

# Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses\*

Donglin Liang and Mary Jean Harrold

Georgia Institute of Technology, Atlanta GA 30332, USA  
{dliang,harrold}@cc.gatech.edu

**Abstract.** 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. Parameterized pointer information can be used by a client program analysis to compute parameterized summary information for a procedure. The client can then instantiate such information at each specific callsite by binding the symbolic names. Compared to *non-parameterized* pointer information, in which memory locations are identified using the same name throughout a program, parameterized pointer information lets the client reduce the spurious information that is propagated across procedure boundaries. Such reduction will improve not only the precision, but also the efficiency of the client. The paper also presents a set of empirical studies. The studies show that (1) the algorithm is efficient; and (2) using parameterized pointer information may significantly improve the precision and efficiency of program analyses.

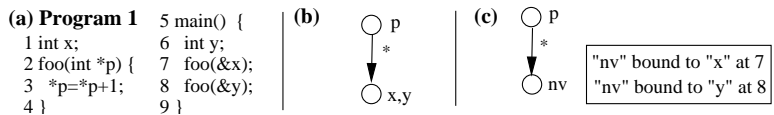
## 1 Introduction

Various pointer analyses 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 must also provide information that determines the memory locations accessed through pointer dereferences. With this information, a program analysis can first replace the pointer dereferences in a program with the memory locations accessed through such dereferences, and then analyze the program in the usual way [2, 16].

Pointer analysis algorithms can differ in the way in which 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, 8, 11, 14, 15, 18, 19]) use the same name to identify a memory location throughout the program. 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 a name as if it were a global variable name.

---

\* Supported by NSF under CCR-9988294, CCR-0096321, and EIA-0196145, by Boeing Aerospace Corporation, and by the State of Georgia under the Yamacraw Mission.



**Fig. 1.** Program 1 (a), non-parameterized (b) and parameterized (c) points-to graph.

A few existing pointer analysis algorithms [7, 21] assign different names to a memory location in different procedures. When the address of a memory location can be passed into a procedure from a callsite, 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 different calling contexts, the symbolic name can be used to identify different memory locations. For example, symbolic name `nv` can be used to identify the memory locations whose addresses are passed into `foo()` (Figure 1(a)) through `p`. Under the context of statement 7, `nv` identifies `x`. Under the context of statement 8, `nv` identifies `y`. The symbolic names used by the algorithms act like reference parameters. Thus, we refer to such symbolic names as *auxiliary parameters*<sup>1</sup>, and refer to pointer information containing auxiliary parameters as *parameterized* pointer information.

For supporting program analyses, parameterized pointer information has several advantages over non-parameterized pointer information. First, parameterized pointer information can be used by a program analysis to compute parameterized summary information for a procedure. Such parameterized summary information can be instantiated at a callsite to compute more accurate information about the callsite. For example, using parameterized pointer information, a program analysis reports that `nv` is modified by `foo()` (Figure 1). The program analysis then instantiates this information at statement 7 by replacing `nv` with `x`, and reports that `x` is modified by `foo()` at statement 7. In contrast, using non-parameterized pointer information, the program analysis reports that both `x` and `y` may be modified by `foo()`. The program analysis then uses this information at statement 7 and reports that `x` and `y` may be modified by `foo()` at statement 7. Second, parameterized pointer information for a procedure is more compact than non-parameterized pointer information: in the parameterized pointer information for a procedure, an auxiliary parameter can be used to represent a set of memory locations that require several names to identify in non-parameterized pointer information. Thus, a program analysis creates and propagates less information when using parameterized pointer information.

The major problem with acceptance of existing algorithms that compute parameterized pointer information is that they are not efficient for analyzing large programs. One reason for this inefficiency is that existing algorithms use a flow-sensitive approach, which may not scale to large programs [7, 14, 21]. A second reason for this inefficiency is that existing algorithms may analyze a procedure more than once [7, 21]. This additional analysis increases the expense of the algorithms. Another problem with acceptance of existing algorithms that compute parameterized pointer information is that none of these algorithms have been compared empirically with algorithms that compute non-parameterized

<sup>1</sup> Auxiliary parameters are similar to *symbolic names* [7] or *extended parameters* [21].

pointer information. Thus, it is unknown how much improvement in precision and performance can be gained by a program analysis that uses parameterized pointer information instead of non-parameterized pointer information.

This paper presents a modular parameterized pointer analysis algorithm (MoPPA) that efficiently computes points-to graphs for a program. MoPPA follows a three-phase flow-insensitive, context-sensitive pointer analysis framework. 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 other algorithms (e.g., [1, 6, 8, 15, 19]) that are intended to handle large programs, a major benefit of MoPPA is that it provides parameterized pointer information. 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 existing algorithms that compute parameterized pointer information, a major benefit of MoPPA is its efficiency. MoPPA processes each pointer assignment only once. By storing global pointer information in one global points-to graph, MoPPA propagates, from one procedure to another, only a small amount of information related to parameters. Therefore, MoPPA can efficiently compute the points-to graphs. Another benefit of MoPPA is its modularity—only the information for the procedures within a strongly connected component of the call graph must be in memory simultaneously. Thus, MoPPA may require less memory.

This paper also presents a set of empirical studies. These studies show that, on subjects of up to 100,000 lines of code,

- MoPPA runs in time close to that required by FICS; such time is close to that required by Steensgaard’s [19], the most efficient flow-insensitive algorithm.
- Using information provided by MoPPA instead of that provided by FICS or Andersen’s algorithm, (a) a program analysis computes on average 12% (maximum 37.9%) fewer flow dependences for a statement in a procedure; (b) a program analysis runs on average 10 (maximum 210 over FICS, maximum 445 over Andersen’s) times faster, and computes on average 25% (maximum 57%) fewer transitive interprocedural flow dependences for a statement in a program; and (c) a program slicer runs on average 7 (maximum 72 over FICS, maximum 106 over Andersen’s) times faster, and computes on average 12% (maximum 45%) smaller slices.

The studies show that (1) MoPPA is efficient and (2) using parameterized pointer information provided by MoPPA can significantly improve the precision and the efficiency of many program analyses.

The significance of this work is that it provides the first algorithm that efficiently (within one minute) computes parameterized pointer information for programs up to 100,000 lines of code. The work also presents, to the best of our knowledge, the first set of empirical studies that compare the results of program analyses computed using parameterized pointer information with that computed

using non-parameterized pointer information. Our studies show that computing parameterized pointer information for large programs is feasible and beneficial.

## 2 Parameterized Points-to Graphs

This section discusses the points-to graphs constructed by MoPPA and 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 field of the structures. A *points-to* edge, labeled with “\*”, represents points-to relations. For example, the edge in Figure 1(b) represents that *p* points to *x* or *y*. For efficiency, MoPPA imposes two constraints on a points-to graph: (1) a memory location can be represented by only one node; (2) labels are unique among the edges leaving a node.<sup>2</sup>

MoPPA computes two kinds of points-to graphs. For a 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 at the beginning of `main()`, global pointer *g* is forced to point to *x*. By making this information available in the global points-to graph, MoPPA avoids propagating such information to procedures in which the computation of the pointer information in the procedures does not involve *g*. When analyzing a program, a program analysis resolves dereferences of global pointers using the global points-to graph.

### 2.2 Naming Memory Locations

MoPPA identifies memory locations in a procedure using three kinds of names: auxiliary parameter, local, and quasi-global. MoPPA uses an auxiliary parameter, when possible, to identify, in procedure *P*, a memory location whose address may be passed into *P* through formal parameters. An auxiliary parameter, as defined in the Introduction, can identify different memory locations under different calling contexts. 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 a callsite 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()` (Figure 1(a)). MoPPA also provides information to map *nv* to *x* at statement 7 and to *y* at statement 8.

MoPPA uses a local name to identify a memory location that cannot be accessed after the procedure returns. A *local name* is a name whose scope includes

---

<sup>2</sup> Similar constraints are also used in Steensgaard’s algorithm [19] and FICS [15].

only one procedure. For example, the memory location for a local variable in procedure  $P$  may be identified using a local name whose scope includes only  $P$ .

MoPPA may use a quasi-global name to identify, in a procedure  $P$ , the memory location for a global variable or a memory location whose address can be passed into  $P$  through global pointers. A *quasi-global name* is a name whose scope may include several procedures, but may not include all procedures in a program. The scope of a quasi-global name ensures that, if a memory location  $loc$  is identified using a quasi-global name  $N$  in  $P$ , then  $loc$  is also identified using  $N$  in  $P$ 's callers. Therefore, MoPPA need not propagate the pointer information for  $loc$  from  $P$  to its callers because such information is stored in the global points-to graph and can be retrieved, using the same name, when the information is needed in  $P$ 's callers. At most one quasi-global name can be used to identify a memory location in different procedures throughout the program.

Using quasi-global names to improve efficiency is one of the features that distinguish MoPPA from Wilson and Lam's algorithm [21]. In their algorithm, all memory locations, including global variables and those that are accessed through global pointers, are identified using extended parameters (similar to auxiliary parameters) in each procedure. Preliminary studies show that many large programs may use a large number of global pointers. For such programs, propagating information for all global pointers from procedure to procedure may be prohibitively expensive. The studies also show that the values of global pointers do not change often in a program. Therefore, introducing symbolic names to represent the memory locations that are accessed through dereferences of global pointers in each procedure might be unnecessary.

MoPPA uses various rules to determine whether to use an auxiliary parameter, a local name, or a quasi-global name to identify global memory locations (global variables), stack-allocated memory locations (local variables), or heap-allocated memory locations in a procedure. For each global variable  $g$  accessed within a procedure  $P$ , MoPPA determines whether  $g$  is accessed only 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, MoPPA uses a quasi-global name to identify  $l$  in the procedures into which  $l$ 's address may be passed through global pointers. MoPPA also identifies  $l$  using this quasi-global name in the callers to these procedures. In the procedures into which  $l$ 's address is passed only through formal parameters or dereferences of formal parameters, MoPPA uses an auxiliary parameter to identify  $l$ .

Identifying local variables with quasi-global names might cause imprecision in the pointer analysis. A local variable  $l$  declared in procedure  $P$  can be accessed only in  $P$  or in the procedures that  $P$  may directly or indirectly call. However, if MoPPA uses a quasi-global name  $N$  to identify  $l$  in the program, then according to the way in which  $N$ 's scope is determined,  $N$  may appear in procedures that  $P$  may never (directly or indirectly) call. Therefore, a program analysis may conclude that  $l$  may be accessed in these procedures and compute spurious information. This kind of imprecision can also be introduced by many other existing algorithms (e.g. [1, 8, 19]). One way to reduce such imprecision is to remove, from  $N$ 's scope, the procedures that  $P$  may never (directly or indirectly) call. However, MoPPA does not include this optimization because preliminary studies show that few local variables may be pointed to by global pointers.

MoPPA attempts to distinguish the memory locations allocated on the heap in a procedure  $P$  under different calling contexts. Unlike algorithms (e.g., [5]) that distinguish these memory locations by extending their names with call strings, MoPPA makes such distinction only if the distinction may improve the precision of program analyses. Suppose that a statement  $s$  in  $P$  allocates memory location  $loc$  on the heap. If  $loc$ 's address is not returned to  $P$ 's callers, then  $loc$  can be accessed only within  $P$ . MoPPA identifies  $loc$  using an *auxiliary local name* whose scope includes only  $P$ . If  $loc$ 's address 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  $loc$  in  $P$ . MoPPA also creates names, using similar rules, to identify  $loc$  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. However, if  $loc$ 's address may be returned to  $P$ 's callers through global pointers or dereferences of global pointers, MoPPA introduces a quasi-global name to identify  $loc$  in  $P$  and in all callers of  $P$ . Therefore, MoPPA does not distinguish memory locations allocated at  $s$  under different calling contexts.

For example, let  $loc$  be the memory location allocated at statement 14 in Figure 2(a). Because  $loc$  is returned to `alloc()`'s callers only through `*f`, MoPPA uses auxiliary parameter `nv2` to identify  $loc$  in  $G_{\text{alloc()}}$  (Figure 2(d)), the points-to graph for `alloc()`. When  $loc$  is returned to `getg()` at statement 10, MoPPA identifies  $loc$  using a quasi-global name `gh` because  $loc$  may be returned to `getg()`'s callers also through global pointer `g`. When  $loc$  is returned to `main()` at statement 3, MoPPA identifies  $loc$  using a local name `lh` because  $loc$  cannot be returned to `main()`'s callers. Compared to the points-to graphs (Figure 2(e)) constructed by FICS, MoPPA computes more precise pointer information.

In summary, to identify memory locations in the procedures with appropriate names, MoPPA first determines the scope of each 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  $N$  identifies a memory location pointed to by another memory location identified by quasi-global name  $N_1$  according to the global points-to graph, and  $N_1$ 's scope includes  $P$ , 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  $P$ .

MoPPA then determines, for a memory location  $loc$  accessed in procedure  $P$ , whether there is a quasi-global name for  $loc$  whose scope includes  $P$ . If so, MoPPA uses this name to identify  $loc$  in  $P$ .

Otherwise, MoPPA determines whether  $loc$  can be accessed before  $P$  is invoked or after  $P$  returns. If that is the case, MoPPA uses an auxiliary parameter to identify  $loc$  in  $P$ . Otherwise, MoPPA uses a local name to identify  $loc$  in  $P$ .

### 3 Computation of Parameterized Points-to Graphs

This section introduces some definitions and 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 [14]. 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  $\omega$  to  $N_2$ ; we denote  $N_1$  as  $\mathcal{E}_\omega\langle N_2 \rangle$ . For example, if pointer  $p$  points to a **struct** with field  $a$  in a C program, then  $\mathcal{E}_*\langle p \rangle$  is  $*p$  and  $\mathcal{E}_{*.a}\langle p \rangle$  is  $(*p).a$ .

Given an object name  $N$  of pointer type, 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 the points-to node for  $N$  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  $e_i$ ,  $1 \leq i \leq k$ , so that  $n_0, e_1, n_1, \dots, e_k, n_k$  is a path in  $G$ , the labels of  $e_1, \dots, e_{k-1}$  match the sequence of dereferences and field accesses in  $N$ , and  $e_k$  is a points-to edge. The points-to node of  $N$  is  $n_k$ .

#### 3.2 Overview of MoPPA

MoPPA computes a global points-to graph  $G_{glob}$  for a program and a procedural points-to graph  $G_P$  for each procedure  $P$ . Let  $g$  be a global pointer. If a memory location  $loc$  may be pointed to by  $\mathcal{E}_\omega\langle g \rangle$  at any point in the program, then a quasi-global name  $N$  identifying  $loc$  must be associated with the points-to node of  $\mathcal{E}_\omega\langle g \rangle$  in  $G_{glob}$ . Let  $v$  be a local pointer declared in  $P$ . If a memory location  $loc$  may be pointed to by  $\mathcal{E}_\omega\langle v \rangle$  at any point in  $P$  under any calling context, then a name  $N$  identifying  $loc$  must be associated with the points-to node of  $\mathcal{E}_\omega\langle v \rangle$  in  $G_P$ . Given the points-to graphs, if a memory location  $loc$  is identified in  $P$  by an auxiliary parameter, then a program analysis can determine the calling contexts under which  $loc$  may be accessed by looking at the binding information at the callsite. However, if  $loc$  is identified in  $P$  by a quasi-global name, a program analysis must assume that  $loc$  may be accessed under each calling context.

In addition to the points-to graphs, MoPPA also computes the set of quasi-global names whose scopes may include  $P$  according to Rules 1–3 in Section 2. MoPPA uses this information to determine the appropriate name for identifying

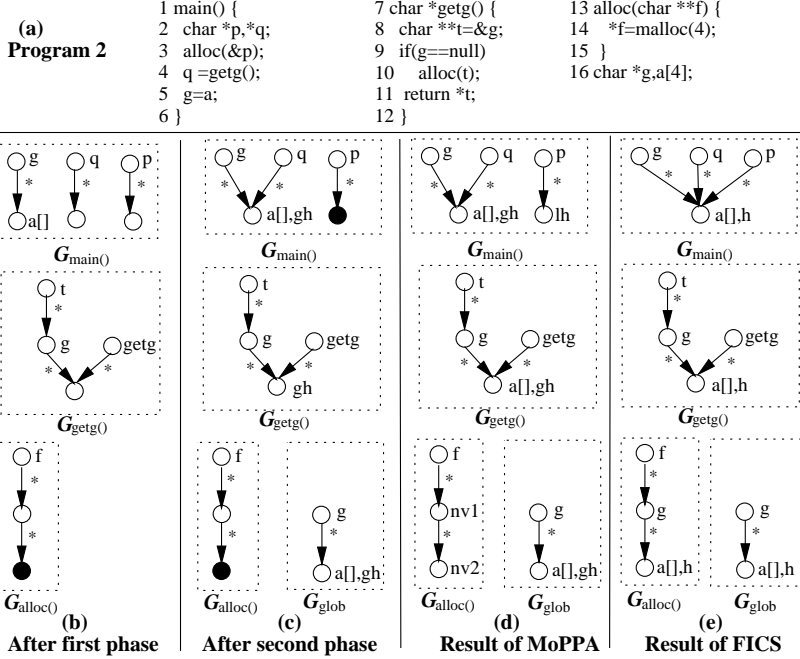


Fig. 2. Program 2 and its points-to graphs.

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 at the nodes associated with the global variable names computed for  $P$ , for all reachable nodes in  $G_{glob}$ . According to Rule 2, the scopes of the names associated with these nodes include  $P$ .

MoPPA performs two major tasks in construction of the points-to graphs. 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 [19]. 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. MoPPA computes the points-to graphs in three phases (Figure 3).

**First phase (lines 1-8).** In the first phase, MoPPA processes each pointer assignment  $lhs = rhs$  in each procedure  $P$  to build  $G_P$ .<sup>3</sup> 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 after this assignment

<sup>3</sup> MoPPA represents the return value of each function with a variable and treats return statements as assignments (e.g., statement 11 in Figure 2(a) is treated as `getg=*t`).

(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  $x$  after the assignment (line 4). If  $rhs$  calls a memory allocation function, MoPPA sets a boolean flag **HasHeap** at the points-to node of  $lhs$  in  $G_P$  (line 5). **HasHeap** of a node indicates that the node represents a heap-allocated memory location whose name has not yet been determined by MoPPA. In various phases of MoPPA, 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 is set. Figure 2(b) shows the points-to graphs constructed by MoPPA during this phase for Program 2 in Figure 2(a). Solid nodes in the graphs indicate that **HasHeap** of these nodes are set.

**Second phase (lines 9-26).** 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 that calls  $Q$  in  $P$ . MoPPA first calls **BindFromCallee()** to search in  $G_Q$  for object names  $\mathcal{E}_{\omega_1}\langle p \rangle$  and  $\mathcal{E}_{\omega_2}\langle q \rangle$  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 the same memory location. 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 pointer, then after  $c$  is executed,  $\mathcal{E}_{\omega_1}\langle a \rangle$  and  $\mathcal{E}_{\omega_2}\langle q \rangle$  may point to the same memory location. 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 2(a), it merges the points-to nodes of  $q$  and  $g$  in  $G_{\text{main}()}$  because **getg** and  $g$  point to the same node in  $G_{\text{getg}()}$  (return value **getg** is treated as a formal parameter in this phase).

MoPPA also determines the memory locations whose addresses may be returned to  $P$  at  $c$  (lines 14–17). 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  $x$ . 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 a heap-allocated memory location. Thus, MoPPA sets **HasHeap** of the points-to node of  $\mathcal{E}_{\omega}\langle a \rangle$  (line 16). For example, when MoPPA processes statement 3 in Figure 2(a), it sets **HasHeap** of  $p$ 's points-to node in  $G_{\text{main}()}$  because **HasHeap** of  $*f$ 's points-to node is set in  $G_{\text{alloc}()}$ .

In the second phase, MoPPA also constructs the global points-to graph  $G_{glob}$  using information in  $G_P$  (lines 19–23). MoPPA first 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  $G_{glob}$ . MoPPA then determines the memory locations that may be pointed to by object names extended from global pointers (lines 20–23). Let  $g$  be a global pointer. If  $G_P$  shows that **HasHeap** of the points-to node of  $\mathcal{E}_{\omega}\langle g \rangle$  is set, then MoPPA creates a new quasi-global name to identify the heap-allocated memory location associated with this node. MoPPA resets **HasHeap** and adds the new name to the points-to node of  $\mathcal{E}_{\omega}\langle g \rangle$  in  $G_P$  (line 21). For example, when MoPPA processes **getg()** in Figure 2(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**, resets **HasHeap**, and adds **gh** to the points-to node of  $g$  in

**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**

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**( $\cdot$ ): set  $HasHeap$  of the points-to node of  $lhs$  in  $G_P$
6.   **endcase**
7. **endfor**
8. add global variable names in each procedure  $P$  to  $GVars[P]$
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** set  $HasHeap$  of  $\mathcal{E}_\omega\langle \mathcal{A}_c(f) \rangle$ 's points-to node in  $G_P$
17.     **endifor**
18.   **endfor**
19.    $BindToGlobal(G_P, globals(G_P), G_{glob})$
20.   **foreach** points-to node  $N$  of  $\mathcal{E}_\omega\langle g \rangle$  in  $G_P$ ,  $g$  is global **do**
21.     **if**  $HasHeap$  of  $N$  is set **then** reset  $HasHeap$  of  $N$  and add a new name to  $N$
22.     copy names from  $N$  to the points-to node of  $\mathcal{E}_\omega\langle g \rangle$  in  $G_{glob}$
23.   **endifor**
24.   **if**  $G_P$  is updated **then** add  $P$ 's callers to  $W_1$
25. **endwhile**
26. compute  $GVars[P]$  for each procedure  $P$  using information from  $P$ 's callees
27. **foreach** procedure  $P$  **do**
28.   compute the quasi-global names whose scopes include  $P$
29. **while**  $W_2 \neq \phi$  **do** /\*  $W_2$ : sorted in topological order \*/
30.   remove  $P$  from the head of  $W_2$
31.   **foreach** callsite  $c$  to  $P$  in  $P'$  **do**
32.      $BindFromCaller(G_{P'}, c, G_P)$
33.     **foreach** name  $n$  at the node of  $\mathcal{E}_\omega\langle a \rangle$  in  $G_{P'}$  and  $a$  is an actual bound to  $f$  at  $c$  **do**
34.       **if**  $n$  is quasi-global name whose scope includes  $P$  **then**
35.         add  $n$  to  $\mathcal{E}_\omega\langle f \rangle$ 's points-to node in  $G_P$
36.       **elseif** no auxil parameter at  $\mathcal{E}_\omega\langle f \rangle$ 's node in  $G_P$  **and** no auxil parameter to reuse **then**
37.         create an auxil parameter at  $\mathcal{E}_\omega\langle f \rangle$ 's points-to node in  $G_P$
38.       **endif**
39.     **endifor**
40.   **endfor**
41.    $BindFromGlobal(G_{glob}, globals(G_P), G_P)$
42.   **foreach** name  $n$  at  $\mathcal{E}_\omega\langle g \rangle$ 's node in  $G_{glob}$  where  $g$  is a global pointer appeared in  $G_P$  **do**
43.     add  $n$  to  $\mathcal{E}_\omega\langle g \rangle$ 's points-to node in  $G_P$
44.   **foreach** node  $N$  whose  $HasHeap$  is set in  $G_P$  **do**
45.     reset  $HasHeap$
46.     **if** no auxiliary parameter associated with  $N$  **then** add a new local name to  $N$
47.   **endifor**
48.   **if**  $G_P$  is updated **then** add  $P$ 's callees to  $W_2$
49. **endwhile**
50. **foreach** callsite  $c$  in  $\mathcal{P}$  **do** compute binding information at  $c$

**end MoPPA**

**Fig. 3.** MoPPA algorithm.

$G_{getg}()$ . MoPPA also propagates names associated with the points-to node of  $\mathcal{E}_\omega\langle g \rangle$  in  $G_P$  to the the points-to node of  $\mathcal{E}_\omega\langle g \rangle$  in  $G_{glob}$ .

In the second phase, MoPPA further computes  $GVars[P]$ , the set of global variable names that appear syntactically in  $P$  or in  $P$ 's callee (line 26). In this phase, MoPPA processes the procedures in a reverse topological (bottom-up)

order on the strongly-connected components of the call graph using a worklist. Figure 2(c) shows the points-to graphs for Program 2 after this phase.

**Third phase (lines 27–50).** In the third phase, MoPPA processes each procedure  $P$  to assign appropriate names to identify the memory locations represented by each node in  $G_P$ . MoPPA completes this task in four steps. First, MoPPA computes, by using  $G_{glob}$  and  $GVars[P]$ , the set of quasi-global names whose scopes include  $P$  (lines 27–28).

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 search for object names  $\mathcal{E}_{\omega_1}\langle a_1 \rangle$  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  $c$ , that point to the same node in  $G_{P'}$  (line 32). MoPPA 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 33–39). Let  $a$  be an actual parameter that is bound to formal parameter  $f$  at  $c$ , and  $n$  be a name identifying memory location  $loc$  in  $G_{P'}$ . If  $n$  is associated with the points-to node of  $\mathcal{E}_{\omega}\langle a \rangle$  in  $G_{P'}$ , then when  $P$  is invoked at  $c$ ,  $\mathcal{E}_{\omega}\langle f \rangle$  may point to  $loc$  at  $P$ 's entry. If  $n$  is a quasi-global name whose scope includes  $P$ , then  $loc$  must be identified by  $n$  in  $P$ . Thus, MoPPA adds  $n$  to the points-to node of  $\mathcal{E}_{\omega}\langle f \rangle$  in  $G_P$ . Otherwise,  $n$  is not a quasi-global name or  $n$  is a quasi-global name but  $n$ 's scope does not include  $P$ . In this case,  $loc$  is identified in  $P$  with an auxiliary parameter. MoPPA checks to see whether 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 further checks the  $k$ -limiting restriction using the approach described in Subsection 3.4. If MoPPA cannot reuse an existing auxiliary parameter, it creates a new auxiliary parameter and adds this auxiliary parameter to this node. For example, when MoPPA processes statement 10 in Figure 2(a), it finds that actual parameter  $\mathbf{t}$  may point to  $\mathbf{g}$ . Because the scope of the quasi-global name for  $\mathbf{g}$  does not include `alloc()`, MoPPA introduces auxiliary parameter `nv1` to identify this memory location and adds `nv1` to the points-to node of  $\mathbf{f}$  in  $G_{\text{alloc}()}$ . 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 nodes in  $G_P$  (lines 41–43). Let  $g_1$  and  $g_2$  be global variable names that appear in  $G_P$  (i.e.,  $g_1, g_2 \in \text{globals}(G_P)$ ). MoPPA calls `BindFromGlobal()` to search, in  $G_{glob}$ , for object names  $\mathcal{E}_{\omega_1}\langle g_1 \rangle$  and  $\mathcal{E}_{\omega_2}\langle g_2 \rangle$  that point to the same node. `BindFromGlobal()` merges the points-to nodes of  $\mathcal{E}_{\omega_1}\langle g_1 \rangle$  and  $\mathcal{E}_{\omega_2}\langle g_2 \rangle$  in  $G_P$ . Let  $g$  be a global variable name that appears 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 44–47). 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 associated with  $N$

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 to identify these memory locations and adds this name to  $N$ . In both cases, MoPPA resets `HasHeap` of  $N$ . For example, MoPPA discovers that, in  $G_{\text{alloc}()}$ , the points-to graph for `alloc()` in Figure 2(a), `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. In another case, MoPPA discovers that, in  $G_{\text{main}()}$ , `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 2(d)).

In the third phase, MoPPA processes the procedures in a topological (top-down) order on the strongly-connected components of the call graph using a worklist. 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 50). This step can be done on-demand when the pointer information is used.

Figure 2(d) shows the points-to graphs that MoPPA computes for Program 2. Compared to the points-to graphs (Figure 2(e)) constructed by FICS for this program, MoPPA computes more compact and more precise pointer information.

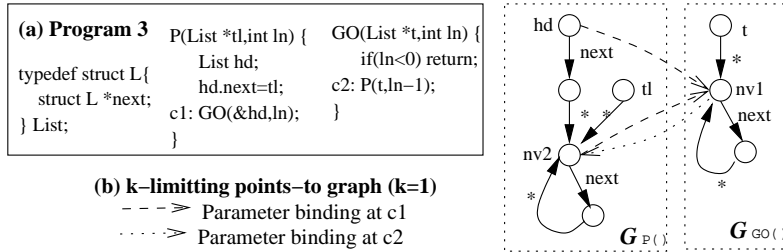
### 3.3 Complexity of MoPPA

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 a procedural points-to graph. Without considering the cost of line 8 and lines 26–28, the time complexity of MoPPA is the same as that of FICS, which is  $O(N * S * \alpha(N * S, p * S))$  [15], given that  $\alpha$  is the inverse Ackermann function,  $N$  is  $(c + p)$  in the absence of recursion and  $(c + p) * S$  in the presence of recursion. The steps taken at lines 8 and 26 are similar to those taken in the modification side-effect analysis. Thus, the time required by these two lines is  $O(n^2)$ . Line 28 can be done by first mapping the names in  $GVars[P]$  to the nodes in  $G_{glob}$ , and then searching in  $G_{glob}$  beginning at these nodes. Thus, the time required by line 28 is  $O(n + S_{glob})$ , where  $S_{glob}$  is the size  $G_{glob}$ . The time complexity of MoPPA is  $O(p * (n + S_{glob}) + n^2 + N * S * \alpha(N * S, p * S))$ .

### 3.4 Handling Recursive Data Structures and Indirect Calls

MoPPA uses a variant of *k-limiting* [13] to handle recursive data structures. The variant limits the number of consecutive *suspicious* nodes—nodes associated with only auxiliary parameters — on a simple path<sup>4</sup> to  $k$  (field nodes are not counted). The restriction is checked only when MoPPA processes a recursive call. For example, when MoPPA binds `nv2` to `G0()` at callsite `c1` in `P()` (Figure 4(a)), it attempts to add a new auxiliary parameter in  $G_{G0()}$ , which would create a simple path with two consecutive suspicious nodes. Thus, when  $k$  is 1, MoPPA reuses `nv1` and binds `nv2` to `nv1`. Figure 4(b) shows the resulting points-to

<sup>4</sup> A *simple* path does not contain two identical nodes.



**Fig. 4.** Example program 3 (a) and its k-limiting points-to graphs (b).

graphs. Note that because `nv2` is supposed to be bound to the memory locations pointed to by `nv1.next`, in the graph, MoPPA creates a new edge from the node representing `nv1.next` to the node representing `nv1`.

MoPPA can use one of the following two solutions to handle programs that contain indirect calls through function pointers. The first solution uses the call graph computed by another algorithm, such as Steensgaard’s algorithm. 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], and thus, increases the complexity of MoPPA. To use the second approach, MoPPA keeps an extra *shadow* points-to graph  $\widehat{G}_P$  for each procedure  $P$  to separate the summary information about  $P$  from the pointer information computed for  $P$ .<sup>5</sup> In the first phase, MoPPA puts the pointer information into both  $\widehat{G}_P$  and  $G_P$ . In the second phase, MoPPA uses  $\widehat{G}_P$  to update both  $\widehat{G}_Q$  and  $G_Q$  if  $P$  is called by  $Q$ , and uses  $\widehat{G}_P$  to update  $G_{glob}$ . In the third phase, MoPPA uses only the normal points-to graphs for the procedures. At the end of the third 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 affected procedures. Otherwise, the algorithm computes the binding information and terminates.

## 4 Empirical Studies

We implemented a prototype of MoPPA using the PROLANGS Analysis Framework (PAF) [10]. Our prototype resolves function pointers using Steensgaard’s algorithm. The prototype is parameterized to treat a structure as either an atomic memory location or a collection of fields. In the latter case, the prototype does not account for accesses that require knowledge of the physical layout of a structure. This limitation may affect the safety of the pointer information. However, it should not significantly affect the validity of our studies because (1) this kind of access is rare in programs, and (2) all the algorithms that we compare are implemented in the same way. More sophisticated techniques will be used to handle such accesses in our future work.

We have performed several empirical studies to evaluate the performance of MoPPA and the effectiveness of using the parameterized pointer information

<sup>5</sup>  $\widehat{G}_P$  may be eliminated if MoPPA can determine the procedures that are directly or indirectly called by  $P$ .

program	Subject Size			Time(s)	
	LOC	Nodes	Procs	$T_M$	$T_F$
dixie	2100	1357	52	0.30	0.19
assem	2510	1993	58	0.67	0.44
smail	3212	2430	59	0.43	0.30
lharc	3235	2539	89	0.51	0.25
simulate	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
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
nethack	32119	31703	701	48.2	132
povray3†	101033	47254	1216	24.1	8.52

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

program	# of heap		# of dependences		
	Mo	FI	Mo	FI	Reduc
dixie	10	7	5.10	6.19	17.7%
assem	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%
simulate	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%
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%
nethack	17	2	5.00	5.40	7.3%
povray3	147	81	10.01	12.12	17.4%

**Table 2.** Left: Number of distinguishable names for heap-allocated locations. Right: Average number of flow dependences for a statement.

† Structures in `povray3` are treated as atomic memory locations.

‡ `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.

provided by MoPPA in program analyses. We compared MoPPA with FICS and Andersen’s algorithm. Other studies (e.g., [6, 12, 15, 9]) show that pointer information computed by these two algorithms is very close in precision to many other existing algorithms, including flow-sensitive algorithms. In addition, MoPPA is implemented using the same framework as FICS. Thus, comparison between MoPPA and FICS can reveal the extra cost required to perform the sophisticated naming scheme used to obtain parameterized pointer information.

We collected the data for the studies on a Sun Ultra 30 workstation with 640MB of physical memory. Structures in programs other than `povray3` are treated as collections of fields. However, structures in `povray3` are treated as atomic memory locations because our system exhausts available memory otherwise.<sup>6</sup> To capture the pointer information introduced by calls to library functions, the prototype used a set of stubs to simulate these functions.<sup>7</sup>

The left side of Table 1 shows our subject programs. For each program, column *LOC* shows the lines of code (comments included), column *Nodes* shows the number of control flow graph nodes, and column *Procs* shows the number of procedures. These subjects have also been used in other studies [14, 15, 17].

<sup>6</sup> Other studies (e.g., [6, 8, 9]) that report results on large programs also treat structures as atomic memory locations.

<sup>7</sup> Similar stubs have been used in other studies (e.g. [14, 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 compared the time to run MoPPA and the time to run FICS on each subject. The right side ( $T_M, T_F$ ) of Table 1 shows the comparison. The time shown in the table excludes time to parse and to resolve function pointers for each subject. The table shows that, for our subjects, although MoPPA can be three times slower than FICS, it is still very efficient. This result suggests that MoPPA will scale to large programs as well as FICS. Note that MoPPA is faster than FICS on `twmc` and `nethack` because MoPPA propagates less information from procedure to procedure.

In the study, we also investigated the effectiveness of MoPPA in distinguishing memory locations allocated on the heap by a procedure under different calling contexts. The left side of Table 2 compares the number of distinguishable names for heap-allocated memory locations used by MoPPA(*Mo*) or by FICS(*FI*). Two names are distinguishable in a program if, according to the pointer information, the memory locations identified by the names are accessed at different sets of statements. A program analysis may compute more precise information when it uses pointer information consisting of more distinguishable names for heap-allocated memory locations. In FICS, we considered the artificial names that represent heap-allocated memory locations. In MoPPA, we considers the quasi-global names and local names that are created for heap-allocated memory locations. The table shows that, for several programs (e.g. `spim`), MoPPA identifies many more distinguishable names for heap-allocated memory locations than FICS. Thus, a program analysis may compute more precise information when using pointer information provided by MoPPA.

## 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 a statement  $s_2$  if  $s_1$  may use the value set by  $s_2$ . Flow dependence can be used in important tasks such as program optimization and data-flow testing.

In this study, we computed the average number of statements on which a statement is flow-dependent. For each callsite, we used its side-effects to compute 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*), and the percentage of spurious flow dependences (*Reduc*) that can be eliminated by using information provided by MoPPA. The table shows that, for several programs (e.g., `morla`), using pointer information provided by MoPPA can significantly (> 10%) reduce the spurious flow dependences, and thus, may significantly improve the precision of program analyses that require data-flow information. Note that, for other programs (e.g., `lharc`) on which the reduction is less significant, using pointer information provided by MoPPA may still improve the precision of program analyses because the spurious information propagated across procedure boundaries may be reduced.

program	Size			Time(s)	
	Mo	FI	Reduc	Mo	FI
dixie	94.3	139.4	32.3%	0.3	3.2
assem	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
simulate	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†	207	293	29.3%	0.6	71.6
spim†	2107	2269	7.1%	743	22k‡
mpgplay†	2054	2170	5.3%	19.4	23.8
espresso†	2888	3321	13.0%	35.9	6209
moria†	3146	*	-	621	*
twmc†	1152	2387	51.7%	11.4	598
nethack†	2628	*	-	1146	*

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

program	Size			Time(s)	
	Mo	FI	Reduc	Mo	FI
dixie	612.4	653.3	6.3%	4.2	20.3
assem	640.6	753.0	14.9%	6.0	79.1
smail	675.5	824.7	18.1%	5.6	205
lharc	560.5	697.6	19.6%	6.7	47.4
simulate	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	540
spim†	3434	*	-	3459	*
mpgplay†	3950	4140	4.6%	91.3	127
espresso†	5125	5744	10.8%	187	14k
moria†	*	*	-	*	*
twmc†	12884	12897	0.1%	851	19k
nethack†	*	*	-	*	*

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

† Data are collected on one slice.

‡ 1k = 1,000.

\* Data are unavailable: the system does not terminate within the time limit (10 hrs.) or runs out of memory. Data for `povray3` are unavailable for the same reason.

### 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 considers the impact on the computation of transitive interprocedural flow dependence. We measured the average number of statements that can transitively affect a specific statement  $s$  through flow dependence. For convenience, we refer to this set of statements as the *data slice* with respect to  $s$ . We also measured the average time to compute a data slice. These measurements serve as an indicator of the impact of using such pointer information on program analyses that require transitive interprocedural flow dependence.

Table 3 shows these two measurements when the pointer information is provided by MoPPA (*Mo*) and FICS (*FI*). The table also shows the reduction in the size of a data slice (*Reduc*) when the pointer information is provided by MoPPA. We obtained the data by running a modified version of our reuse-driven slicer [16]. 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 and FICS on program slicing [20], a program analysis that requires transitive flow dependences. We measured the average size of a program slice and the average time to compute such a slice. We obtained the data by running our reuse-driven program slicer on each subject. Table 4 shows the re-

program	(I) Dependence			(II) Data slice†					(III) Program slice†				
	Size			Size			Time(s)		Size			Time(s)	
	Mo	And	Reduc	Mo	And	Reduc	Mo	And	Mo	And	Reduc	Mo	And
space	2.83	3.03	6.5%	207	296	30.1%	0.6	78.8	2249	2504	10.2%	9.5	620
spim	26.6	38.5	31%	2107	2371	11.1%	743	1849	3434	3704	7.3%	3459	5094
mpgplay	6.94	6.96	0.2%	2054	2170	5.3%	19.4	23.6	3950	4120	4.1%	91.3	115
espresso	4.10	4.50	8.9%	2888	3374	14.4%	35.9	16k‡	5125	5732	10.6%	187	20k
moria	9.17	11.6	21%	3146	*	-	621	*	*	*	-	*	*
twmc	4.26	4.49	5.3%	1152	2385	51.7%	11.4	901	12884	12893	0.1%	851	20k
nethack	5.0	5.33	6.1%	2628	*	-	1146	*	*	*	-	*	*

**Table 5.** Comparing MoPPA (*Mo*) and Andersen’s algorithm (*And*): (I) Average number of flow dependences; (II) Size of a data slice and time in seconds to compute such slice; (III) Size of a program slice and time in seconds to compute such slice.

†Data are reported for one slice.

‡1k = 1,000.

\* Data are unavailable: the system does not terminate within the time limit (10 hours) or runs out of memory. Data for `povray3` are unavailable for the same reason.

sults when the pointer information is provided by MoPPA (*Mo*) and FICS (*FI*). The table also shows the reduction in the size of a program slice (*Reduc*) when the pointer information is provided by MoPPA. The results indicate that, for many our subjects (e.g., `sma1l`), using pointer information provided by MoPPA can significantly improve the precision and efficiency of program slicing.

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.

#### 4.4 Study 4

The goal of this study is to compare MoPPA with Andersen’s algorithm [1] for supporting program analyses. We repeated Studies 2 and 3 on these two algorithms. Table 5 shows the results. Due to space limitations, we show only programs over 10,000 lines of code. The results closely resemble those presented in Studies 2 and 3. For example, for `space`, when information provided by MoPPA is used, Table 5 shows that a slicer runs 65 times faster and computes 10% smaller slices than using that provided by Andersen’s algorithm, whereas Table 4 shows that a slicer runs 56 times faster and computes 10% smaller slices than using that provided by FICS. This similarity is not surprising because other studies [15, 16] show that the information computed by FICS is almost the same as that computed by Andersen’s algorithm. The results strengthen our conclusion that using parameterized pointer information provided by MoPPA may significantly improve the precision and efficiency of program analyses.

## 5 Related Work

Beginning with FICS [15], several flow-insensitive, context-sensitive algorithms [8, 9] that compute one solution for each procedure have been developed. MoPPA

extends one of these algorithms by parameterizing the pointer information for each procedure. Similar approaches can be used to extend other algorithms.

Emami et al.’s algorithm [7] and Wilson and Lam’s algorithm [21] also compute parameterized pointer information. Emami et al.’s algorithm analyzes a procedure under each specific calling context. The algorithm uses symbolic names to represent local variables that are indirectly accessed but not visible in current procedure. The symbolic names let the algorithm reuse the pointer information computed for a procedure under several calling contexts if the alias configuration for inputs are the same under these calling contexts. Wilson and Lam’s algorithm further develops this idea. To increase the opportunity for reuse, the algorithm uses extended parameters to represent global variables and memory locations that can be accessed through dereferences of formal parameters and global pointers in a procedure. In both algorithms, when the alias configuration for inputs of a procedure changes, the procedure must be reanalyzed. MoPPA differs from these two algorithms in that MoPPA analyzes a procedure independently of its calling contexts. Thus, the summary information computed for a procedure can be used for all its calling contexts. The maximum reuse, its flow-insensitivity, and the separation of global information from local information contribute to the efficiency and the scalability of MoPPA.

Other pointer analysis algorithms (e.g., [3, 14]) also use symbolic names to represent memory locations whose addresses are passed into a procedure from calling contexts. However, in these algorithms, because the symbolic names are created before the pointer information at the callsites is computed, two symbolic names may represent the same memory location under a calling context. In the final pointer solution, the symbolic names must be replaced with concrete values. Therefore, these algorithms do not provide parameterized pointer information.

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]. MoPPA and RCI differ in that, (1) RCI computes non-parameterized pointer information, and (2) RCI computes pointer information using a flow-sensitive approach, and thus, may not scale to large programs. Another modular pointer analysis algorithm is Cheng and Hwu’s flow-sensitive algorithm [4], which uses access paths<sup>8</sup> to identify a memory location. One way that MoPPA differs from Cheng and Hwu’s algorithm is efficiency. Cheng and Hwu’s algorithm must propagate global pointer information from procedure to procedure, and must iterate over pointer assignments and points-to relations using an approach similar to Andersen’s algorithm [1]. In contrast, MoPPA reduces the information propagated across procedure boundaries by capturing the global pointer information in a global graph. MoPPA also avoids iteration over pointer assignments and points-to relations by merging points-to nodes using an approach similar to Steensgaard’s algorithm [19]. Another way that MoPPA differs from Cheng and Hwu’s algorithm is its support for interprocedural program analyses. Access paths used in Cheng and Hwu’s algorithm can also identify different memory locations in a procedure under dif-

---

<sup>8</sup> An *access path* is similar to an object name defined in this paper.

ferent calling contexts. However, because one memory location may be identified by several access paths, a program analysis using this pointer information may propagate more information across procedure boundaries than using the 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.

Many flow-insensitive algorithms can be described as building points-to graphs or as generating and solving a set of constraints. Both approaches have advantages and disadvantages. Foster, Fahndrich, and Aiken proposed a *polymorphic* flow-insensitive points-to analysis framework that computes pointer information by solving constraints [9]. When the framework uses term constraints, the resulting algorithm is a variant of FICS. The 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 [9].

Some existing pointer analysis algorithms [3, 14] 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. Although such conditions may help a program analysis reduce the amount of spurious information propagated across procedure boundaries [17], adding conditions to the points-to relations may increase the complexity of the pointer analysis. Studies show that these algorithms may not scale to large programs [3, 14].

## 6 Conclusion

This paper presents MoPPA, a modular algorithm that computes parameterized pointer information for C. The paper also presents studies that compare MoPPA with FICS and Andersen’s algorithm. The studies show that MoPPA is efficient and that using parameterized pointer information provided by MoPPA can significantly improve the precision and efficiency of program analyses.

Due to space limitations, this paper does not present the details of handling memory accesses using constructs such as `casting` that require knowledge of the physical layout of a structure. Several existing approaches (e.g., [22]) can be incorporated into MoPPA to handle such accesses. Our future work will include investigation of the impact of different approaches on MoPPA. Our future work will also include additional empirical studies on larger programs.

## 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*, pages 46–55, Nov. 1998.
3. R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *26th Symposium on Principles of programming languages*, pages 133–146, Jan. 1999.

4. B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *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 *Symposium on Principles of Programming Languages*, pages 232–245, Jan. 1993.
6. M. Das. Unification-based pointer analysis with directional assignments. In *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 *Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
8. M. Fahndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation*, pages 253–263, June 2000.
9. J. S. Foster, M. Fahndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of 7th International Static Analysis Symposium*, June 2000.
10. P. L. R. Group. PROLANGS Analysis Framework. <http://www.prolangs.rutgers.edu/>, Rutgers University, 1998.
11. 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.
12. M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, Aug. 2000.
13. 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.
14. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Conference on Programming Language Design and Implementation*, pages 235–248, July 1992.
15. 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.
16. 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.
17. 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.
18. E. Ruf. Context-insensitive alias analysis reconsidered. In *Conference on Programming Language Design and Implementation*, pages 13–23, June 1995.
19. B. Steensgaard. Points-to analysis in almost linear time. In *23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.
20. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
21. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.
22. 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.