using this scheme, information cached by Slicer is seldom reused because, when Slicer encounters a call node, it propagates only the memory locations that are new to that call node. Therefore, our prototype does not cache information in Slicer.

Table 1 shows a subset of the subject programs we used in our studies. The last two columns in the table give information about recursion in the programs: column 4 shows

[1] Details of these algorithms and how they compare to each other can be found in [9].

program	alias	S	S'	R_S	T	T'	R_T
loader†	ST	16.0	4.7	70.8	21.9	8.0	63.3
	LH	9.5	4.8	49.8	7.3	4.3	40.5
	AND	9.5	4.8	49.8	7.3	4.3	40.6
	LR	9.7	4.8	50.5	7.7	4.6	40.1
ansi-tape†	ST	21.9	10.1	53.8	18.5	6.3	65.7
	LH	18.1	11.9	33.8	11.9	6.2	47.8
	AND	16.2	11.2	30.9	6.5	4.5	29.7
	LR	15.6	11.2	28.7	6.2	4.4	28.8
dixie†	ST	26.0	6.2	76.1	44.0	9.4	78.7
	LH	13.7	6.9	49.9	15.0	6.2	59.1
	AND	11.6	7.2	38.2	9.4	5.4	42.3
	LR	11.5	7.2	36.8	8.8	5.0	43.3
learn†	ST	12.3	4.2	65.8	33.8	12.5	63.0
	LH	8.6	4.8	45.1	23.2	13.1	43.9
	AND	6.9	4.6	33.8	12.8	10.6	17.6
	LR	7.0	5.0	28.5	18.0	15.0	16.7
unzip†	ST	24.2	9.1	62.4	42.5	16.1	62.1
	LH	14.4	9.6	33.8	17.0	11.1	34.8
	AND	13.8	9.7	29.4	12.9	9.9	23.3
	LR	10.9	8.7	20.0	10.6	8.7	17.8
lharc†	ST	12.0	7.2	40.0	10.7	5.8	45.7
	LH	10.9	7.4	32.3	8.4	5.3	37.0
	AND	10.1	7.3	27.2	7.7	5.2	32.6
	LR	7.4	6.1	16.6	6.5	4.6	30.0
flex‡	ST	22.3	13.4	40.0	760.3	523.3	31.2
	LH	16.8	13.8	18.2	566.5	484.6	14.4
	AND	16.2	13.9	14.2	512.3	475.7	7.1
	LR	14.6	13.5	8.1	582.8	517.0	11.3
space‡	ST	77.7	14.2	81.8	1528	274.3	82.1
	LH	72.5	17.4	76.0	649.1	179.5	72.3
	AND	70.9	19.1	73.1	642.0	182.6	71.6
	LR	50.7	18.9	62.6	553.7	176.8	68.1
bison‡	ST	14.8	10.0	32.2	122.6	54.1	55.9
	LH	14.4	10.0	30.8	108.1	49.3	54.4
	AND	12.4	10.6	14.4	61.2	45.6	25.5
larn‡	ST	58.1	20.4	64.9	3477	1531	56.0
	LH	27.9	21.0	24.8	1076	807.2	25.0
	AND	26.7	21.0	21.3	902.4	760.3	15.7
mpeg-play‡	ST	15.7	13.0	16.9	289.9	198.0	31.7
	LH	15.1	13.6	10.3	131.5	126.7	3.6
	AND	15.0	13.6	9.2	136.6	131.0	4.1

†Data are collected from all slices of the program.
‡Data are collected from one slice.

Table 2. Average size of GMOD and reduced GMOD and average time in seconds of computing a slice.

the number of recursive functions and column 5 shows the number of functions that can be reached by recursive calls. From the table, we can see that, even for a program with a small number of recursive functions, the number of functions that can be reached from recursive calls can be very large. For this type of programs, we expect that the overestimate will yield a very imprecise result.

5.1 Study 1

In Study 1, we investigated the effectiveness of using equivalence classes to achieve more reuse in the computation of slices. We compared the average number of nonlocal memory locations that may be modified by a procedure (GMOD) with the average number of memory locations that remain after all memory locations except the representative memory locations for equivalence classes (Reduced GMOD) are removed from the GMOD set. Because our algorithm improves the HC algorithm by computing summary information only for memory locations in the Re-

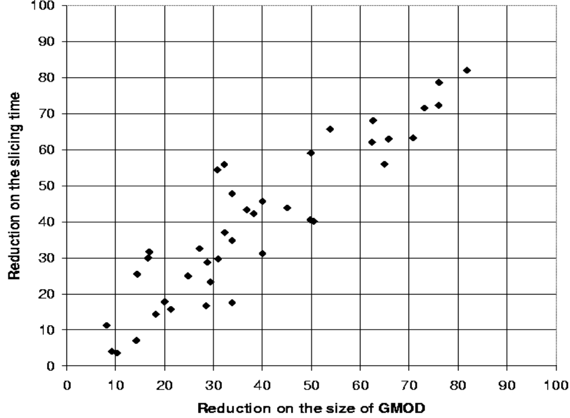

Figure 8. Correlation of the reduction on the size of GMOD and slicing time.

duced GMOD sets, the reduction in the sizes of the GMOD sets indicates the effectiveness of using equivalence classes to achieve more reuse. We also compared the average time of computing a slice without using equivalence classes with the average time of computing a slice using equivalence classes. For each subject program, we ran the HC slicer and our slicer using alias information provided by each of the four alias-analysis algorithms and collected the data.

Table 2 shows the results of this study: the second column shows the alias-analysis algorithm used; the third column shows S, the average size of GMOD sets; the fourth column shows S', the average size of the Reduced GMOD sets; and the fifth column shows R_S, the percentage reduction of the GMOD sets using equivalence classes. The table shows that, for the programs we studied, equivalence analysis often groups several memory locations into one equivalence class. Thus, using equivalence classes can effectively achieve more reuse in the computation of slices.

In the table, the sixth column shows T, the average time of computing a slice with the GMOD sets, the seventh column shows T', the average time of computing a slice with the Reduced GMOD sets, and the eighth column shows R_T, the percentage of reduction in the average time. The table shows that, for many programs, using equivalence classes can significantly reduce the cost of computing a slice. The table also shows an almost-linear relation between the reduction in the size of GMOD sets and the reduction in the time of computing a slice; such a relation can be visualized with the scatter diagram in Figure 8. This result suggests that achieving more reuse using equivalence analysis can effectively improve the performance of the HC algorithm.

5.2 Study 2

In Study 2, we investigated the effectiveness of our approach for handling recursion to improve the precision of slicing in the presence of recursion. We compared the average size (*size*) of a slice computed using the overestimation approach (HC) with the average size of a slice computed using our approach (Recur). We also compared the average time (*time*) to compute a slice using these two approaches. Similar to Study 1, we ran the slicers using alias information provided by each of the four alias-analysis algorithms.

Table 3 shows the results of this study.[2] The table shows that, for subject programs in which only a small number of functions can be reached by recursive calls (e.g. `loader`), our approach computes the same size slices as the overestimation approach. For these programs, the two approaches use a similar amount of time to compute a slice. The table also shows that, for subject programs in which a large number of functions can be reached by recursive calls (e.g. `lharc`), our approach computes significantly smaller slices than the overestimation approach. The table further shows that our approach might even improve the performance of the slicer on some of these programs. Among the four subject programs on which our approach computes smaller slices, our approach runs even faster than the overestimation approach on two of them (`flex,larn`). This result seems reasonable because the overestimation approach must take all non-local memory locations referenced in the procedures reached by a recursive call as a safe estimate of the IRSet. This estimate of IRSet can be much larger than the real IRSet for this recursive call. Thus, using this approach, the slicer might have to propagate additional memory locations throughout the rest of the program after the recursive call.

6 Related Work

Several researchers have reported techniques to slice recursive programs. Hwang et al. [7] proposed an algorithm that inlines each (recursive) call with the procedure body, and computes a slice until a fixed point is reached. However, in the worst case, this algorithm runs in time exponential in the size of the program. This algorithm also handles only the case in which a procedure directly calls the procedure itself; the algorithm must be modified to handle the case in which a procedure indirectly calls itself. Our approach has advantages over Hwang et al's algorithm in that (1) it runs in polynomial time in the worst case and (2) it handles the case in which a procedure directly or indirectly calls itself.

Livadas and Croll [11] use an approach, similar to ours, to handle recursion in the computation of summary edges for constructing system dependence graphs [6]: their approach detects the strongly-connected components in the call graph and then, uses an iteration approach to compute a fixed point over the procedures in the component. One way that our approach for handling recursive programs differs from theirs is that our algorithm computes summary

[2] `Space` and `learn` are not shown here because they do not have recursion.

program	*alias*	*size*			*time*	
		HC	*Recur*	*%HC*	*HC*	*Recur*
loader†	ST	237	237	100.0	21.9	22.6
	LH	196	196	100.0	7.3	7.3
	AND	196	196	100.0	7.3	7.3
	LR	197	197	100.0	7.7	7.7
ansi-tape†	ST	290	290	100.0	18.5	18.7
	LH	284	284	100.0	11.9	12.0
	AND	277	277	100.0	6.5	6.5
	LR	300	300	100.0	6.2	6.2
dixie†	ST	709	633	89.3	44.0	90.4
	LH	708	632	89.3	15.0	31.4
	AND	708	632	89.3	9.4	16.5
	LR	704	628	89.2	8.8	15.8
unzip†	ST	808	807	99.8	42.5	43.3
	LH	807	806	99.8	17.0	17.0
	AND	807	805	99.8	12.9	13.0
	LR	805	803	99.8	10.6	10.6
lharc†	ST	786	562	71.6	10.7	13.4
	LH	786	489	62.3	8.4	9.5
	AND	784	488	62.3	7.7	9.1
	LR	796	587	73.8	6.5	10.9
flex‡	ST	2026	1871	92.3	760.3	556.6
	LH	2023	1865	92.2	566.5	407.0
	AND	2021	1863	92.2	512.3	359.5
	LR	2006	1864	92.9	582.8	355.9
bison‡	ST	2394	2362	98.7	122.6	108.8
	LH	2394	2362	98.7	108.1	97.8
	AND	2338	2306	98.6	61.2	54.9
larn‡	ST	6626	4484	67.7	3477.3	3205.1
	LH	6602	4427	67.1	1075.6	801.5
	AND	6592	4383	66.5	902.4	554.8
mpeg-play‡	ST	5708	5708	100.0	289.9	290.5
	LH	3935	3935	100.0	131.5	133.7
	AND	3935	3935	100.0	136.6	138.6

†Data are collected from all slices of the program.
‡Data are collected from one slice.

Table 3. Average size of a slice and average time in seconds to compute a slice.

information on-demand. Another difference is that our approach detects the mutual dependence among the computations of entries in *cache*. Unlike strongly-connected components in a call graph, entries in *cache* that are involved in mutual dependence can change, and additional mutual dependence can be detected, even during the iteration phase. This can lead to situations in which the slicer must suspend one iteration and begin another (one invocation of `ResolvePending` can be nested in another). Livadas and Croll's approach cannot handle such a situation.

7 Conclusions

We presented an approach that, when applied to the reuse-driven interprocedural slicing algorithm, can achieve more reuse in the presence of pointers. We also presented an approach that can compute more precise slices for programs that contain recursive functions. Our empirical studies show that our first approach can effectively achieve more reuse in computing slices for programs that use pointer variables, and significantly reduce the cost of the computation of slices. Our empirical studies also show that, for many programs, our approach of handling recursion can significantly improve the precision over the overestimate approach used by the HC algorithm.

Our future work includes performing more empirical studies, especially on larger subject programs, to further investigate the effectiveness of our approaches. We are also investigating how to generalize the first approach to apply to other slicing algorithms.

References

[1] L. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.

[2] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *The 6th ACM Symposium on Foundations of Software Engineering*, pages 46–55, Nov. 1998.

[3] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, September 1991.

[4] P. L. R. Group. PROLANGS Analysis Framework. http://www.prolangs.rutgers.edu/, Rutgers University, 1998.

[5] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *The 20th International Conference on Software Engineering*, pages 74–83, Apr. 1998.

[6] 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.

[7] J. C. Hwang, M. W. Du, and C. C. R. Finding program slices for recursive procedures. In *Proceedings of 12th Annual International Computer Software and Application Conference*, pages 220–227, 1988.

[8] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of 1992 ACM Symposium on Programming Language Design and Implementation*, pages 235–248, June 1992.

[9] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Joint 7th European Software Engineering Conference and 7th ACM Symposium on Foundations of Software Engineering*, Sept. 1999.

[10] D. Liang and M. J. Harrold. Equivalence analysis: A general technique to improve the efficiency of data-flow analyses in the presence of pointers. In *Program Analysis for Software Tools and Engineering '99*, Sept. 1999.

[11] P. E. Livadas and S. Croll. System dependence graph construction for recursive programs. In *The 17th Annual International Computer Software and Application Conference*, pages 414–420, Nov. 1993.

[12] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis 4th International Symposium, SAS '97, Lecture Notes in Computer Science Vol 1302*, pages 16–34, Sept. 1997.

[13] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[14] M. Weiser. Program slicing. *IEEE Trans. on Softw. Eng.*, 10(4):352–357, July 1984.