Towards Efficient and Accurate Program Analysis Using Light-Weight Context Recovery

Donglin Liang and Mary Jean Harrold
The Ohio State University
395 Dreese Lab, 2015 Neil Avenue
Columbus, OH 43210, USA
{dliang,harrold}@cis.ohio-state.edu

ABSTRACT

To compute accurate information efficiently for programs that use pointer variables, a program analysis must account for the fact that a procedure may access different sets of memory locations when the procedure is invoked under different callsites. This paper presents *light-weight context recovery*, a technique that can efficiently determine whether a memory location is accessed by a procedure under a specific callsite. The paper also presents a technique that uses this information to improve the precision and efficiency of program analyses. Our empirical studies show that (1) light-weight context recovery can be quite precise in identifying the memory locations accessed by a procedure under a specific callsite and (2) distinguishing memory locations accessed by a procedure under different callsites can significantly improve the precision and the efficiency of program analyses on programs that use pointer variables.

Keywords: Program analysis, slicing, aliasing.

1 INTRODUCTION

Software development, testing, and maintenance activities are important but expensive. Thus, researchers have investigated ways to provide software tools to improve the efficiency, and thus reduce the cost, of these activities. Many of these tools require program analyses to extract information about the program. For example, tools for debugging, program understanding, and impact analysis use program slicing (e.g., [4, 5, 14]) to focus attention on those parts of the software that can influence a particular statement. To support software engineering tools effectively, a program analysis must be sufficiently efficient so that the tools will have a reasonable response time or an acceptable throughput. Moreover, the program analysis must be sufficiently precise so that useful information will not be hidden within spurious information.

```
1.  int x;
2.  f(int* p) {
3.    *p = *p+x;
4.  }
5.  f1(int* q) {
6.    int z = 0;
7.    f(&z);
8.    f(q);
9.  }
10. int y;
11. main() {
12.   int w = 1;
13.   x = 1;
14.   y = 1;
15.   f1(&y);
16.   f(&w);
17.   printf("%d",w);
18. }
```

Figure 1: Example Program.

Many program analyses can effectively compute program information for programs that do not use pointer variables. However, when applying these techniques to programs that use pointer variables, several issues must be considered. First, in programs that use pointer variables, two different names may reference the same memory location at a program point. For example, in the program in Figure 1, both pointer dereference `*p` and variable name `y` can reference the memory location for `y` at statement 3. This phenomenon, called *aliasing*, must be considered when computing safe program information. For example, without considering the effects of aliasing, a program analysis would ignore the fact that `y` is referenced by `*p` in statement 3 and thus, conclude incorrectly that procedure `f()` does not modify `y`. Second, in programs that use pointer variables, a procedure can access different memory locations through pointer dereferences when the procedure is invoked at different callsites. For example, `f()` accesses `y` and `x` when it is called by statement 8, but it accesses `w` and `x` when it is called by statement 16. A program analysis that cannot distinguish the memory locations accessed by the procedure under the context of a specific callsite might compute spurious program information for this callsite. For example, a program analysis might report that `y`, `z`, and `w` are modified by `f()` when `f()` is called by statement 8. Furthermore, a program analysis that propagates the spurious program information throughout the program will be unnecessarily inefficient.

Many existing techniques (e.g., [2, 7, 11, 13]) compute safe program information by accounting for the effects of aliasing in the analysis. However, only a few of these techniques (e.g., [7, 11]) can distinguish memory locations accessed by a procedure under the context of specific callsites. These techniques use conditional analysis that attaches conditions to the information generated during program analysis. For example, `*p` references `w` at statement 3 (Figure 1) if `*p` is aliased to `w` at the entry to `f()`. Thus, a program analysis reports that `w` is modified by statement 3 under this condition. To determine whether `w` is modified by `f()` when statement 8 is executed, the program analysis determines whether `*p` is aliased to `w` at the entry to `f()` by checking the alias information at statement 8. Because `*q` is not aliased to `w` at statement 8, `*p` is not aliased to `w` at the entry to `f()` when `f()` is invoked at statement 8. Thus, the program analysis reports that `w` is not modified when statement 8 is executed.

Although a technique using conditional analysis can distinguish memory locations accessed by a procedure under specific callsites, and thus avoid computing spurious program information, it can be inefficient. First, it requires conditional alias information, which currently can be provided only by expensive alias-analysis algorithms (e.g., [6]). Second, it can increase the cost of computing the program information. For example, without using conditional analysis, the complexity of computing interprocedural reaching definitions is $O(n^2 v)$ where n is the size of the program and v is the number of names that references memory locations whereas using conditional analysis, the complexity is $O(n^2 v^5)$ [11].

To compute accurate program information without using expensive conditional alias information or adding complexity to a program analysis, we propose a new approach that has two steps. First, it efficiently identifies, by examining the ways in which memory locations are accessed in a procedure, the set of memory locations that can be accessed by the procedure under a specific callsite. Second, it uses this set of memory locations to reduce the spurious information propagated from the procedure to the callsite or from the callsite to the procedure.

To efficiently identify the memory locations that can be accessed by a procedure under a specific callsite, we developed a technique, *light-weight context recovery*. This technique is based on the observation that, under the context of a callsite, a formal parameter for a procedure typically points to the same set of memory locations, throughout the procedure, as the actual parameter to which it is bound if the formal parameter is a pointer. If a memory location is accessed exclusively through the pointer dereferences of this formal parameter, then such a memory location can be accessed by the procedure under a specific callsite only if the memory location can be accessed at the callsite through the pointer dereferences of the actual parameter to which it is bound. This allows the technique to identify the memory locations that are accessed by a procedure under a specific callsite.

To reduce the spurious information propagated across procedure boundaries by program analyses, we also developed a technique that uses the information about memory locations computed by light-weight context recovery. At each callsite, program information about a memory location that is not identified as being accessed by the called procedure need not be propagated from the callsite to the called procedure or from the called procedure to the callsite. Thus, this technique can improve the precision and the efficiency of program analyses.

This paper presents our light-weight context recovery algorithm (Section 2). The paper also illustrates, using interprocedural slicing [14], the technique that uses information provided by the light-weight context recovery algorithm to improve the program analyses (Section 3). The main benefit of our approach is that it is efficient: it can use alias information provided by efficient alias analysis algorithms, such as Liang and Harrold's [8] and Andersen's [1]; the light-weight context recovery is almost as efficient as modification side-effects analysis; using information provided by light-weight context recovery in a program analysis adds little cost to the program analysis. A second benefit of our approach is that, in many cases, it can identify a large number of memory locations whose information need not be propagated to specific callsites. Thus, it can provide significant improvement in both the precision and the efficiency of the program analysis. A third benefit of our approach is that it is orthogonal to many other techniques that improve the efficiency of program analyses. Thus, our approach can be used with those techniques to improve further the efficiency of program analyses.

This paper also presents a set of empirical studies in which we investigate the effectiveness of using light-weight context recovery to improve the efficiency and the precision of program analyses (Section 4). These studies show a number of interesting results:

– For many programs that we studied, the light-weight context recovery algorithm computes, with relatively little increase in cost, a significantly smaller number of memory locations as being modified at a callsite than that computed by the traditional modification side-effect analysis algorithm.

– For several programs, using alias information provided by Liang and Harrold's algorithm or Andersen's algorithm, the light-weight context recovery algorithm reports almost the same modification side-effects at a callsite as Landi, Ryder, and Zhang's algorithm [7], which must use conditional alias information.

– Using information provided by the light-weight context recovery can reduce the sizes of slices computed using the reuse-driven slicing algorithm [4] and the time required to compute such slices.

2 LIGHT-WEIGHT CONTEXT RECOVERY

This section first gives some definitions and then presents the light-weight context recovery algorithm.

Definitions

Memory locations in a program are referenced through object names; an *object name* consists of a variable and a possibly empty sequence of dereferences and field accesses. If an object name contains no dereferences, then the object name is a *direct* object name. Otherwise, the object name is an *indirect* object name. For example, `x.f` is a direct object name, whereas `*p` is an indirect object name. A direct object name *obj* represents a memory location, to which we refer as $\mathcal{L}(obj)$.

We say object name N_1 is an *extension* of object name N_2 if N_1 is constructed by applying a possibly empty sequence of dereferences and field accesses ω to N_2; in this case, we denote N_1 as $\mathcal{E}_\omega\langle N_2\rangle$. We refer to N_2 as a *prefix* of N_1. If ω is not empty, then N_1 is a *proper extension* of N_2, and N_2 is a *proper prefix* of N_1. If N is a formal parameter and a is the actual parameter that is bound to N at callsite c, we define a function $\mathcal{A}_c(\mathcal{E}_\omega\langle N\rangle)$ that returns object name $\mathcal{E}_\omega\langle a\rangle$.

For example, suppose that `r` is a pointer that points to a struct with field f. Then $\mathcal{E}_*\langle \mathtt{r}\rangle$ is $*\mathtt{r}$, $\mathcal{E}_{*.f}\langle \mathtt{r}\rangle$ is $(*\mathtt{r}).f$, and `r` is a proper prefix of `*r`. If `r` is a formal parameter to function F and `*q` is the actual parameter bound to `r` at a callsite c to F, then $\mathcal{A}_c(\mathcal{E}_{*.f}\langle \mathtt{r}\rangle)$ is $(**\mathtt{q}).f$.

Given an object name *obj* and a statement s, an alias analysis can determine a set of memory locations that can be aliased to *obj* at s. We refer to this set of memory locations as *obj*'s *accessed set* at s, and denote this set as $ASet(obj, s)$. For example, in Figure 1, $ASet(*\mathtt{p}, 3)$ is $\{\mathtt{y},\mathtt{z},\mathtt{w}\}$ when Landi and Ryder's alias analysis algorithm is used. Given a memory location *loc* and a procedure P, the *name set* of *loc* in P contains the object names that are used to reference *loc* in P. For example, the name set for `y` in `f()` (Figure 1) is $\{*\mathtt{p}\}$. A memory location *loc* is *dereference-used* in object name *obj* at statement s if *loc* is aliased to a proper prefix of *obj* at s. For example, suppose that `q` points to `r` and `r` points to `x` at statement s. Then `r` is dereference-used in `**q` at s because `r` is aliased to `*q` at s.

Light-Weight Context Recovery

To identify memory locations that can be accessed by a procedure P under a specific callsite, light-weight context recovery considers the nonlocal memory locations in P. If the name set of a nonlocal memory location *loc* contains a direct object name, then this technique reports that *loc* is accessed under any callsite to P. If the name set of *loc* contains a single indirect object name, then this technique computes additional information to determine whether *loc* can be accessed under a specific callsite to P. In other cases, for efficiency, this technique assumes *loc* is accessed under any callsite to P.

Suppose that indirect object name *obj* is the only object name in the name set of nonlocal memory location *loc* in procedure P. Then, if *loc* is referenced in P, it must be referenced through *obj*. If none of the memory locations dereference-used in *obj* is modified by statements in P, then when P is executed, *obj* references the same memory location at any point in P. In this case, if *obj* is an extension of a formal parameter, then when P is called at callsite c, *obj* must reference the same memory location as the one referenced by $\mathcal{A}_c(obj)$ at c. Thus, if *loc* is referenced in P under c, *loc* must be referenced by $\mathcal{A}_c(obj)$ at c. During program analysis, we must propagate the information for *loc* from P to c only if $\mathcal{A}_c(obj)$ is aliased to *loc* at c. This property of memory locations gives us an opportunity to avoid propagating (i.e., *filter*) some spurious information when the information is propagated from the procedure to a callsite during program analyses. We say that, if *loc* has this property in P, then *loc* is eligible to be filtered under callsites to P, and we say that *loc* is a *candidate*. More precisely, *loc* is a candidate in P if the following conditions hold:

Condition 1: *loc*'s name set in P contains a single indirect name *obj*.
Condition 2: *obj* is a proper extension of a formal parameter to P.
Condition 3: the memory locations dereference-used in *obj* are not modified in P.

For example, in Figure 1, the name set of `w` in `f()` contains only `*p`, a proper extension of formal parameter `p`. The value of `p` does not change in `f()`. Thus, `w` is a candidate in `f()`. Because $\mathcal{A}_{16}(*\mathtt{p})$ is `w`, we need to propagate the information for `w` from `f()` to statement 16. However, because $\mathcal{A}_7(*\mathtt{p})$ is `z`, we need not propagate the information for `w` from `f()` to statement 7.

Light-weight context recovery processes the procedures in a bottom-up order (reverse topological order) on the call graph to identify the memory locations that are candidates in each procedure. Figure 2 shows algorithm `ContextRecovery` that performs the processing. For a nonlocal memory location *loc* referenced in procedure P, `ContextRecovery` computes a mark, $Mark_P[loc]$, whose value can be unmarked (`U`), eligible ($\surd$), or ineligible($\times$). By default, $Mark_P[loc]$ is initialized to `U`. If *loc* is a candidate in P, then $Mark_P[loc]$ is $\surd$. In this case, if *obj* is the object name in *loc*'s name set in P, then `ContextRecovery` stores *obj* in $OBJ_P[loc]$. If *loc* is not a candidate in P, then $Mark_P[loc]$ is $\times$ and $OBJ_P[loc]$ is ϕ. `ContextRecovery` also sets $MOD_P[loc]$ to true if *loc* is modified by a statement in P so that it can check whether the memory locations dereference-used in an object name have been modified in P.

`ContextRecovery` examines the way *loc* is accessed in P, and calls `Update()` at various points (e.g., lines 5 and

```
algorithm   ContextRecovery(𝒫)
input       𝒫: a program
global      Mark_P: maps memory locations in procedure P to marks
            MOD_P: indicate whether a memory location is modified
            OBJ_P: maps memory locations to object names
declare     W: list of procedures sorted in reverse topological order
begin ContextRecovery
1.   foreach procedure P do       /*Intraprocedural phase*/
2.     foreach statement s in P do
3.       foreach object name obj in s do
4.         if obj is direct then
5.           Update(P,ℒ(obj),F,φ)
6.         else
7.           foreach memory location loc in ASet(obj, s) do
8.             Update(P,loc,T,obj)
9.           endfor
10.        endif
11.      endfor
12.      update MOD_P for memory locations modified at s
13.    endfor
14.    add P to W
15.  endfor
16.  while W≠φ do              /*Interprocedural phase*/
17.    take the first procedure P from W
18.    foreach callsite c to R in P do
19.      foreach nonlocal memory loc referenced in R do
20.        if Mark_R[loc] ≠ √ then
21.          Update(P,loc,F,φ)
22.        elseif loc∈ASet(𝒜_c(OBJ_R[loc]), c) then
23.          if 𝒜_c(OBJ_R[loc]) is indirect then
24.            Update(P,loc,T,𝒜_c(OBJ_R[loc]))
25.          else
26.            Update(P,loc,F,φ)
27.          endif
28.        endif
29.        update MOD_P[loc]
30.      endfor
31.    endfor
32.    validate object names in OBJ_P with MOD_P
33.    if Mark_P or MOD_P updated then add P's callers to W
34.  endwhile
end ContextRecovery
procedure   Update(Q,l,m,o)
input       Q is a procedure, l is a memory location,
              m is a boolean, and o is an object name or φ
global      Mark_Q: marks for memory locations in Q
            OBJ_Q: array of object names
begin Update
35.  if m =F then
36.    Mark_Q[l] := ×; OBJ_Q[l] := φ
37.  elseif m =T and Mark_Q[l] = U then
38.    if o is an extension of a formal then
39.      Mark_Q[l] := √; OBJ_Q[l] := o
40.    else
41.      Mark_Q[l] := ×; OBJ_Q[l] := φ
42.    endif
43.  elseif m =T and Mark_Q[l] = √ then
44.    if o≠OBJ_Q[l] then
45.      Mark_Q[l] := ×; OBJ_Q[l] := φ
46.    endif
47.  endif
end Update
```

Figure 2: Algorithm identifies candidate memory locations.

8) to compute $Mark_P[loc]$ and $OBJ_P[loc]$. `Update()` inputs procedure Q, a memory location l, a boolean flag m, and an object name o. When `ContextRecovery` detects that a memory location is accessed at a statement in P, if the memory location is accessed through an indirect object name, the algorithm calls `Update()` with m as T (true). Otherwise, if the memory location is not accessed through an indirect object name, the algorithm calls `Update()` with m as F (false).

`Update()` sets the values for $Mark_Q[l]$ and $OBJ_Q[l]$ according to m and the current value of $Mark_Q[l]$. If m is F, then l is not a candidate in Q because condition 1 is violated. Thus, $Mark_Q[l]$ is updated with $\times$ and $OBJ_Q[l]$ is updated with ϕ (line 36). If m is T and the current value of $Mark_Q[l]$ is `U`, then `Update()` checks o to see whether o is an extension of a formal parameter to Q (lines 37-38). If so, then l is a candidate according to the information available thus far. Thus, $Mark_Q[l]$ is updated with $\surd$ and $OBJ_Q[l]$ is updated with o (line 39). Otherwise, l is not a candidate because condition 2 is violated. Thus, $Mark_Q[l]$ is updated with $\times$ and $OBJ_Q[l]$ is updated with ϕ (line 41). If m is T and $Mark_Q[l]$ is $\surd$, then `Update()` checks whether o is the same as $OBJ_Q[l]$ (lines 43-44). If they are not the same, then l can be accessed in Q through more than one object name. This violates condition 1. Thus, l is not a candidate and $Mark_Q[l]$ is updated with $\times$ and $OBJ_Q[l]$ is updated with ϕ (line 45). For the other cases, $Mark_Q[l]$ and $OBJ_Q[l]$ remain unchanged.

`ContextRecovery` computes $Mark$, OBJ, and MOD using an intraprocedural phase and an interprocedural phase. In the intraprocedural phase, `ContextRecovery` processes the object names appearing at each statement s in a procedure P (lines 2-13). If an object name *obj* is direct, `ContextRecovery` calls `Update`(P,$\mathcal{L}(obj)$,F,ϕ) (lines 4-5). If *obj* is indirect, then for each memory location *loc* in $ASet(obj, s)$, `ContextRecovery` calls `Update`(P,loc,T,obj) (lines 7-9). For each memory location l that is modified at s, `ContextRecovery` also set $MOD_P[l]$ to be true (line 12).

For example, when `ContextRecovery` processes statement 3 in `f()` in Figure 1, it checks `*p`. `*p` is an indirect name and $ASet(*\texttt{p}, 3) = \{\texttt{y}, \texttt{z}, \texttt{w}\}$. Thus, the algorithm calls `Update(f,y,T,*p)`, `Update(f,z,T,*p)`, and `Update(f,w,T,*p)`. `ContextRecovery` also checks `x` at statement 3. Because `x` is a direct name, `ContextRecovery` calls `Update(f, x,F,`ϕ`)`. After `f()` is processed, $Mark_f$ and OBJ_f have the following values:

$$\begin{array}{ll} Mark_f[\texttt{x}] = \times, & OBJ_f[\texttt{x}] = \phi \\ Mark_f[\texttt{y}] = \surd, & OBJ_f[\texttt{y}] = *\texttt{p} \\ Mark_f[\texttt{z}] = \surd, & OBJ_f[\texttt{z}] = *\texttt{p} \\ Mark_f[\texttt{w}] = \surd, & OBJ_f[\texttt{w}] = *\texttt{p} \end{array}$$

In the interprocedural phase, `ContextRecovery` processes each callsite c to procedure R in a procedure P (lines 16-34) using the worklist W. For each nonlocal memory location *loc* referenced in R, `ContextRecovery` checks $Mark_R[loc]$ (line 20). If $Mark_R[loc]$ is not $\surd$, then `ContextRecovery` assumes that *loc* is accessed by R under any callsite to R, including c. Thus, `ContextRecovery` calls `Update`(P,loc,F,ϕ) to indicate that *loc* is accessed by R under c in some unknown way (line 21). Otherwise, if $Mark_R[loc]$ is $\surd$, then `ContextRecovery` checks whether *loc* is in $ASet(\mathcal{A}_c(OBJ_R[loc]), c)$ (line 22). If *loc* is in $ASet(\mathcal{A}_c(OBJ_R[loc]), c)$, then *loc* is referenced by R under c through object name $\mathcal{A}_c(OBJ_R[loc])$. `Context-`

`Recovery` calls `Update`(P,loc,T,$\mathcal{A}_c(OBJ_R[loc])$) if $\mathcal{A}_c(OBJ_R[loc])$ is indirect, and calls `Update`(P,loc,F,ϕ) if $\mathcal{A}_c(OBJ_R[loc])$ is direct (lines 23–27). If loc is not in $ASet(\mathcal{A}_c(OBJ_R[loc]), c)$, then loc is not referenced by R under c. `ContextRecovery` does nothing in this case. In the interprocedural phase, when a memory location loc is processed, if $MOD_R[loc]$ is true, then $MOD_P[loc]$ is also set to true (line 29).

For example, when `ContextRecovery` processes the callsite to `f()` at statement 8 (Figure 1), it first checks `x`. $Mark_f[\mathtt{x}]$ is $\times$. Thus, the algorithm calls `Update(f1,x,F,`ϕ`)`. The algorithm then checks `y`. $Mark_f[\mathtt{y}]$ is $\surd$. The algorithm checks whether `y` is in $ASet(\mathcal{A}_8(*\mathtt{p}), 8)$. $\mathcal{A}_8(*\mathtt{p}) = *\mathtt{q}$, and $ASet(*\mathtt{q}, 8) = \{\mathtt{y}\}$. Thus, the algorithm calls `Update(f1,y,T,*q)`. The algorithm finally checks `z` and `w` and does nothing because `z` and `w` are not in $ASet(*\mathtt{q}, 8)$. The values for $Mark_{f1}$ and OBJ_{f1} change from

$$Mark_{f1}[\mathtt{x}] = \mathtt{U}, \quad OBJ_{f1}[\mathtt{x}] = ?$$
$$Mark_{f1}[\mathtt{y}] = \mathtt{U}, \quad OBJ_{f1}[\mathtt{y}] = ?$$

to

$$Mark_{f1}[\mathtt{x}] = \times, \quad OBJ_{f1}[\mathtt{x}] = \phi$$
$$Mark_{f1}[\mathtt{y}] = \surd, \quad OBJ_{f1}[\mathtt{y}] = *\mathtt{q}$$

after statement 8 is processed.

In the interprocedural phase, `ContextRecovery` also validates the indirect object names appearing in OBJ_P to make sure that the memory locations dereference-used in such indirect names are not modified in P (line 32). Suppose that indirect name *obj* is assigned to $OBJ_P[loc]$. If there is a memory location l that is dereference-used in *obj* at a statement in P such that $MOD_P[l]$ is true, then loc is ineligible because condition 3 is violated. Thus, `ContextRecovery` updates $Mark_P[loc]$ with $\times$ and $OBJ_P[loc]$ with ϕ.

After `ContextRecovery` processes P in the interprocedural phase, if MOD_P or $Mark_P$ is updated, then the algorithm puts P's callers in W (line 33). `ContextRecovery` continues until W becomes empty. Table 1 shows the results computed by `ContextRecovery` for the example program (Figure 1).

$Mark$	OBJ	MOD
$Mark_f[\mathtt{x}] = \times$ $Mark_f[\mathtt{y}] = \surd$ $Mark_f[\mathtt{z}] = \surd$ $Mark_f[\mathtt{w}] = \surd$	$OBJ_f[\mathtt{x}] = \phi$ $OBJ_f[\mathtt{y}] = *\mathtt{p}$ $OBJ_f[\mathtt{z}] = *\mathtt{p}$ $OBJ_f[\mathtt{w}] = *\mathtt{p}$	$MOD_f[\mathtt{y}] = true$ $MOD_f[\mathtt{z}] = true$ $MOD_f[\mathtt{w}] = true$
$Mark_{f1}[\mathtt{x}] = \times$ $Mark_{f1}[\mathtt{y}] = \surd$	$OBJ_{f1}[\mathtt{x}] = \phi$ $OBJ_{f1}[\mathtt{y}] = *\mathtt{q}$	$MOD_{f1}[\mathtt{y}] = true$
$Mark_{main}[\mathtt{x}] = \times$ $Mark_{main}[\mathtt{y}] = \times$	$OBJ_{main}[\mathtt{x}] = \phi$ $OBJ_{main}[\mathtt{y}] = \phi$	$MOD_{main}[\mathtt{x}] = true$ $MOD_{main}[\mathtt{y}] = true$

Table 1: $Mark$, OBJ, and MOD for example program.

Given that n is the size of the program, the complexity of `ContextRecovery` is $O(n^2)$ in the absence of recursion and $O(n^3)$ in the presence of recursion. The complexity of `ContextRecovery` is the same as the algorithm for computing modification side effects for the procedures.

3 USING LIGHT-WEIGHT CONTEXT RECOVERY TO IMPROVE SLICING

This section shows how the information computed using `ContextRecovery` can improve interprocedural slicing. Other program analyses, such as computing interprocedural reaching definitions and constructing system-dependence graphs, can be improved in a similar way.

Interprocedural Slicing

Program slicing is a technique to identify statements in a program that can affect the value of a variable v at a statement s ($\langle s, v\rangle$ is called the *slicing criterion*) [14]. Program slicing can be used to support tasks such as debugging, regression testing, and reverse engineering.

One approach of program slicing first computes data and control dependences among the statements and builds a system-dependence graph, and then computes the slice by solving a graph-reachability problem on this graph [5]. Other approaches, such as the one used in the reuse-driven interprocedural slicing algorithm [4], use precomputed control-dependence information, but compute the data-dependence information on demand using control-flow graphs[1] (CFGs) for the procedures. We use the reuse-driven slicing algorithm as an example to show how a program analysis can be improved using information provided by light-weight context recovery.

The reuse-driven slicer computes an interprocedural slice for criterion $\langle s, v\rangle$ by invoking a partial slicer on the procedures of the program. The reuse-driven slicer first invokes the partial slicer on P_s, the procedure that contains s, to identify a subset of statements in P_s or in procedures called by P_s and a subset of inputs to P_s that may affect v at s. We refer to s and v as the *partial slicing standard* used by the partial slicer and denote it as $[s, v]$. We refer to the subset of statements identified by the partial slicer as a *partial slice* with respect to $[s, v]$. We also refer to the subset of inputs identified by the partial slicer as *relevant inputs* with respect to $[s, v]$.

When P_s is not the main function of the program, the statements in procedures that call P_s might also affect the slicing criterion $\langle s, v\rangle$ through the relevant inputs of $[s, v]$. Therefore, after P_s is processed, for each callsite c_i that calls P_s, the reuse-driven slicer binds each relevant input f back to c_i and creates a new partial slicing standard $[c_i, a_i]$, given that a_i is bound to f at c_i. The reuse-driven slicer then invokes the partial slicer on $[c_i, a_i]$ to identify the statements in P_{c_i} that should be included in the slice. The algorithm continues until no additional partial slicing standards can be generated. The algorithm returns the union of all partial slices computed by the partial slicer as the program slice for $\langle s, v\rangle$.

Figure 3 shows the call multigraph[2] for the program

[1]Nodes represent statements; edges represent control transfer.
[2]Nodes represent procedures; edges represent callsites.

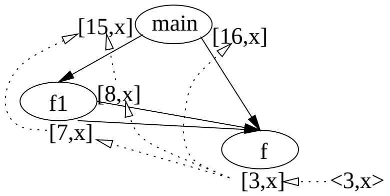

Figure 3: Call multigraph annotated with partial slicing standards for $\langle 3, \mathtt{x}\rangle$; solid lines show multigraph edges; dotted lines show relationships among partial slicing standards.

in Figure 1. The multigraph is annotated with partial slicing standards created by the reuse-driven slicer to compute the slice for $\langle 3, \mathtt{x}\rangle$. The reuse-driven slicer first invokes the partial slicer on `f()` with respect to $[3, \mathtt{x}]$. The partial slicer computes partial slice $\{3\}$ and relevant input set $\{\mathtt{x}\}$. After `f()` is processed, the reuse-driven slicer creates new partial slicing standards $[7, \mathtt{x}]$ and $[8, \mathtt{x}]$ for the callsites to `f()` in `f1()` and partial slicing standard $[16, \mathtt{x}]$ for the callsite to `f()` in `main()`, and invokes the partial slicer on these standards. The partial slicer computes relevant input set $\{\mathtt{x}\}$ for $[7, \mathtt{x}]$ and $[8, \mathtt{x}]$. After the partial slicer finishes processing $[7, \mathtt{x}]$ or $[8, \mathtt{x}]$, the reuse-driven slicer further creates partial slicing standard $[15, \mathtt{x}]$ for the callsite to `f1()` in `main()`, and invokes the partial slicer on this standard. The resulting slice for $\langle 3, \mathtt{x}\rangle$ is $\{3, 7, 8, 13, 15, 16\}$, the union of all partial slices computed during the processing.

The partial slicer computes the partial slice for standard $[s, v]$ by propagating memory locations backward throughout P_s using P_s's CFG. For each node N in the CFG of P_s, the partial slicer computes two sets of memory locations: $IN[N]$ at the entry of N; $OUT[N]$ at the exit of N. $IN[N]$ is computed using $OUT[N]$ and information about N. $OUT[N]$ is computed as the union of the $IN[]$ sets of N's CFG successors. The partial slicer iteratively computes $IN[]$ and $OUT[]$ for each node in P_s until a fixed point is reached. The formal parameters and nonlocal memory locations in $IN[]$ at P_s's entry are the relevant inputs with respect to $[s, v]$.

When N is not a callsite, the partial slicer can compute $IN[N]$ by considering $OUT[N]$ and those memory locations whose values are modified or used at N. If memory locations in $OUT[N]$ can be modified at N, then N and statements on which N is control dependent are added to the slice. When N is a callsite to procedure Q, the partial slicer must process Q to compute $IN[N]$ and to identify the statements in Q for inclusion in the partial slice. Figure 4 shows `ProcessCall()`, the procedure that processes a callsite c to Q.

`ProcessCall()` uses a cache to store the partial slice and the relevant inputs for each partial slicing standard created by the reuse-driven slicer. For each memory location u in $OUT[c]$, `ProcessCall()` binds u to u' in Q (line 2). If u' is not modified by Q or by procedures called by Q, `ProcessCall()` simply adds u to $IN[c]$ (lines 3-4). Otherwise, `ProcessCall()` creates a partial slicing standard $[Q.exit, u']$, in which $Q.exit$ is the exit of Q. `ProcessCall()` then checks the cache against $[Q.exit, u']$ (line 6). If the cache does not contain information for $[Q.exit, u']$, then `ProcessCall()` invokes `ComputePSlice()` on $[Q.exit, u']$, and stores the partial slice and the relevant inputs returned by `ComputePSlice()` in the cache (line 7). `ProcessCall()` then merges the partial slice with the program slice (line 9), and calls `BackBind()` (not show) to bind the relevant inputs back to c and adds them to the $IN[c]$ (line 10). After c has been processed, if some statements in Q are included in the slice, then c and statements on which c is control dependent are added to the slice.

```
procedure   ProcessCall(c, IN, OUT)
input       c: a call node that calls Q
            OUT: the set OUT[c]
output      IN: the set IN[c]
globals     cache[s, v]: pair of (pslice, relInputs)
                previously computed by ComputePSlice on [s, v]
begin ProcessCall
1.   foreach u in OUT do
2.     u' = Bind(u, c, Q)
3.     if u' is not modified by Q or pocedures called by Q then
4.       IN = IN∪{u}
5.     else
6.       if cache[Q.exit, u'] is NULL then
7.         cache[Q.exit, u'] = ComputePSlice(Q.exit, u')
8.       endif
9.       add cache[Q.exit, u'].pslice to the slice
10.      IN = IN∪BackBind(cache[Q.exit, u'].relInputs, c,Q)
11.    endif
12.  endfor
end ProcessCall
```

Figure 4: Procedure processes callsites using caching.

For example, to compute the slice for $\langle 17, \mathtt{w}\rangle$, the slicer first propagates `w` from statement 17 to $OUT[16]$. Because statement 16 is a callsite, the slicer propagates `w` into `f()` and creates a new partial slicing standard $[4, \mathtt{w}]$. The slicer then invokes the partial slicer on $[4, \mathtt{w}]$, and computes partial slice $\{3\}$ and relevant inputs $\{\mathtt{x,y,z,w,p}\}$. The slicer binds `x, y, z, w`, and `p` back to statement 16 and puts `x,y,z,` and `w` in $IN[16]$. The slicer keeps processing and adds statements 3, 6, 7, 8, 12, 13, 14, 15, 16, and 17 to the slice.

Interprocedural Slicing Using Information Provided by Light-Weight Context Recovery

The precision and the performance of the reuse-driven slicer can be improved if it can identify the set of memory locations that can be modified by a procedure under a specific callsite. To do this, before the slicer propagates a memory location from the callsite to the called procedure, it first checks whether the memory location can be modified by the procedure under this callsite. If the memory location cannot be modified by the procedure under this callsite, then the reuse-driven slicer does not propagate the memory location into the called procedure. Similarly, the precision and the performance of the reuse-driven slicer can be improved if it can identify the set of memory locations that can be referenced by a procedure under a specific callsite. The slicer propagates, from the called procedure to a callsite, only the memory locations that are referenced under the callsite. Both of these improvements can reduce the spurious information propagated across the procedure boundaries, and thus can improve the precision and efficiency of the reuse-driven slicer.

For example, consider the actions of the reuse-driven slicer for $\langle 17, \texttt{w}\rangle$ (Figure 1), if the two improvements, described above, are made. The improved slicer first propagates `w` from statement 17 to statement 16. Because `f()` modifies `w` when it is invoked by statement 16, the improved slicer propagates `w` from statement 16 into `f()`, and creates partial slicing standard $[4, \texttt{w}]$. The improved slicer then invokes the partial slicer on $[4, \texttt{w}]$, adds statement 3 to the partial slice, and identifies `x`, `y`, `z`, `w`, and `p` as the relevant inputs. The improved slicer checks statement 16 and finds that only `x` and `w` can be referenced when `f()` is invoked by statement 16. Thus, the improved slicer adds only `x` and `w` to $IN[16]$. The improved slicer further propagates `x` and `w` to $OUT[15]$. Because `f1()` modifies neither `x` nor `w` when it is invoked at statement 15, the improved slicer propagates `x` and `w` directly to $IN[15]$, without propagating them into `f1()`. The improved slicer continues propagating the memory locations throughout the program, and adds statements 3, 12, 13, 16, 17 to the slice. This example shows that using specific callsite information can help the reuse-driven slicer compute more precise slices.

We modify `ProcessCall()` (Figure 4) to use the set of memory locations that are modified by a procedure under a specific callsite to reduce the spurious information propagated from a callsite to the called procedure. Figure 5 shows the modified `ProcessCall()` (lines 2'–5' replace lines 2–5 in the original version). For each u in $OUT[c]$, the new `ProcessCall()` first checks whether u is modified by Q at c (line 2'). If u is not modified by Q at c, then the new `ProcessCall()` adds u to $IN[c]$ (line 3'). If u is modified by Q at c, then the new `ProcessCall()` binds u to u' in Q, creates partial slicing standard $[Q.exit, u']$, and continues the computation in the usual way (lines 5'–10). Because u being modified by Q at c implies that u' is modified by Q, the new `ProcessCall()` need not check u'.

We also modify `BackBind()` to use the set of memory locations that are referenced by a procedure under a specific callsite to reduce the spurious information propagated from a procedure to its callsites. Figure 5 shows the modified `BackBind()` in which line 16 has been changed. `BackBind()` checks each memory location l in its input $eVars$ (line 13). If l is a formal parameter, `BackBind()` adds the memory locations that are bound to l at c to $CalleeVars$ (lines 14-15). Otherwise, the new `BackBind()` checks whether l is referenced when P is invoked at c (line 16). If so, then `BackBind()` puts l in $CalleeVars$ (line 17). Finally, `BackBind()` returns $CalleeVars$ (line 20).

```
procedure   ProcessCall(c, IN, OUT)
input       c: a call node that calls Q
            OUT: the set OUT[c]
output      IN: the set IN[c]
globals     cache[s, v]: pair of (pslice, relInputs)
                previously computed by ComputePSlice for [s, v]
begin ProcessCall
1.    foreach u in OUT do
2'.     if u is not modified by Q at c then
3'.       IN = IN ∪ {u}
4'.     else
5'.       u' = Bind(u, c, Q)
6.        if( cache[Q.exit, u'] is NULL) then
7.          cache[Q.exit, u'] = ComputePSlice(Q.exit, u')
8.        endif
9.        add cache[Q.exit, u'].pslice to the slice
10.       IN = IN ∪ BackBind(cache[Q.exit, u'].relInputs, c,Q)
11.     endif
12.   endfor
end ProcessCall
function    BackBind(eVars, c, P)
input       eVars: memory locations reaching the entry of P
            P: a procedure
            c: a call node that calls P
output      memory locations at callsite
begin BackBind
13.   foreach memory location l in eVars do
14.     if l is formal parameter then
15.       add memory locations bound to l at c into CalleeVars
16.     elseif l is referenced by P at c then    /*old: 16. else */
17.       add l to into CalleeVars
18.     enif
19.   endfor
20.   return CalleeVars
end BackBind
```

Figure 5: `ProcessCall()` (modified) and `BackBind()`.

The sets of memory locations that are modified by a procedure P and that are referenced by P under a specific callsite c can be determined by using conditional analysis if conditional alias information for the program is available [7]. However, because conditional alias information is expensive to compute and to store, this might be infeasible for large programs.

Instead, we can use $Mark_P$, OBJ_P, and MOD_P computed by `ContextRecovery` to determine the memory locations that can be modified by P under callsite c. For a nonlocal memory location loc in P, if $MOD_P[loc]$ is true and $Mark_P[loc]$ is $\times$, then loc can be modified by P under any callsite to P, including c. If $MOD_P[loc]$ is true and $Mark_P[loc]$ is $\surd$, and if loc is in $ASet(\mathcal{A}_c(OBJ_P[loc]), c)$, then loc can be modified by P under c. Otherwise, loc is not modified by P under c.

For example, according to the result in Table 1, $Mark_f[\texttt{y}] = \surd$, $OBJ_f[\texttt{y}] = *\texttt{p}$, and $MOD_f[\texttt{y}] = true$. Thus, `f()` modifies `y` when `f()` is invoked at statement 8 in Figure 1 because $ASet(\mathcal{A}_8(*\texttt{p}), 8)$ (i.e. $ASet(*\texttt{q}, 8)$) contains `y`. However, `f()` does not modify `y` when `f()` is invoked at statement 16 because $ASet(\mathcal{A}_{16}(*\texttt{p}), 16)$ (i.e. $ASet(\texttt{w}, 16)$) does not contain `y`.

We can use a similar approach to determine the memory locations that can be referenced by P when P is invoked from c. For a memory location loc, if $Mark_P[loc]$ is $\times$, then loc can be referenced by P under c. If $Mark_P[loc]$ is $\surd$ and $OBJ_P[loc]$ is obj, and if loc is in $ASet(\mathcal{A}_c(obj), c)$, then loc can be referenced by P under c. Otherwise, loc is not referenced by P under c.

4 EMPIRICAL STUDIES

We performed several studies to evaluate the effectiveness of using light-weight context recovery to improve the precision and the efficiency of program analyses. We implemented `ContextRecovery` and the reuse-driven slicing algorithm that uses information provided by `ContextRecovery` using PROLANGS Analysis Framework (PAF) [3]. In the studies, we compared the results computed with alias information provided by Liang and Harrold's algorithm (LH) [8] and by Andersen's algorithm (AND) [1].[3] We gathered the data for the studies on a Sun Ultra30 workstation with 640MB physical memory and 1GB virtual memory.[4] The left side of Table 2 gives information about the subject programs we used.

program	*CFG Nodes*	*LOC*	*LH*		*AND*	
			CI	*CR*	*CI*	*CR*
loader	819	1132	0.07	0.11	0.08	0.11
dixie	1357	2100	0.12	0.19	0.11	0.17
learn	1596	1600	0.11	0.2	0.11	0.17
unzip	1892	4075	0.14	0.22	0.14	0.21
assembler	1993	2510	0.26	0.35	0.23	0.34
smail	2430	3212	0.28	0.52	0.29	0.59
lharc	2539	3235	0.19	0.35	0.22	0.35
simulator	2992	3558	0.47	0.59	0.49	0.57
arc	3955	7325	0.38	0.77	0.38	0.68
space	5601	11474	1.48	1.62	1.86	1.91
larn	11796	9966	2.18	2.85	2.11	2.84
espresso	15351	12864	7.34	8.81	8.62	15.25
moria	20316	25002	29.29	38.98	22.49	24.79
twmc	22167	23922	2.98	4.69	3.53	7.96

Table 2: Information about subject programs (left) and time in seconds for context-insensitive modification side effect analysis (CI) and for `ContextRecovery` (CR) (right).

Study 1

The goal of study 1 is to evaluate the efficiency of our algorithm (CR). We compared the time required to run CR on a program and the time required to compute modification side-effects of the procedures in the program with a context-insensitive algorithm (CI). We make this comparison because (1) the time for computing modification side-effects is relatively small compared to the time required for many program analyses and (2) our algorithm can be used instead of CI to compute more precise modification side-effects that are required for many program analyses. The right side of Table 2 shows the results computed using alias information provided by the LH algorithm and by the AND algorithm. From the table, we can see that, for the subjects we studied, our algorithm is almost as efficient as the algorithm that computes the modification side-effects of the procedures in the program. This suggests that the time added by our algorithm is negligible in many program analyses.

[3]See [8] for a detailed comparison of these two algorithms.

[4]Because we simulate the effects of library functions using new stubs with greater details, data reported in these studies for the subject programs differ from those reported in our previous work.

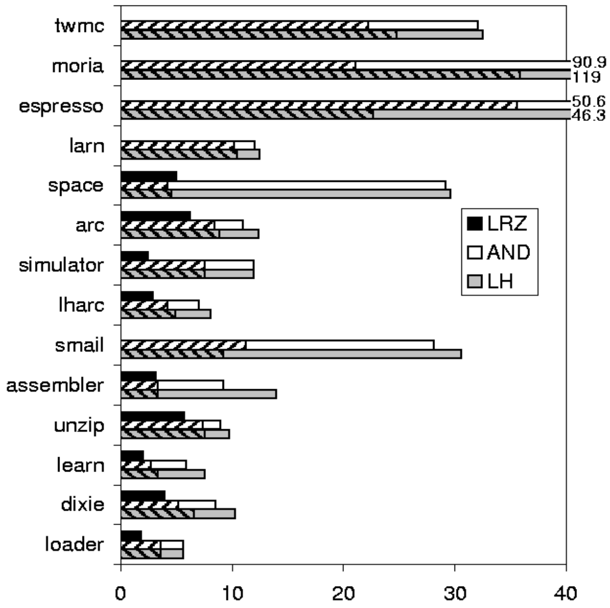

Figure 6: Average sizes of MOD at a callsite.

Study 2

The goal of study 2 is to evaluate the precision of our algorithm in the identification of memory locations that are modified by a procedure under a specific callsite (MOD at a callsite). We compared the size of MOD at a callsite computed by the traditional context-insensitive modification side-effect analysis algorithm (the CI-MOD algorithm) and by our algorithm. The reduction of the size of MOD at a callsite indicates the effectiveness of our technique in filtering spurious information at a callsite. We also compared the results computed by our algorithm with the results computed by Landi, Ryder, and Zhang's modification side effect analysis algorithm (the LRZ algorithm) [7] that uses conditional analysis. The results computed by the LRZ algorithm can be viewed as a lower bound for our algorithm. We used our implementations of the CI-MOD algorithm and of our algorithm, and used the implementation of the LRZ algorithm provided with PAF.

Figure 6 shows the results of this study. In the graph, the total length of each bar indicated by either AND or LH represents the average size of MOD at a callsite computed by the CI-MOD algorithm using the alias information provided by the AND algorithm or the LH algorithm. On each bar, the length of the slanted segment represents the average size of MOD at a callsite computed by our algorithm using alias information provided by the corresponding alias analysis algorithm. For example, using the alias information provided by the AND algorithm, the CI-MOD algorithm reports that a callsite modifies 29 memory locations in `space`. Using the same alias information, however, our algorithm reports that a callsite modifies only 4.2 memory locations. The graph shows that for most subject programs we studied, our algorithm can compute significantly more precise MOD at a callsite than the CI-MOD algorithm. Thus, we expect that, using information provided by our algorithm, we can significantly reduce the spurious information propagated across procedure boundaries.

Figure 6 also shows the average size of MOD at a callsite computed by the LRZ algorithm.[5] In the graph, the length of each bar indicated by LRZ represents the average size of MOD at a callsite computed by the LRZ algorithm. For example, the LRZ algorithm reports that a callsite in `space` can modify 5 memory locations. Note that because the LRZ algorithm uses alias information computed by Landi and Ryder's algorithm [6], which treats the modification of a field of a recursive structure as the modification of all the fields of such a structure in some cases, this algorithm reports a larger set of memory locations than our algorithm for `space`. The graph shows that, for several programs we studied, the size of MOD at a callsite computed by our algorithm is close to that computed by the LRZ algorithm. This result suggests that our algorithm can be quite precise in identifying memory locations that can be modified by a procedure under a specific callsite. The graph also shows that the precision of MOD at a callsite computed by our algorithm varies for different programs. This suggests that the effectiveness of improving program analyses using information provided by our algorithm might depend on how the program is written.

Study 3

The goal of study 3 is to evaluate the effectiveness of using information provided by light-weight context recovery in improving the precision and efficiency of the reuse-driven slicing algorithm. We compared the size of a slice and the time to compute a slice with or without using information provided by light-weight context recovery. Table 3 shows the results.

The left side of Table 3 shows S, the average size of a slice computed using information provided by light-weight context recovery, and S', the average size of a slice computed without using such information. The table also shows the ratio of S to S' in percentage. From the table, we can see that, for some programs, using information provided by light-weight context recovery can significantly improve the precision of computing interprocedural slices. However, for other programs, we do not see significant improvement. One explanation for this may be that, on these programs, the precision of the interprocedural slicing is not sensitive to the precision of identifying memory locations that are modified or referenced at a statement. This result is consistent with results reported in [8, 13], which show that the precision of interprocedural slicing is not very sensitive to the precision of alias information.

		average size			*time in seconds*		
name	*alias*	*S*	*S'*	*S/S'*	*T*	*T'*	*T/T'*
load-er†	LH	187	241	77.9%	2.3	6.4	35.0%
	AND	187	241	77.9%	2.3	6.4	35.2%
dixie†	LH	629	648	97.0%	10.0	23.0	43.6%
	AND	609	648	94.0%	5.3	12.3	42.9%
learn†	LH	499	501	99.6%	16.2	29.7	54.6%
	AND	479	501	95.6%	10.9	21.2	51.5%
unzip†	LH	791	806	98.1%	8.5	12.1	70.6%
	AND	791	806	98.1%	8.3	9.7	85.7%
assem-bler†	LH	744	751	99.1%	15.2	93.8	16.2%
	AND	630	750	84.0%	8.7	49.6	17.6%
smail‡	LH	1066	1087	98.1%	158	545	29.0%
	AND	1032	1090	94.7%	129	390	33.0%
lharc†	LH	620	710	87.4%	18.4	59.0	31.1%
	AND	546	698	78.3%	10.3	39.5	26.2%
simu-later†	LH	1174	1178	99.7%	11.1	18.5	59.8%
	AND	1174	1178	99.7%	10.8	18.7	58.0%
arc†	LH	788	804	98.1%	9.4	12.9	72.5%
	AND	771	803	96.1%	7.6	9.2	82.2%
space‡	LH	2028	2161	93.9%	31.8	577	5.5%
	AND	2019	2161	93.4%	31.2	574	5.4%
larn‡	LH	4590	4612	99.5%	642	977	65.7%
	AND	4576	4603	99.4%	619	798	77.5%

		average size			*time in hours*		
name	*alias*	*S*	*S'*	*S/S'*	*T*	*T'*	*T/T'*
espre-sso‡	LH	5704	5705	100%	1.2	4.7	25.4%
	AND	5704	5705	100%	6.9	7.3	93.7%
moria‡	LH	7820			28	>100	
	AND	7822			7.1	>100	
twmc‡	LH	4331	4331	100%	1.9	2.2	83.7%
	AND	4327	4327	100%	1.7	2.0	84.1%

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

Table 3: Average size of a slice (left) and average time to compute a slice (right).

The right side of the Table 3 shows T, the average time to compute a slice using information provided by light-weight context recovery, and T', the average time to compute a slice without using information provided by light-weight context recovery. The table also shows the ratio of T to T'. From the table, we see that using information provided by light-weight context recovery, we can significantly reduce the time required to compute interprocedural slices. This suggests that that our technique might be very effective in improving the efficiency of many program analyses.

5 RELATED WORK

In previous work, we observed that memory locations pointed to by the same pointers in a procedure have the same program information in the procedure. We developed a technique [9] that partitions such memory locations into equivalence classes. Memory locations in an equivalence class share the same program information in a procedure. Therefore, when the procedure is analyzed, only the information for a representative of

[5]Data for some programs are unavailable: it takes more than 10 hours to compute alias information using [6].

each equivalence class is computed. This information is then reused for other memory locations in the same equivalence class. Experiments [9, 10] show that this technique can effectively improve the performance of program analyses. Using information provided by light-weight context recovery improves the performance of program analyses in another dimension, and thus, can be used together with equivalence analysis to further improve the efficiency of program analyses.

Horwitz, Reps, and Binkley [5] present a technique that uses the sets of variables that may be modified or may be referenced by a procedure to avoid including unnecessary callsites in a slice. This technique is needed so that a system-dependence-graph based slicer can compute slices that are as precise as those computed by other interprocedural slicers (e.g. [4]). Our technique differs from theirs in that our technique uses the sets of memory locations that may be accessed by a procedure under a specific callsite to filter spurious program information. Thus, our technique can improve the precision and performance of many program analyses on which Horwitz, Reps, and Binkley's technique cannot apply.

There are many other techniques that can improve the performance of program analyses. For example, program fragment analysis [12] can analyze an arbitrary portion of a large system by using a less precise estimate about the whole system that can be obtained using a less expensive algorithm. Our light-weight context-recovery technique can be used with many of these approaches to improve further the performance of data-flow analyses. For techniques such as fragment analysis that trade precision for efficiency, using our technique might be able to compensate for the loss of precision.

6 CONCLUSION AND FUTURE WORK

We presented a light-weight context recovery algorithm, and illustrated a technique that uses the information provided by the light-weight context recovery to improve the precision and the efficiency of program analyses. We also conducted several empirical studies. The results of our studies suggest that, in many cases, light-weight context recovery can be quite effective in improving the precision and efficiency of program analyses.

There are several areas in which we will continue our work. First, we will perform the empirical studies presented in this paper on larger programs to further validate the conclusions that we made. Second, we will perform empirical studies to evaluate the effectiveness of combining light-weight context recovery with equivalence analysis in improving the efficiency of computing interprocedural slices. Third, we will apply our technique to other program analyses and evaluate its effectiveness on those program analyses. Finally, we will perform empirical studies to compare our technique with conditional analysis.

ACKNOWLEDGMENTS

This work was supported by NSF under grants CCR-9696157 and CCR-9707792 to Ohio State University.

REFERENCES

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

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

[3] Programming Languages Research Group. PROLANGS Analysis Framework. Rutgers University, http://www.prolangs.rutgers.edu/, 1998.

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

[5] 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.

[6] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of SIGPLAN '92 Conf. on Prog. Lang. Design and Implem.*, pages 235–248, June 1992.

[7] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[8] 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*, pages 199–215, September 1999.

[9] 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*, pages 39–46, September 1999.

[10] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *International Conference on Software Maintenance*, pages 421–430, September 1999.

[11] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations in C programs. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[12] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 235–252, September 1999.

[13] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *The 4th International Symposium on Static Analysis*, pages 16–34, September 1997.

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