Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses

Donglin Liang and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{dliang,harrold}@cc.gatech.edu

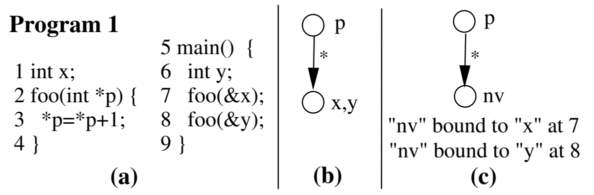
Abstract

Pointer information that is provided by many algorithms identifies a memory location using the same name throughout a program. Such pointer information is inappropriate for use in analyzing C programs because, using such information, a program analysis may propagate a large amount of spurious information across procedure boundaries. This paper presents a modular algorithm that efficiently computes *parameterized* pointer information in which symbolic names are introduced to identify memory locations whose addresses may be passed into a procedure. Because a symbolic name may identify different memory locations when the procedure is invoked under different callsites, using parameterized pointer information can help a program analysis reduce the spurious information that is propagated across procedure boundaries. The paper also presents a set of empirical studies, that demonstrate (a) the efficiency of the algorithm, and (b) the benefits of using parameterized pointer information over using non-parameterized pointer information in program analyses. The studies show that using parameterized pointer information may significantly improve the precision and the efficiency of many program analyses.

1. INTRODUCTION

Various pointer analysis techniques have been developed to facilitate program analyses of C programs. To support these program analyses, a pointer analysis must associate names with memory locations. A pointer analysis also must provide information that helps program analyses determine the memory locations that may be accessed through pointer dereferences. With this information, a program analysis can first replace the pointer dereferences in a C program with the memory locations accessed through such dereferences, and then analyze the program in the usual way [2, 14].

Pointer analysis algorithms can differ in the way in which



Program 1

```
                      5 main() {
1 int x;              6   int y;
2 foo(int *p) {       7   foo(&x);
3   *p=*p+1;          8   foo(&y);
4 }                   9 }
```

Figure 1: (a) Program 1, (b) non-parameterized points-to graph, (c) parameterized points-to graph.

they assign names to memory locations. Such differences can significantly impact the precision and the efficiency of the program analyses that use the pointer information. Many existing pointer analysis algorithms (e.g., [1, 3, 5, 6, 10, 12, 13, 16, 17]) use the same name to identify a memory location throughout the program. These algorithms use variable names to identify memory locations allocated on the stack, and use an artificial name to identify the memory locations allocated on the heap at a specific statement. Because a memory location may be accessed throughout the program, its name can appear in several procedures. Therefore, a program analysis that uses this pointer information usually treats such name as if it were a global variable.

Only a few existing pointer analysis algorithms [7, 19] assign different names for a memory location in different procedures. When the address of a memory location can be passed into a procedure through formal parameters or global pointers, these algorithms use a symbolic name to identify the memory location within the procedure. If the pointer information computed for the procedure is used under more than one calling context, the symbolic name can be used to identify different memory locations under different calling contexts. For example, the algorithms can use a symbolic name `nv` to identify the memory locations whose addresses are passed into procedure `foo()` (Figure 1(a)) through `p`. When the pointer information computed for `foo()` is used under the context of statement 7, `nv` identifies `x`. When this pointer information is used under the context of statement 8, `nv` identifies `y`. The symbolic names introduced by these algorithms act like reference parameters. Thus, we refer to such symbolic names as *auxiliary parameters*, and we refer to pointer information that contains auxiliary parameters as *parameterized* pointer information.

For supporting program analyses of C programs, parameterized pointer information has several advantages over non-parameterized pointer information. First, parameterized pointer information for a procedure is more compact than non-parameterized pointer information. In a procedure, parameterized pointer information can use an auxiliary parameter to represent a set of memory locations. In contrast, non-parameterized pointer information may require several names for the same set of memory locations. Thus, when a program analysis analyzes a procedure using parameterized pointer information, it creates and propagates less information than using non-parameterized pointer information. For example, using parameterized pointer information, a program analysis would propagate information for `nv` in procedure `foo()` in Figure 1(a). In contrast, using non-parameterized pointer information, the program analysis would propagate information for `x` or `y` in `foo()`. Second, without special consideration, a program analysis algorithm that uses non-parameterized pointer information may propagate spurious information across procedure boundaries. For example, using non-parameterized pointer information, a program analysis finds that `x` or `y` may be modified at statement 3 in Figure 1 and propagates this information back to statement 7. Thus, it reports that `foo()` may modify `x` and `y` at statement 7. In contrast, using parameterized pointer information, a program analysis finds that `nv` is modified at statement 3. When this information is propagated back to statement 7, because `nv` identifies the memory location for `x` under the context of statement 7, the program analysis reports that only `x` is modified by `foo()` at statement 7. Therefore, using parameterized pointer information, the program analysis can reduce the amount of spurious information propagated across procedure boundaries.

One major problem with existing algorithms that compute parameterized pointer information is efficiency. First, existing algorithms use a flow-sensitive approach. Empirical results suggest that flow-sensitive pointer analysis algorithms may not scale to large programs [7, 12, 19]. Second, existing algorithms may analyze a procedure more than once [7, 19]. This additional analysis increases the expense of these algorithms. Third, existing algorithms might compute several versions of pointer information for each procedure. Using these different versions of pointer information may increase the complexity of program analyses.

To efficiently compute parameterized pointer information, we developed a modular parameterized pointer analysis algorithm (MoPPA). MoPPA efficiently computes pointer information for a program following a framework similar to the flow-insensitive, context-sensitive pointer analysis algorithm (FICS) [13], which we previously developed. Like FICS, MoPPA computes, in three phases, points-to graphs that represent the pointer information for a program. Unlike FICS, which identifies a memory location using the same name throughout a program, MoPPA uses, when possible, auxiliary parameters to identify memory locations whose addresses are passed into a procedure. MoPPA also distinguishes the memory locations that are dynamically allocated in a procedure when the procedure is invoked under different calling contexts.

Compared to FICS and other algorithms (e.g., [1, 6, 17]) that are intended to handle large programs, a major benefit of MoPPA is that it provides parameterized pointer information. The parameterized pointer information lets a program analysis reduce the spurious information propagated across procedure boundaries. Such reduction may significantly improve both the efficiency and the precision of program analyses. Another benefit of MoPPA over these algorithms is that MoPPA can distinguish the memory locations dynamically allocated in a procedure under different calling contexts. Therefore, MoPPA may provide more precise pointer information than these algorithms.

Compared to other existing algorithms that compute parameterized pointer information, a major benefit of MoPPA is its efficiency. MoPPA processes each pointer assignment only once. In the second and the third phases, MoPPA propagates, from one procedure to another, only a small amount of information that is related to parameters. Therefore, MoPPA can efficiently compute the points-to graphs. Another benefit of MoPPA over existing algorithms is its modularity — only the information for the procedures within a strongly connected component on the call graph must be in memory simultaneously. Therefore, compared to existing algorithms that compute parameterized pointer information, MoPPA requires less memory.

This paper first discusses the approach used by MoPPA to assign names to memory locations. The paper then presents the details of the MoPPA algorithm. The paper also presents a set of empirical studies that demonstrate (a) the efficiency of MoPPA and (b) the benefits of using parameterized pointer information provided by MoPPA over using non-parameterized pointer information provided by FICS in program analyses. To the best of our knowledge, these studies are the first that compare the results of program analyses computed using parameterized pointer information with that computed using non-parameterized pointer information. The studies show that using parameterized pointer information can significantly improve the precision and the efficiency of many program analyses.

2. PARAMETERIZED POINTS-TO GRAPHS

This section first briefly introduces the points-to graphs constructed by MoPPA, and then discusses the approach used by MoPPA to assign names to memory locations.

2.1 Points-to Graphs

MoPPA uses points-to graphs to represent pointer information. In a points-to graph, a node represents a set of memory locations, whose names are associated with the node. A *field access* edge, labeled with a field name, connects a node representing structures to a node representing a specific field of the structures. A *points-to* edge, labeled with "*", represents points-to relations. For example, the points-to graph in Figure 1(b) represents the fact that `p` may point to `x` or `y`. For efficiency, MoPPA imposes two constraints on a points-to graph: (1) each memory location can be represented by only one node; (2) labels are unique among the edges leaving a node. Similar constraints are also used to implement Steensgaard's algorithm [17] and FICS [13].

For a given program, MoPPA computes a *global* points-to graph that represents the pointer information related to global pointers. For each procedure in the program, MoPPA computes a *procedural* points-to graph that represents the pointer information related to the local pointers in the procedure. The separation of global pointer information from local pointer information lets MoPPA reduce the amount of information that it propagates across procedure boundaries.

For example, suppose that a statement in a procedure forces a global pointer g to point to a heap-allocated memory location. Without the separation, this pointer information would have to be propagated to every procedure in the program, even if g is irrelevant to the computation of the pointer information for the procedure. MoPPA avoids such propagation by making this information available in the global points-to graph. When a program analysis analyzes a program, it resolves the dereferences of global pointers using the global points-to graph.

2.2 Naming Memory Locations

Unlike FICS, which uses the same name to identify each memory location throughout a program, MoPPA uses an auxiliary parameter, when possible, to identify a memory location in the points-to graph for a procedure P. To support program analyses, MoPPA also provides binding information that maps an auxiliary parameter in P to the names that identify the same memory locations at the callsites to P. For example, MoPPA uses auxiliary parameter `nv` to identify memory locations for `x` and `y` in the points-to graph (Figure 1(c)) for `foo()` in program 1 (Figure 1(a)). It also provides information to map `nv` to `x` at statement 7 and to `y` at statement 8. Unlike FICS, which uses one artificial name to identify all the memory locations allocated on the heap at a statement s in a procedure P, MoPPA attempts to distinguish the memory locations allocated at s when P is invoked from different callsites. Unlike other algorithms (e.g., [5]) that blindly distinguish these memory locations by extending their names with call strings, MoPPA makes such distinction only if the distinction may help the program analysis compute more precise information.

To handle heap-allocated memory locations and global variables effectively, MoPPA introduces quasi-global names. A *quasi-global name* is a name whose scope may include several procedures, but does not necessarily include all procedures in a program. MoPPA can identify a memory location *loc* using a quasi-global name in some procedures, and identify *loc* using auxiliary parameters in the other procedures that access *loc*. Such a naming scheme is more flexible than that used in existing algorithms, in which a memory location must be identified either using a *global name* whose scope includes all the procedures or using a set of *local names* whose scopes include only individual procedures.

MoPPA uses information collected from a program to determine the kind of name that it uses to identify a memory location in a procedure. We consider global variables, local variables, and heap-allocated memory locations. For each global variable g accessed within a procedure P, MoPPA determines whether g is only accessed using its address that is passed into P through formal parameters. If this is the case, MoPPA uses an auxiliary parameter to identify g in P. For example, MoPPA uses auxiliary parameter `nv` to identify `x` in `foo()` in Figure 1(a). However, if g is accessed using its variable name or using an address that is passed into P through global pointers, then MoPPA uses g's variable name as the quasi-global name to identify g in P (e.g., `x` in `main()` in Figure 1(a)). This quasi-global name is also used to identify g in P's direct or indirect callers.

For a local variable l that is declared in P, if l cannot be accessed through dereferences of global pointers in the program, MoPPA uses l's variable name as a local name to identify l in P. In any other procedure where l may be accessed, MoPPA uses an auxiliary parameter to identify l. However, if l can be accessed through dereferences of global pointers in the program, then MoPPA identifies l using a technique that is similar to the one used to identify a global variable. In P or in a procedure where l's address may be passed into the procedure through global pointers or dereferences of global pointers, MoPPA uses l's variable name as a quasi-global name to identify l. In a procedure where l's address is passed into the procedure only through formal parameters or dereferences of formal parameters, MoPPA uses an auxiliary parameter to identify l.

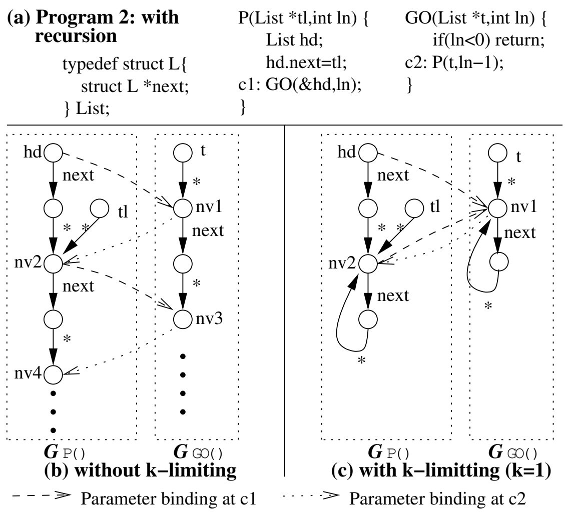
In general, MoPPA uses the following rules to determine the scope of a quasi-global name N:

- **Rule 1.** If N is the variable name of a global variable and N syntactically appears in a procedure P, then N's scope includes P.
- **Rule 2.** If the memory location identified by N is pointed to by the memory location identified by another quasi-global name N_1 according to the global points-to graph, and N_1's scope includes a procedure P, then N's scope includes P.
- **Rule 3.** If N's scope includes a procedure P, then N's scope includes all the procedures that call P.

Rule 1 considers global variables. Rule 2 determines whether the address of a memory location can be passed into a procedure through global pointers or dereferences of global pointers. Rule 3 accounts for the fact that P is a part of its callers. These rules can be efficiently evaluated by MoPPA.

MoPPA uses a local name, an auxiliary parameter, or a quasi-global name to identify a memory location that is allocated on the heap within a procedure P. Suppose that a statement s in P allocates memory locations on the heap. We consider three cases. In the first case, the addresses of these memory locations are not returned to P's callers. In this case, these memory locations can be accessed only within P. MoPPA identifies these memory locations using an *auxiliary local name* whose scope includes only P. In the second case, the addresses of these memory locations may be returned to P's callers through the return value or dereferences of formal parameters, but not through global pointers or dereferences of global pointers. MoPPA uses an auxiliary parameter to identify these memory locations in P. MoPPA also creates names, using similar rules, to identify these memory locations in P's callers. Because different names may be created to identify the memory locations returned by P at different callsites, MoPPA can distinguish, in P's callers, the memory locations allocated at s under different calling contexts. In the third case, the addresses of these memory locations may be returned to P's callers through global pointers or dereferences of global pointers. MoPPA introduces a quasi-global name to identify these memory locations. According to Rule 3, this name will be used in all callers of P. Therefore, in this case, MoPPA does not distinguish memory locations allocated at s under different calling contexts.

For example, consider the points-to graphs (Figure 3(d)) computed by MoPPA for Program 3 in Figure 3(a). Let *loc* be the memory location allocated at statement 14. In $G_{\texttt{alloc()}}$, the points-to graph for `alloc()`, MoPPA uses auxiliary parameter `nv2` to identify *loc*. That *loc* is returned to `alloc()`'s callers only through `*f` is the basis for this naming decision. When *loc* is returned to `getg()` at statement


(a) Program 2: with recursion

```
typedef struct L{
    struct L *next;
} List;

P(List *tl,int ln) {
    List hd;
    hd.next=tl;
c1: GO(&hd,ln);
}

GO(List *t,int ln) {
    if(ln<0) return;
c2: P(t,ln−1);
}
```

Figure 2: Problem with recursion and its solution.

10, MoPPA identifies *loc* using a quasi-global name `gh`. That *loc* may be returned to `getg()`'s callers not only through the return value, but also through global pointer `g`, is the basis for this naming decision. When *loc* is returned to `main()` at statement 3, MoPPA identifies *loc* using an auxiliary local name `lh`. That *loc* cannot be returned to `main()`'s callers is the basis for this naming decision. Compared to the points-to graphs (Figure 3(e)) constructed by FICS, we can see that MoPPA computes more precise pointer information.

2.3 Handing Recursion

In the presence of recursive data structures, MoPPA may introduce an infinite number of auxiliary parameters in procedures involved in recursion. For example, MoPPA keeps generating auxiliary parameters to identify the memory locations in a linked-list in construction of the points-to graphs ($G_{\texttt{P()}}$, $G_{\texttt{GO()}}$ in Figure 2(b)) for `P()` and `GO()` in Figure 2(a). The same problem occurs with other algorithms [7, 19].

MoPPA solves this problem using a variant of *k-limiting* [11]. The variant considers simple paths[1] that contain *suspicious* nodes — nodes associated only with auxiliary parameters. The variant limits the number of consecutive suspicious nodes on a simple path to k (MoPPA ignores field nodes when it looks for consecutive suspicious nodes). When MoPPA processes a recursive call, if it needs to add a suspicious node, it checks the restriction. If adding the new node would create a simple path containing more than k consecutive suspicious nodes, MoPPA searches, on the path, for an existing suspicious node that represents memory locations whose types overlap the types of the memory locations represented by the new node. If MoPPA finds such a node, it uses this node. Otherwise, it uses a new node. For example, when MoPPA binds `nv2` to `GO()` at callsite `c1` in `P()` (Figure 2(a)), it attempts to add a new suspicious node in $G_{\texttt{GO()}}$. Adding such a node creates a simple path that contains two consecutive suspicious nodes. Thus, if k is 1, MoPPA reuses the node that is associated with `nv1` and binds `nv2` to `nv1`. Figure 2(c) shows the resulting points-to graphs.

[1]A *simple* path does not contain two identical nodes.

3. COMPUTATION OF PARAMETERIZED POINTS-TO GRAPHS

This section first introduces some definitions, and then gives an overview of MoPPA.

3.1 Definitions

Memory locations in a program are accessed through *object names*, each of which consists of a variable and a possibly empty sequence of dereferences and field accesses [12]. Object name N_1 is *extended* from object name N_2 if N_1 can be constructed by applying a possibly empty sequence of dereferences and field accesses ω to N_2; in this case, we denote N_1 as $\mathcal{E}_\omega\langle N_2\rangle$. For example, suppose that p is a pointer that points to a **struct** with field a in a C program. Then $\mathcal{E}_*\langle p\rangle$ is $*p$, $\mathcal{E}_*\langle *p\rangle$ is $**p$, and $\mathcal{E}_{*.a}\langle p\rangle$ is $(*p).a$.

Given an object name N, if N is of pointer type, then the *points-to node* of N in a points-to graph G is the node that represents the memory locations that may be pointed to by N. To find N's points-to node in G, an algorithm first locates or creates, in G, a node n_0 that represents the variable in N. The algorithm then locates or creates a sequence of nodes n_i and edges e_i, $1 \le i \le k$, so that $n_0, e_1, n_1, ..., e_k, n_k$ is a path in G, the labels of $e_1, ..., e_{k-1}$ match the sequence of dereferences and field accesses in N, and e_k is a points-to edge. N's points-to node is n_k.

3.2 Overview of MoPPA

In addition to computing the points-to graphs, MoPPA also computes, in various phases, the set of quasi-global names whose scopes may include a procedure P according to Rules 1–3 in Section 2. MoPPA uses this information to determine the kind of name that it uses to identify a memory location in P. To compute this information, MoPPA first collects the global variable names that syntactically appear in P or in procedures directly or indirectly called by P. According to Rules 1 and 3, the scopes of these names include P. MoPPA then searches, beginning from the nodes associated with the global variable names computed for P, for all reachable nodes in G_{glob}. The names associated with these nodes identify the memory locations whose addresses may be passed into P through global pointers or dereferences of global pointers. According to Rule 2, the scopes of these names include P.

MoPPA computes the points-to graphs for a program in three phases (Figure 4). In each phase, MoPPA performs two tasks. The first task detects each pair of object names that may point to common memory locations. MoPPA merges the points-to nodes of these two object names in a points-to graph to ensure that each common memory location pointed to by these two object names is represented by only one node. This merging operation is a variant of the "join" in Steensgaard's algorithm [17]. The second task determines the memory locations represented by each node in the points-to graphs. MoPPA picks appropriate names to identify these memory locations at the node. We discuss how these two tasks are performed in each phase.

3.2.1 First phase: Lines 1–8

In the first phase, MoPPA processes each pointer assignment $lhs = rhs$ in each procedure P to build P's points-to graph G_P. If rhs is an object name, then MoPPA merges the points-to nodes of lhs and rhs in G_P to capture the fact that lhs and rhs point to the same memory location

(a) Program 3

```
1 main() {
2   char *p,*q;
3   alloc(&p);
4   q =getg();
5   g=a;
6 }
7 char *getg() {
8   char **t=&g;
9   if(g==null)
10    alloc(t);
11  return *t;
12 }
13 alloc(char **f) {
14   *f=malloc(4);
15 }
16 char *g,a[4];
```

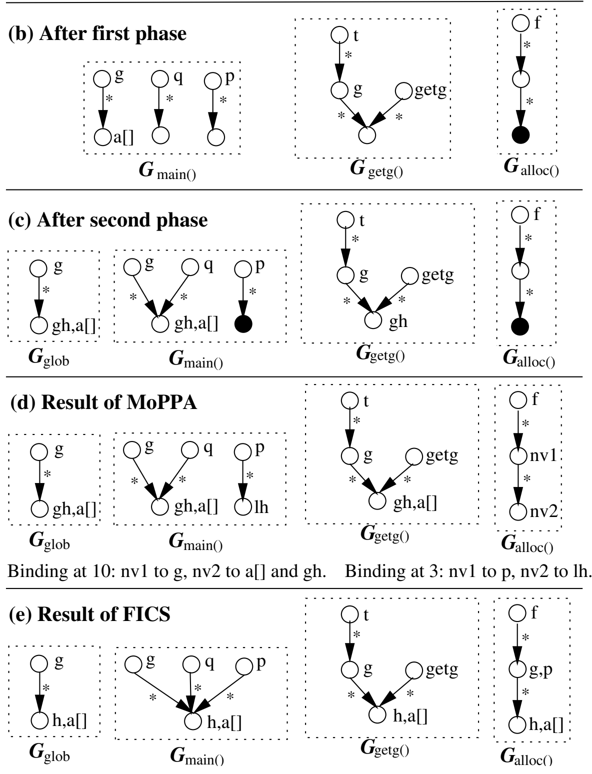


Figure 3: Program 3 and its points-to graphs.

```
Algorithm   MoPPA
input       $\mathcal{P}$: the program to be analyzed
output      a set of points-to graphs
declare     $GVars[P]$: global variable names collected for $P$
function    $\mathcal{A}_c(f)$: return the actual parameter bound to $f$ at $c$
            globals($G$): return the global variable names in $G$

 begin MoPPA
/* First phase (lines 1–8) */
 1. foreach pointer assignment $lhs = rhs$ in each procedure $P$ do
 2.   case $rhs$ do
 3.     object name: merge points-to nodes of $lhs$ and $rhs$ in $G_P$
 4.     "&x": add x to the points-to node of $lhs$ in $G_P$
 5.     malloc(): set HasHeap of $lhs$'s points-to node in $G_P$
 6.   endcase
 7. endfor
 8. add global variable names in each procedure $P$ to $GVars[P]$
/* Second phase (lines 9–30) */
 9. add all procedures in $\mathcal{P}$ to worklists $W_1$ and $W_2$
10. while $W_1 \neq \phi$ do /* $W_1$: sorted in reversed topological order*/
11.   remove $P$ from the head of $W_1$
12.   foreach callsite $c$ to $Q$ in $P$ do
13.     BindFromCallee($G_Q$,globals($G_Q$),$G_P$,$c$)
14.     foreach points-to node $N$ of $\mathcal{E}_\omega\langle f\rangle$ in $G_Q$
                  where $f$ is a formal parameter do
15.       copy names from $N$ to $\mathcal{E}_\omega\langle\mathcal{A}_c(f)\rangle$'s points-to node in $G_P$
16.       if HasHeap of $N$ is set then
17.         set HasHeap of $\mathcal{E}_\omega\langle\mathcal{A}_c(f)\rangle$'s points-to node in $G_P$
18.     endfor
19.   endfor
20.   BindToGlobal($G_P$,globals($G_P$),$G_{glob}$)
21.   foreach points-to node $N$ of $\mathcal{E}_\omega\langle g\rangle$ in $G_P$, $g$ is global do
22.     copy names from $N$ to points-to node of $\mathcal{E}_\omega\langle g\rangle$ in $G_{glob}$
23.     if HasHeap of $N$ is set then
24.       reset HasHeap of $N$ add a new name to $N$
25.       add the new name to $\mathcal{E}_\omega\langle g\rangle$'s points-to node in $G_{glob}$
26.     endif
27.   endfor
28.   add procedures calling $P$ to $W_1$ if $G_P$ is updated
29. endwhile
30. compute $GVars[P]$ for each procedure $P$ from $P$'s callees
/* Third phase (lines 31–55) */
31. foreach procedure $P$ do
32.   compute the quasi-global names whose scopes include $P$
33. while $W_2 \neq \phi$ do     /* $W_2$: sorted in topological order */
34.   remove $P$ from the head of $W_2$
35.   foreach callsite $c$ to $P$ in $P'$ do
36.     BindFromCaller($G_{P'}$,$c$,$G_P$)
37.     foreach name $n$ in points-to node of $\mathcal{E}_\omega\langle a\rangle$ in $G_{P'}$
                  and $a$ is actual parameter bound to $f$ at $c$ do
38.       if $n$ is quasi-global name whose scope includes $P$ then
39.         add $n$ to $\mathcal{E}_\omega\langle f\rangle$'s points-to node in $G_P$
40.       elseif no auxil parameter at $\mathcal{E}_\omega\langle f\rangle$'s points-to node in $G_P$
41.         create a auxil parameter at $\mathcal{E}_\omega\langle f\rangle$'s points-to node in $G_P$
42.       endif
43.     endfor
44.   endfor
45.   BindFromGlobal($G_{glob}$,globals($G_P$),$G_P$)
46.   foreach name $n$ in $\mathcal{E}_\omega\langle g\rangle$'s points-to node in $G_{glob}$
                  where $g$ is global pointer appeared in $G_P$ do
47.     add $n$ to $\mathcal{E}_\omega\langle g\rangle$'s points-to node in $G_P$
48.   foreach node $N$ whose HasHeap is set in $G_P$ do
49.     reset HasHeap
50.     if no auxiliary parameter associated with $N$ then
51.       add a new local name to $N$
52.   endfor
53.   add the procedures called by $P$ to $W_2$ if $G_P$ is updated
54. endwhile
55. foreach callsite $c$ do compute binding information endfor
 end MoPPA
```

Figure 4: MoPPA algorithm.

after this assignment (line 3). If *rhs* is an address-taking expression "&x", then MoPPA adds variable name x to the points-to node of *lhs* in G_P to indicate that *lhs* points to the memory location for x after the assignment (line 4). If *rhs* is a call to a memory allocation function, MoPPA sets a boolean flag HasHeap for the points-to node of *lhs* in G_P (line 5). HasHeap of a node is used to indicate that the node represents a heap-allocated memory location whose name has not yet been determined by MoPPA. In various phases of the algorithm, when two nodes N_1 and N_2 are merged, if HasHeap of N_1 or N_2 is set, then HasHeap of the resulting node will be set.

In the first phase, MoPPA introduces a variable to represent the return value of each function and treats a return statement as an assignment. For example, MoPPA introduces getg to represent the return value of function getg() in Figure 3(a), and treats return statement 11 as assignment getg=*t. In the first phase, MoPPA also collects the global variable names that syntactically appear in P (line 8).

Figure 3(b) shows the points-to graphs constructed by MoPPA during this phase for Program 3 in Figure 3(a). A solid node in the graphs indicates that HasHeap of this node is set.

3.2.2 *Second phase: Lines 9–30*

In the second phase, MoPPA processes the callsites in each procedure P to consider the effects, on G_P, of the procedures called by P. Let c be a callsite in P and Q be the procedure that c invokes. MoPPA first calls BindFromCallee() to detect pairs of parameter-related object names that may point to the same memory locations (line 13). BindFromCallee() searches in G_Q for object names $\mathcal{E}_{\omega1}\langle p\rangle$ and $\mathcal{E}_{\omega2}\langle q\rangle$ that point to the same node. If p and q are formal parameters bound to a_1 and a_2 respectively at c, then after c is executed,

$\mathcal{E}_{\omega 1}\langle a_1\rangle$ and $\mathcal{E}_{\omega 2}\langle a_2\rangle$ may point to common memory locations. Thus, `BindFromCallee`() merges the points-to nodes of $\mathcal{E}_{\omega 1}\langle a_1\rangle$ and $\mathcal{E}_{\omega 2}\langle a_2\rangle$ in G_P. If p is a formal parameter bound to a at c and q is a global variable, then after c is executed, $\mathcal{E}_{\omega 1}\langle a\rangle$ and $\mathcal{E}_{\omega 2}\langle q\rangle$ may point to common memory locations. Thus, `BindFromCallee`() merges the points-to nodes of $\mathcal{E}_{\omega 1}\langle a\rangle$ and $\mathcal{E}_{\omega 2}\langle q\rangle$ in G_P. For example, when MoPPA processes statement 4 in Figure 3(a), it merges the points-to nodes of `q` and `g` in $G_{\texttt{main()}}$ because $G_{\texttt{getg()}}$ shows that `getg` and `g` point to the same node (MoPPA treats return value `getg` as a formal parameter).

MoPPA also determines the memory locations whose addresses may be returned to P at callsite c (lines 14–18). If G_Q shows that a name `x` is associated with the points-to node of $\mathcal{E}_{\omega}\langle f\rangle$, in which f is a formal parameter bound to a at c, then, after c is executed, $\mathcal{E}_{\omega}\langle a\rangle$ may point to the memory locations identified by `x`. MoPPA adds `x` to the points-to node of $\mathcal{E}_{\omega}\langle a\rangle$ in G_P (line 15). If `HasHeap` of the points-to node of $\mathcal{E}_{\omega}\langle f\rangle$ is set, then after c is executed, $\mathcal{E}_{\omega}\langle a\rangle$ may point to memory locations that are allocated from the heap in Q. Thus, MoPPA sets `HasHeap` of the points-to node of $\mathcal{E}_{\omega}\langle a\rangle$ (line 17). For example, when MoPPA processes statement 3 in Figure 3(a), it sets `HasHeap` of the points-to node of `p` in $G_{\texttt{main()}}$ because $G_{\texttt{alloc()}}$ shows that `HasHeap` of the points-to node of `*f` is set.

In the second phase, MoPPA also constructs the global points-to graph G_{glob} using information in G_P (lines 20–27). MoPPA calls `BindToGlobal`() to search in G_P for object names $\mathcal{E}_{\omega 1}\langle g_1\rangle$ and $\mathcal{E}_{\omega 2}\langle g_2\rangle$, where g_1 and g_2 are global variables that point to the same node. `BindToGlobal()` merges the points-to nodes of $\mathcal{E}_{\omega 1}\langle g_1\rangle$ and $\mathcal{E}_{\omega 2}\langle g_2\rangle$ in G_{glob} to indicate that $\mathcal{E}_{\omega 1}\langle g_1\rangle$ and $\mathcal{E}_{\omega 2}\langle g_2\rangle$ point to the same memory locations.

MoPPA also determines the memory locations that may be pointed to by object names extended from global variables (lines 21–27). Let g be a global variable. If G_P shows that `x` is associated with the points-to node of $\mathcal{E}_{\omega}\langle g\rangle$, then when P is executed, $\mathcal{E}_{\omega}\langle g\rangle$ may point to the memory location identified by `x`. MoPPA adds `x` to the points-to node of $\mathcal{E}_{\omega}\langle g\rangle$ in G_{glob} to capture this information (line 22). If G_P shows that `HasHeap` of the points-to node of $\mathcal{E}_{\omega}\langle g\rangle$ is set, then when P is executed, $\mathcal{E}_{\omega}\langle g\rangle$ may point to a memory location allocated in P. MoPPA creates a new quasi-global name to identify this memory location, and adds this name to the points-to node of $\mathcal{E}_{\omega}\langle g\rangle$ in both G_P and G_{glob}. MoPPA also resets `HasHeap` of the points-to node of $\mathcal{E}_{\omega}\langle g\rangle$ in G_P to indicate that the heap-allocated memory location has been assigned a name. For example, when MoPPA processes `getg()` in Figure 3(a) in the second phase, it finds that `HasHeap` of the points-to node of `g` is set. Thus, the algorithm creates a name `gh`, and adds this name to the points-to node of `g` both in $G_{\texttt{getg()}}$ and in G_{glob}. The algorithm also resets `HasHeap` of `g`'s points-to node in $G_{\texttt{getg()}}$.

In the second phase, MoPPA further computes the set of global variable names that appear syntactically in procedure P (line 30). In this phase, MoPPA processes the procedures in a reverse topological (bottom-up) order on the strongly-connected components of the call graph. Within a strongly-connected component, MoPPA iterates over the procedures until the points-to graph computed for each procedure stabilizes. Figure 3(c) shows the points-to graphs constructed for Program 3 in Figure 3(a) after this phase.

3.2.3 *Third phase: Lines 31–55*

In the third phase, MoPPA processes each procedure P to determine the memory locations represented by each node in G_P and assigns appropriate names to identify these memory locations. MoPPA completes this task in four steps. First, MoPPA computes, by using G_{glob} and the set of global variable names computed for P in the first two phases, the set of quasi-global names whose scopes include P (lines 31–32).

Second, MoPPA processes each callsite c that calls P to capture the pointer information introduced by parameter bindings. Let P' be the procedure that contains c. MoPPA first calls `BindFromCaller`() to detect pairs of object names that are extended from actual parameters at c and may point to the same node in $G_{P'}$ (line 36). If `BindFromCaller`() finds that $\mathcal{E}_{\omega 1}\langle a_1\rangle$ and $\mathcal{E}_{\omega 2}\langle a_2\rangle$, in which a_1 and a_2 are bound to f_1 and f_2 respectively at c, may point to the same node in $G_{P'}$, it merges the points-to nodes of $\mathcal{E}_{\omega 1}\langle f_1\rangle$ and $\mathcal{E}_{\omega 2}\langle f_2\rangle$ in G_P. MoPPA also determines the memory locations that may be pointed to by object names extended from formal parameters (lines 37–43). Let a be an actual parameter that is bound to formal parameter f at c. If $G_{P'}$ shows that name n is associated with the points to node of $\mathcal{E}_{\omega}\langle a\rangle$, then when P is invoked at c, $\mathcal{E}_{\omega}\langle f\rangle$ may point to the memory locations identified by n at P's entry. If n is a quasi-global name whose scope includes P, then MoPPA adds n to the points-to node of $\mathcal{E}_{\omega}\langle f\rangle$ in G_P. Otherwise, MoPPA checks to see if there is an auxiliary parameter associated with the points-to node of $\mathcal{E}_{\omega}\langle f\rangle$ in G_P. If no auxiliary parameter exists, then MoPPA creates a new auxiliary parameter and adds this auxiliary parameter to this node. For example, when MoPPA processes the callsite to `alloc()` at statement 10 in Figure 3(a), it finds that `t` may point to `g`. Because `g`'s scope does not include `alloc()`, MoPPA introduces auxiliary parameter `nv1` to identify this memory location and adds `nv1` to the points-to node of `f` in $G_{\texttt{alloc()}}$. Note that in the third phase, if two nodes N_1 and N_2 are merged, at most one auxiliary parameter is kept in the resulting node.

Third, MoPPA further determines, by examining G_{glob}, the memory locations that may be represented by each node in G_P (lines 45–47). Let g_1 and g_2 be global variables that appear in G_P (i.e., $g_1, g_2 \in \texttt{globals}(\texttt{G}_\texttt{P})$). MoPPA calls `BindFromGlobal`() to search, in G_{glob}, object names $\mathcal{E}_{\omega 1}\langle g_1\rangle$ and $\mathcal{E}_{\omega 1}\langle g_1\rangle$ that point to the same node. `BindFromGlobal`() merges the points-to nodes of $\mathcal{E}_{\omega 1}\langle g_1\rangle$ and $\mathcal{E}_{\omega 1}\langle g_1\rangle$ in G_P. Let g be a global variable that appear in G_P. If G_{glob} shows that name n is associated with the points-to node of $\mathcal{E}_{\omega}\langle g\rangle$, then MoPPA adds n to the points-to node of $\mathcal{E}_{\omega}\langle g\rangle$ in G_P.

Fourth, MoPPA assigns names for the unnamed heap-allocated memory locations represented by nodes in G_P (lines 48–52). MoPPA examines, in G_P, each node N whose `HasHeap` is set. If an auxiliary parameter *aux* is associated with N, then N is pointed to by an object name extended from formal parameters. Therefore, the heap-allocated memory locations represented by N may be returned to P's callers. MoPPA reuses *aux* to identify these memory locations. However, if no auxiliary parameter is associated with N, then these heap-allocated memory locations are not returned to P's callers. MoPPA creates a new local name and add this name to N to identify these heap-allocated memory locations. In both cases, MoPPA resets `HasHeap` of N. For example, when MoPPA examines $G_{\texttt{alloc()}}$, the points-to graph for `alloc()` in Figure 3(a), it discovers that `HasHeap` of the points-to node of `*f` is set and an auxiliary parameter `nv2` is associated with this node. Therefore, it reuses `nv2` to identify the heap-allocated memory locations represented by this node. However, when MoPPA examines $G_{\texttt{main()}}$, it discovers that `HasHeap` of the points-to node of `p` is set but no auxiliary parameter is associated with this node. Therefore, it creates a local name `lh` to identify the heap-allocated memory locations represented by this node (Figure 3(d)).

In the third phase, MoPPA processes the procedures in a topological (top-down) order on the strongly-connected components of the call graph. Within a strongly-connected component, MoPPA iterates over the procedures until the points-to graph for each procedure stabilizes. After all the points-to graphs stabilize, MoPPA processes each callsite c to compute the binding information between the names in the procedure containing c and the auxiliary parameters in the called procedure (line 55). This step can be done on-demand when the pointer information is used.

Figure 3(d) shows the points-to graphs that MoPPA computes for Program 3. Compared to the points-to graphs (Figure 3(e)) constructed by FICS for this program, we can see that MoPPA computes more compact and more precise pointer information than FICS.

3.3 Complexity of MoPPA

Let p be the number of procedures in a program $\mathcal{P}$, c be the number of callsites in $\mathcal{P}$, and S be the worst-case actual size of the points-to graph for a procedure. Without considering the cost of line 8 and lines 30–32, the time complexity of MoPPA is the same as the time complexity of FICS, which is $O(N * S * \alpha(N * S, p * S))$ [13], given that α is the inverse Ackermann function, N is $(c + p)$ in the absence of recursion, and N is $(c + p) * S$ in the presence of recursion. The steps taken at lines 8 and 30 are very similar to those taken in the computation of modification side-effects for the procedures. Therefore, the time taken by these two lines is $O(n^2)$. Line 32 can be done by first mapping the names in $GVars[P]$ to the nodes in G_{glob}, and then searching in G_{glob} beginning from these nodes. Therefore, the time taken by this line is $O(n + S_{glob})$, where S_{glob} is the size G_{glob}. Thus, the time complexity of MoPPA is $O(p * (n + S_{glob}) + n^2 + N * S * \alpha(N * S, p * S))$. With the assumption that S_{glob} is $O(n)$, the time complexity can be simplified to $O(n^2 + N * S * \alpha(N * S, p * S))$.

3.4 Handling Indirect Calls

Because the complete call graph for a program that contains indirect calls through function pointers is not directly available, MoPPA may encounter difficulties in analyzing such a program. There are two possible solutions. The first solution uses the call graph computed by another algorithm. For example, we can construct the call graph for a program by resolving indirect calls using pointer information provided by Steensgaard's algorithm. We can further refine the result using function prototypes [2].

The second solution begins the analysis with a partial call graph, and computes the complete call graph during the analysis. This approach requires iterations between the bottom-up phase and the top-down phase [4]. To use the second approach, MoPPA keeps an extra *shadow* points-to graph $\widehat{G}_P$ for each procedure P.[2] MoPPA uses $\widehat{G}_P$ to separate the summary information about P from the pointer information computed for P. In the first phase, MoPPA computes the pointer information and puts the information in both $\widehat{G}_P$ and G_P. In the second phase when MoPPA processes a callsite c to Q in P, it uses $\widehat{G}_Q$ to update both $\widehat{G}_P$ and G_P. In the second phase when MoPPA computes the global points-to graph, it uses the shadow points-to graphs. In the third phase, MoPPA uses only the normal points-to graphs for the procedures. Before computing the binding information for each callsite at the end of this phase, MoPPA first examines each indirect call. If MoPPA discovers new callees, it expands the call graph and repeats the second and third phases starting only from the procedures that might be affected. Otherwise, the algorithm computes the binding information and terminates.

[2]$\widehat{G}_P$ may be eliminated if MoPPA can determine the procedures that are directly or indirectly called by P.

	Subject Size			*Time*	
program	*LOC*	*Nodes*	*Procs*	T_M	T_F
dixie	2100	1357	52	0.30	0.19
learn	1600	1596	50	0.31	0.17
diff	1730	1932	44	0.34	0.18
assembler	2510	1993	58	0.67	0.44
smail	3212	2430	59	0.43	0.30
lharc	3235	2539	89	0.51	0.25
simulator	3558	2992	114	0.50	0.27
flex	6902	3762	93	0.89	0.32
rolo	4748	3874	142	0.90	0.54
space	11474	5601	137	1.75	1.36
bison	7893	6533	134	1.10	0.62
spim	24322	11352	263	3.43	2.49
mpgplay	17263	11864	135	3.68	2.36
espresso	12864	15351	306	6.01	4.61
moria	25002	20316	482	7.70	3.34
twmc	23922	22167	247	3.92	4.24

Table 1: Left: Sizes of the subject programs. Right: Time in seconds for MoPPA (T_M) and FICS (T_F).

4. EMPIRICAL STUDIES

We have implemented a prototype of MoPPA using the PROLANGS Analysis Framework (PAF) [9]. Our prototype handles function pointers using the first approach discussed in Section 3.3. We have also performed several empirical studies to evaluate the performance of MoPPA and the effectiveness of using the parameterized pointer information provided by MoPPA in program analyses. We collected the data for the studies on a Sun Ultra 30 workstation with 640MB of physical memory. To allow the algorithm to capture the pointer information introduced by calls to library functions, we created a set of stubs that simulate these functions. A similar approach using stubs has been used in other prototypes (e.g. [12, 13]).

The left side of Table 1 shows the subject programs we used. Column *LOC* shows the number of lines of code, column *Nodes* shows the number of control flow graph nodes created to represent each program, and column *Procs* shows the number of procedures. These subject programs have also been used in many other studies [12, 13, 15].

4.1 Study 1

The goal of this study is to evaluate the performance of MoPPA. To investigate the time efficiency of MoPPA, we compare the time required to run MoPPA and the time required to run FICS on each subject program. The right side

	# of heap		# of flow dependences		
program	Mo	FI	Mo	FI	Reduce
dixie	10	7	5.10	6.19	17.7%
learn	4	4	3.30	3.64	9.4%
diff	20	7	3.40	3.57	4.8%
assembler	17	15	2.68	3.62	25.8%
smail	12	7	2.60	3.20	19.0%
lharc	3	3	2.47	2.52	2.0%
simulator	3	3	2.49	2.58	3.4%
flex	39	7	5.85	6.57	11.0%
rolo	27	10	3.96	4.41	10.3%
space	11	11	2.83	3.03	6.5%
bison	83	81	7.88	7.89	0.0%
spim	131	17	26.6†	42.8†	37.9%
mpgplay	64	58	6.94	6.97	0.4%
espresso	238	111	4.10	4.31	4.8%
moria	1	1	9.17	11.66	21.3%
twmc	144	113	4.26	4.51	5.6%

† `spim` contains two large procedures with over 1000 nodes. Without considering these two procedures, the result for MoPPA is 2.83, and the result for FICS is 2.95.

Table 2: Left: Number of distinct heap-allocated memory locations. Right: Average number of flow dependences for a statement.

	Size			Time	
program	Mo	FI	Reduce	Mo	FI
dixie	94.3	139.4	32.3%	0.3	3.2
learn	21.8	25.0	12.8%	0.2	0.7
diff	30.3	34.3	11.7%	0.1	0.1
assembler	121.2	174.5	30.6%	0.6	11.7
smail	52.7	123.3	57.3%	0.2	42.1
lharc	64.6	113.7	43.2%	0.3	1.6
simulator	264.4	317.6	16.8%	0.4	2.4
flex	138.0	223.9	38.4%	2.1	4.4
rolo	68.2	95.6	28.6%	0.5	3.1
space	128.6	136.7	5.9%	0.5	47.4
bison†	1120	1164	3.8%	5.1	10.0
spim†	2107	⋆	–	742.7	⋆
mpgplay†	2054	2170	5.3%	19.4	23.8
espresso†	2888	3321	13.0%	35.9	6209
moria†	3146	⋆	–	621.3	⋆
twmc†	1152	2387	51.7%	11.4	598.4

† Data are collected on one data slice.
⋆ Data are unavailable because the system does not terminate within the time limit (10 hours) we set.

Table 3: Left: average size of a data slice, Right: average time in seconds to compute a data slice.

(T_M, T_F) of Table 1 shows the comparison. The table shows that, on the programs we studied, although MoPPA can be 2 to 3 times slower than FICS, it is still very efficient for all programs. This results suggest that MoPPA will scale to large programs as well as FICS.

In the study, we also investigate the effectiveness of MoPPA in distinguishing memory locations allocated on the heap in a procedure when the procedure is invoked under different callsites. The left side of table 2 compares the number of distinguishable heap-allocated memory locations when the pointer information is computed by MoPPA or FICS. Two memory locations are not distinguishable if they are are always used in the same way in a program. Column Mo shows the results for MoPPA and column FI shows the results for FICS. The table shows that, for several programs, MoPPA identifies many more distinguishable heap-allocated memory locations than FICS. These results suggest that using pointer information provided by MoPPA may help a program analysis compute more precise information.

4.2 Study 2

The goal of this study is to evaluate the impact of using pointer information provided by MoPPA and FICS on the computation of flow dependence, one variety of data dependence, within a procedure. A statement s_1 is *flow-dependent* on another statement s_2 if s_1 may use the value set by s_2. Flow dependence has been used in important tasks such as program optimization and program understanding.

In this study, we computed the average number of statements on which a statement is flow-dependent. For each callsite, we use its side-effects to compute the flow dependences. The right side of table 2 shows the results of this study when the pointer information is provided by MoPPA (*Mo*) or FICS (*FI*). The table also shows the percentage of spurious flow dependences (*Reduce*) that can be eliminated by using pointer information provided by MoPPA. The table shows that, for several programs (e.g., `smail`), using pointer information provided by MoPPA can significantly ($> 10\%$) reduce the spurious flow dependences. Thus, on these programs, using pointer information provided by MoPPA may significantly improve the precision of the program analyses that require data-flow information. Note that, for other programs (e.g., `lharc`) on which the reduction in flow dependences is insignificant, using pointer information provided by MoPPA may still improve the precision of program analyses on these programs by reducing the spurious information propagated across procedure boundaries.

4.3 Study 3

The goal of this study is to evaluate the impact of using pointer information provided by MoPPA or FICS on the precision and the efficiency of program analyses that require transitive interprocedural flow dependence. The study consists of two parts. The first part of the study considers the impact on the computation of transitive flow dependence. We measure the average number of statements that can transitively affect a specific statement s through flow dependence. For convenience, we refer to these set of statement as the *data slice* with respect to s. We also measure the average time to compute a data slice. These measurements can serve as an indicator to the impact of using such pointer information on program analyses that require transitive interprocedural flow dependence.

Table 3 shows these two measurements we obtain when the pointer information is provided by MoPPA (*Mo*) or FICS (*FI*). The table also shows the reduction in the size of a data slice (*Reduce*) when the pointer information is provided by MoPPA instead of FICS. We obtain the data by running a modified version of our reuse-driven slicer [14] on each subject. The table shows that, for many programs we studied (e.g., `smail`), using pointer information provided by MoPPA can significantly improve the precision and the efficiency of the computation of transitive flow dependence.

The second part of the study considers the impact of using pointer information provided by MoPPA or FICS on program slicing [18], a program analysis that requires transitive flow dependences. We measure the average size of a

	Size			*Time*	
program	*Mo*	*FI*	*Reduce*	*Mo*	*FI*
dixie	612.4	653.3	6.3%	4.2	20.3
learn	469.1	489.4	4.1%	8.9	24.6
diff	283.1	293.6	3.6%	0.9	1.6
assembler	640.6	753.0	14.9%	6.0	79.1
smail	675.5	824.7	18.1%	5.6	205.3
lharc	560.5	697.6	19.6%	6.7	47.4
simulator	1172	1176	0.3%	5.1	18.6
flex†	602	1088	44.6%	16.9	22.9
rolo†	1131	1283	11.8%	19.7	97.4
space†	2249	2504	10.2%	9.5	539.8
bison†	2823	2824	0.0%	21.1	48.5
spim†	3443	⋆	–	4085	⋆
mpgplay†	3950	4140	4.6%	91.3	127.0
espresso†	5125	5744	10.8%	187.2	13560
moria†	⋆	⋆	–	⋆	⋆
twmc†	12884	12897	0.1%	851.1	19379

† Data are collected on one program slice.

⋆ Data are unavailable because the system does not terminate within the time limit (10 hours) we set.

Table 4: Left: Average size of a program slice, Right: Average time in seconds to compute a program slice.

program slice and the average time to compute a program slice. Table 4 shows these two measurements obtained in the study when the pointer information is provided by MoPPA (*Mo*) or (*FI*). The table also shows the reduction in the size of a program slice (*Reduce*) when the pointer information is provided by MoPPA instead of FICS. We obtain the data by running our reuse-driven program slicer on each subject. The table shows that, for many programs that we studied (e.g., `smail`), using pointer information provided by MoPPA can significantly improve the precision and the efficiency.

By considering the results of both parts of the study, we can conclude that using parameterized pointer information provided by MoPPA may significantly improve the precision and efficiency of many program analyses.

5. RELATED WORK

Several other existing pointer analysis algorithms use a modular approach for computing pointer information. One such algorithm is Chatterjee, Ryder, and Landi's *Relevant Context Inference* (RCI) [3]. Like FICS and MoPPA, RCI first uses a bottom-up phase to consider the effect of a procedure on each callsite that calls this procedure. RCI then uses a top-down phase to compute the memory locations whose addresses may be passed into a procedure.

RCI differs from MoPPA in two ways. First, RCI computes non-parameterized pointer information. RCI uses unknown initial values for parameters and globals at the entry of a procedure. At first glance, these unknown initial values seem to serve the same purpose as auxiliary parameters. However, because unknown initial values are created before the pointer information is computed at the callsites, two unknown initial values may represent the address of the same memory location under a calling context. Therefore, in the final pointer solution, these unknown initial values must be replaced with concrete values.[3] Second, RCI computes pointer information using a flow-sensitive approach. Because propagating information using flow-sensitive approach is expensive, RCI may not scale to large programs.

Another modular pointer analysis algorithm is Cheng and Hwu's algorithm [4]. Like MoPPA, Cheng and Hwu's algorithm is flow-insensitive. Unlike MoPPA and many other algorithms that use exactly one name to identify each memory location at a statement, Cheng and Hwu's algorithm uses access paths[4] to identify each memory location.

One way that MoPPA distinguishes itself from Cheng and Hwu's algorithm is efficiency. We compare these two algorithms in three aspects. First, Cheng and Hwu's algorithm must propagate pointer information for global pointers from procedure to procedure. In contrast, MoPPA uses a global points-to graph to capture the pointer information for the global pointers. Therefore, MoPPA propagates less information across procedure boundaries than Cheng and Hwu's algorithm. Second, in the intraprocedural phase, Cheng and Hwu's algorithm must iterate over the pointer assignments within a procedure in a way similar to that used in Andersen's algorithm [1]. In contrast, MoPPA processes each procedure in the intraprocedural phase using an approach similar to Steensgaard's algorithm [17], which processes each pointer assignment only once. Third, in the interprocedural phases, Cheng and Hwu's algorithm must iterate over the points-to relations computed for a procedure when the algorithm discovers a new points-to relation. In contrast, MoPPA may merge two nodes or add a name to a node when it discovers a new points-to relation. Therefore, MoPPA is more efficient than Cheng and Hwu's algorithm.

Another way that MoPPA distinguishes itself from Cheng and Hwu's algorithm is the support for interprocedural program analyses that use the pointer information. Similar to auxiliary parameters, access paths used in Cheng and Hwu's algorithm can identify different memory locations in a procedure when the procedure is invoked at different callsites. However, because one memory location might be identified by several access paths in pointer information provided by Cheng and Hwu's algorithm, a program analysis using this pointer information may have to propagate more information across procedure boundaries than using pointer information provided by MoPPA. In addition, mapping an access path from a called procedure to a calling procedure is more expensive than mapping an auxiliary parameter. Therefore, using the information provided by Cheng and Hwu's algorithm in program analyses may be less efficient than using the information provided by MoPPA.

Foster, Fahndrich, and Aiken proposed a *polymorphic* flow-insensitive points-to analysis framework that computes pointer information by solving a set of constraints [8]. This framework also consists of a bottom-up phase and a top-down phase to propagate information. When this framework uses term constraint, the resulting algorithm is very similar to FICS. This framework differs from MoPPA in that it computes non-parameterized pointer information. Studies show that their current implementation of the framework may not scale to large programs [8].

Some existing pointer analysis algorithms [3, 12] provide *conditional pointer information*, in which a points-to relation may be associated with a condition that specifies the calling contexts under which this relation may hold. Al-

[3]The same argument applies to *non-visible variables* used in Landi and Ryder's algorithm [12].

[4]An *access path* is similar to an object name defined in this paper.

though such conditions may help a program analysis reduce the amount of spurious information propagated across procedure boundaries [15], adding conditions to the points-to relations may increase the complexity of the pointer analysis. Studies show that existing algorithms that provide conditional pointer information may not scale to large programs [3, 12].

6. CONCLUSION

This paper presents MoPPA, a modular algorithm that computes parameterized pointer information for C programs. The paper also presents a set of empirical studies that compare MoPPA with FICS. The empirical results show that both MoPPA and FICS can efficiently compute pointer information for programs. The empirical results also show that using pointer information provided by MoPPA can significantly improve both the precision and the efficiency of many program analyses.

Due to space limitation, this paper does not present the details of handling memory accesses using knowledge of the physical layout of a structure. Several existing approaches (e.g., [20]) can be incorporated into MoPPA to handle such accesses. Our future work will include investigation of the impact of different approaches on MoPPA.

In future work, we will also conduct more empirical studies on larger programs to investigate the effectiveness of using the pointer information provided by MoPPA in various program analyses. In addition, we will compare the effectiveness of using different approaches to handle function pointers in MoPPA.

Acknowledgments

This work was supported in part by NSF under NYI Award CCR-0096321 and ESS Award CCR-9707792 to Ohio State University, by funds from to Georgia Tech through the Yamacraw Mission, and by a grant from Boeing Aerospace Corporation. Alessandro Orso made many helpful suggestions that improved the presentation of the paper.

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 *6th International Symposium on the Foundations of Software Engineering (FSE-98)*, pages 46–55, Nov. 1998.

[3] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of the 26th Symposium on Principles of programming languages*, pages 133–146, 1999.

[4] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of 2000 Conference on Programming Language Design and Implementation*, pages 57–69, June 2000.

[5] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of 2000 Conference on Programming Language Design and Implementation*, June 2000.

[7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of 1994 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[8] J. S. Foster, M. Fahndrich, and A. Aiken. Polymorphic verus monomorphic flow-insensitive points-to analysis for c. In *Proceedings of 7th International Static Analysis Symposium*, June 2000.

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

[10] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[11] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. 1979.

[12] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, July 1992.

[13] 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, Sept. 1999.

[14] 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, Sept. 1999.

[15] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[16] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–23, June 1995.

[17] 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, Jan. 1996.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[19] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[20] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. *ACM SIGPLAN Notices*, 34(5):91–103, May 1999.