Context-Sensitive, Procedure-Specific Points-To Analysis*

Donglin Liang and Mary Jean Harrold
The Ohio State University
{dliang,harrold}@cis.ohio-state.edu

Abstract

This paper presents a context-sensitive, procedure-specific points-to analysis algorithm that computes a separate points-to graph for each procedure. The algorithm utilizes the fact that a procedure references only a small set of variables, and computes aliasing information that is almost as precise as that computed by Andersen's algorithm and almost as efficient as Steensgaard's algorithm. In some cases, our algorithm computes even more precise aliasing information than Andersen's algorithm because it propagates information only through legal call/return sequences (context-sensitive). Our empirical studies show that our algorithm scales to large programs better than Andersen's algorithm: on the large subject programs on which Andersen's algorithm requires a large amount of time, our algorithm computes the aliasing information in time close to that used by Steensgaard's algorithm. Our empirical studies also show that flow-insensitive aliasing analysis algorithms, such as our algorithm and Andersen's algorithm, can compute aliasing information that is close in precision to the aliasing information computed by the more expensive flow-sensitive and context-sensitive aliasing analysis algorithms.

Keywords: Aliasing analysis, points-to graph, pointer analysis.

1 Introduction

Many program analyses, optimization, and software tools require the computation of safe alias information. Alias information can be computed with flow-sensitive, context-sensitive algorithms [5, 11, 18]. *Flow-sensitive* algorithms account for the order in which statements are executed; *context-sensitive* algorithms consider only legal call-return sequences. These flow-sensitive, context-sensitive algorithms compute alias information that holds at specific program points — we refer to these algorithms as *program-point specific* algorithms. The potentially large number of iterations required by these algorithms, however, makes them costly in both time and space. Thus, they may not be efficient for large programs, which limits their applicability in practice.

Alias information can be computed more efficiently with *flow-insensitive*, *context-insensitive* algorithms [1, 15, 17], which ignore the program's control flow and calling structure. These flow-insensitive, context-insensitive algorithms compute alias information that holds at every program point — we refer to these algorithms as *program-specific* algorithms. These algorithms run faster and require less memory than flow-sensitive algorithms.

Steensgaard's algorithm [17], the most efficient of these program-specific algorithms, runs in time almost linear in the size of the program. In many cases, however, the algorithm can report a large number of

*This work was supported in part by a grant from Microsoft Inc., by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University.

spurious variables as being modified by a statement in a program [19]. Thus, the information that the algorithm computes is often too imprecise to be useful for many applications. Andersen's algorithm [1], on the other hand, provides much more precise alias information than Steensgaard's algorithm. The algorithm, however, may require iteration among pointer-related assignments ($O(n^3)$ time where n is the program size), and requires that the entire program be in memory during analysis. Thus, this algorithm may still be too expensive in time and space to be applicable to large programs. Shapiro and Horwitz [15] propose algorithms that have lower complexity than Andersen's algorithm and can provide more precise alias information than Steensgaard's algorithm. However, their algorithms do not deal with fields of structures. Their experiments show that, on average, their algorithms are an order of magnitude slower than Steensgaard's algorithm, and they compute solutions with precisions that vary from that provided by Steensgaard's algorithm to that provided by Andersen's algorithm; thus, they cannot provide sufficiently precise aliasing information for some programs.

Our context-sensitive, procedure specific algorithm computes, for each procedure, the points-to information that holds at all points in that procedure — we refer to this algorithm as a *procedure-specific* algorithm. Our algorithm also considers the calling context, which means that it propagates information only through legal call/return sequences. Hence, our algorithm can compute more precise points-to information than Steensgaard's algorithm. Our algorithm has three phases. In the first phase, our algorithm uses an approach similar to Steensgaard's, to process pointer-related assignments and compute points-to information for each procedure in a program. In the second phase, our algorithm uses a bottom-up approach to propagate points-to information from the called procedures (callees) to the calling procedures (callers). Finally, in the third phase, our algorithm uses a top-down approach to propagate points-to information from callers to callees.[1]

The main benefit of our algorithm is that it efficiently computes a points-to solution with high precision. Like Steensgaard's algorithm, our algorithm provides safe points-to information by processing each pointer-related assignment only once. However, our algorithm computes a separate points-to graph for each procedure. Because a single procedure typically contains only a few pointer-related variables and assignments, for large programs, our algorithm computes equivalence classes that are much smaller than those computed by Steensgaard's algorithm, and provides points-to information that is almost as precise as that computed by Andersen's algorithm. Our experiments show that, for the programs we studied, our algorithm runs almost as fast as Steensgaard's, but provides solutions close to Andersen's. Another benefit of our algorithm is that it is modular. Because procedures in a strongly-connected component of the call graph are in memory only thrice – once for each phase – our algorithm is more suitable than Andersen's for analyzing large programs.

There are two main contributions of this paper. First, it presents our alias-analysis algorithm. The paper describes the algorithm, illustrates it with an example, and compares it to other alias-analysis algorithms. Second, it gives the results of several studies in which we compare our algorithm, two other flow-insensitive algorithms (Steensgaard's and Andersen's), and Landi and Ryder's flow-sensitive algorithm [11]. Ours is the first study to compare Andersen's algorithm, the most precise flow- and context-insensitive alias-analysis algorithm, with a flow- and context-sensitive algorithm. Our studies suggest that, for many programs, the flow- and context-insensitive algorithm can compute a solution that is close to that computed by a flow- and context-sensitive algorithm.

[1]Future work includes extending our algorithm to handle function pointers using an approach similar to [2].

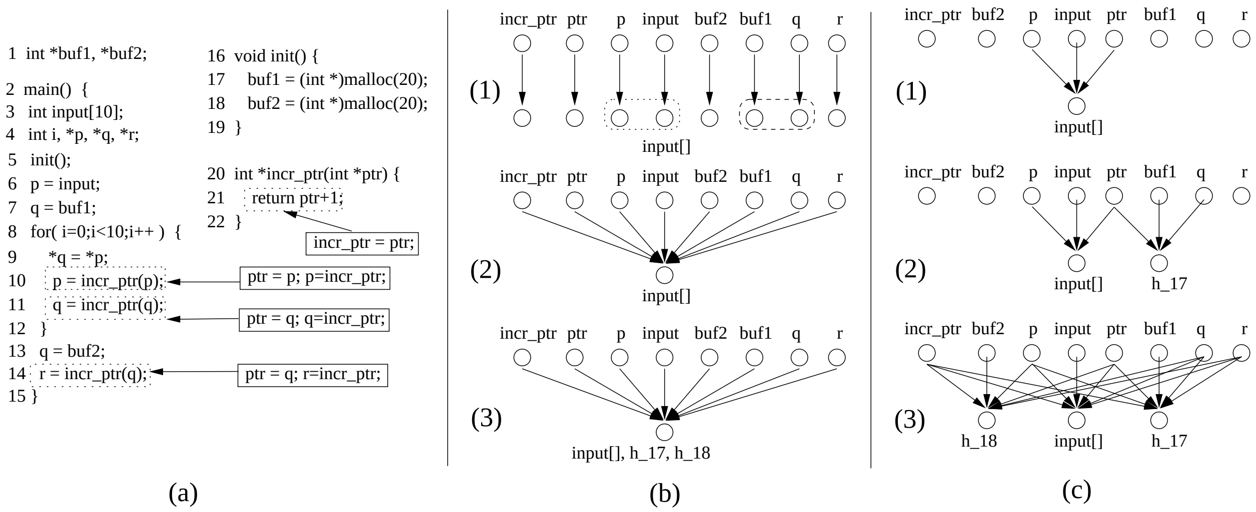

Figure 1: Example program (a), points-to graph using Steensgaard's algorithm (b), points-to graph using Andersen's algorithm (c).

In the next section, we discuss Steensgaard's and Anderson's algorithms and illustrate them with an example. In Section 3, we present our procedure-specific algorithm, and in Section 4, we present the results of three studies that compare our algorithm with three other algorithms. In Section 5, we describe related work and compare it to our work. Finally, in Section 6, we conclude and discuss some future work.

2 Program-Specific Alias-Analysis Algorithms

Program-specific alias-analysis algorithms process pointer-related assignments in a program in an arbitrary order. These algorithms replace a call statement with a set of assignments that represent the bindings of actual parameters and formal parameters. The algorithms compute *safe* points-to information: for any pointer-related assignment, the set of locations pointed to by the left-hand side is a superset of the set of locations pointed to by the right-hand side.

We can view both Steensgaard's algorithm and Andersen's algorithm as building points-to graphs [15].[2] Vertices in a *points-to graph* represent equivalence classes of *memory locations* (i.e., variables and heap-allocated objects), and edges represent points-to relations among the locations.

Steensgaard's algorithm forces all locations pointed to by a pointer to be in the same equivalence class, and, when it processes a pointer-related assignment, it forces the left-hand and right-hand sides of the assignment to point to the same equivalence class. Because of this processing, when new pointer-related assignments are processed, the points-to graph remains safe at a previously-processed pointer-related assignment, which lets Steensgaard's algorithm safely estimate the alias information by processing each pointer-related assignment only once.

Figure 1(b) shows various stages in the construction of the points-to graph for the example program of Figure 1(a) using Steensgaard's algorithm. The top graph (labeled (1), (b)) shows the points-to graph in its initial stage, where all pointers, except *input*, point to empty equivalence classes. When Steensgaard's algorithm processes statement 6, it merges the equivalence class pointed to by *input* with the equivalence class pointed to by *p*; the merged equivalence class is illustrated by the dotted box. Steensgaard's algorithm

[2]A points to graph is similar to an *alias graph* [3].

processes statement 7 similarly; the merged equivalence class is illustrated by the dashed box. The middle graph (labeled (2),(b)) shows the points-to graph after Steensgaard's algorithm has processed $main()$. The algorithm simulates the bindings of parameters and return values at callsites with a set of assignment statements. The assignments in the solid boxes in Figure 1 represent statements 10, 11, and 14.

The bottom graph (labeled (3),(b)) shows the points-to graph after Steensgaard's algorithm processes the entire program. When the algorithm processes statements 17 and 18, it uses $h\_\langle statement\_number \rangle$ to represent the objects returned by **malloc()**. This graph illustrates that Steensgaard's algorithm can introduce many spurious points-to relations.

Andersen's algorithm uses separate equivalence classes to represent different locations. Andersen's algorithm processes a pointer-related assignment by propagating the points-to set from the right-hand side to the left-hand side. Propagating new locations in this way, however, may cause the alias information at a previously-processed pointer-related assignment to be unsafe. To maintain safety, Andersen's algorithm iterates over previously processed pointer-related assignments until the points-to graph provides a safe alias solution.

Figure 1(c) shows various stages in the construction of the points-to graph using Andersen's algorithm for the example program. The top graph (labeled (1),(c)) shows the points-to graph constructed by Andersen's algorithm after it processes $main()$. When the algorithm processes statement 6, it propagates the points-to set from *input* to p. The algorithm performs no actions on the points-to graph when it process statement 7 because the right-hand side of the statement does not yet point to any memory locations. When the algorithm processes statement 10, like Steensgaard's algorithm, it simulates the bindings of the parameters with the assignments shown in the solid boxes. Therefore, it propagates the points-to set of p to the points-to set of ptr. The middle graph (labeled (2),(c)) shows the points-to graph after Andersen's algorithm processes statement 17. The algorithm puts h_17 into $buf1$'s points-to set, which causes the alias information at statement 7 to be unsafe. To provide a safe solution, Andersen's algorithm then processes statement 7 again, which subsequently requires statements 11 and 14 to be reprocessed. The bottom graph (labeled (3),(c)) shows the complete solution computed by Andersen's algorithm. This graph illustrates that Andersen's algorithm can compute smaller points-to sets than Steensgaard's algorithm for some pointer variables. However, Andersen's algorithm requires more steps than Steensgaard's algorithm and the points-to graph constructed by Andersen's algorithm is more complex than that constructed by Steensgaard's algorithm.

3 A Flow-Insensitive, Context-Sensitive Alias-Analysis Algorithm

Our flow-insensitive, context-sensitive alias-analysis algorithm (FICS), computes separate points-to information for each procedure in a program. In this section, we first present some definitions that we use to discuss our algorithm. We next give an overview of the algorithm. We then discuss the details of the algorithm. Finally, we show the complexity of our algorithm.

3.1 Definitions

In a program, memory locations can be referred to by an *object name* [11], which consists of a variable and a possibly empty sequence of dereferences and field accesses. We say that an object name N_1 is *extended* from another object name N_2 if N_1 can be constructed by applying a possibly empty sequence of dereferences and

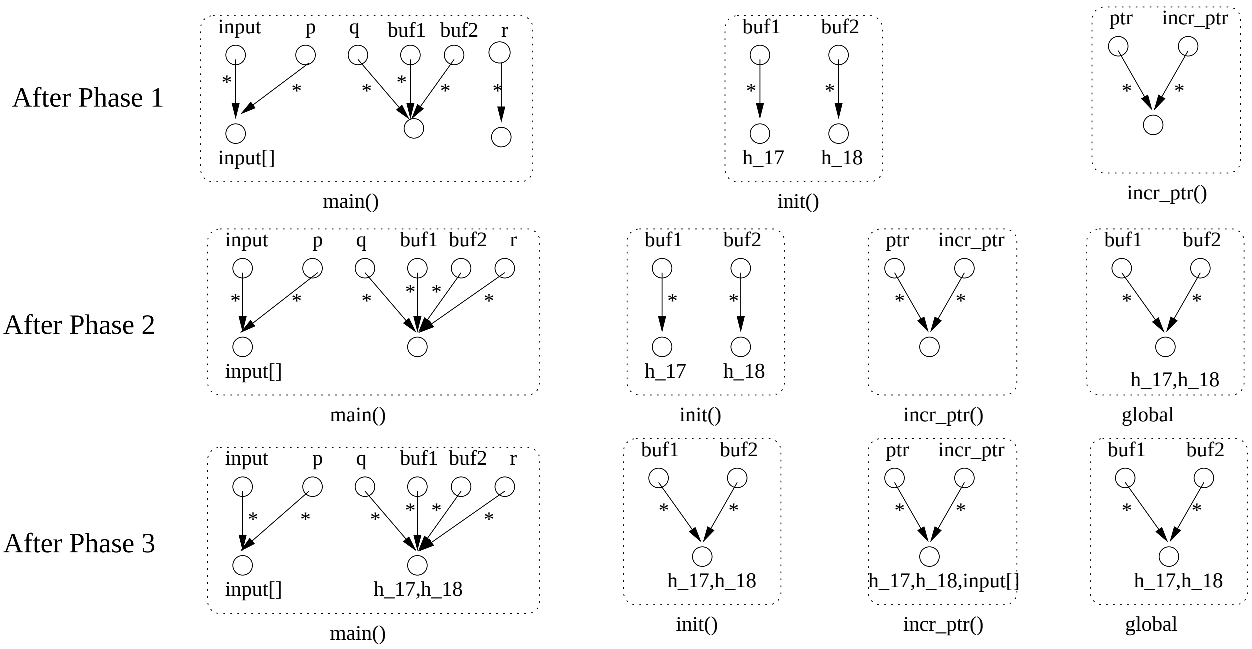

Figure 2: Points-to graphs constructed by FICS algorithm.

field accesses ω to N_2; in this case, we denote N_1 as $\mathcal{E}_\omega\langle N_2\rangle$. If N is a formal parameter and a is the object name of the actual parameter that is bound to N at callsite c, we define a function $\mathcal{A}_c(\mathcal{E}_\omega\langle N\rangle)$ that returns the object name $\mathcal{E}_\omega\langle a\rangle$.

For example, suppose that p is a pointer that points to a **struct** with fields a and b (in the C language). Then $\mathcal{E}_*\langle p\rangle$ is $*p$, $\mathcal{E}_*\langle *p\rangle$ is $**p$, $\mathcal{E}_{*.a}\langle p\rangle$ is $(*p).a$, etc. As another example, if p is a formal parameter to function F, and $*q$ is an actual parameter bound to p at callsite c to F, then $\mathcal{A}_c((*p).a)$ returns $(**q).a$.

We extend a points-to graph to represent structure variables: a vertex representing a structure variable can emit edges to represent field accesses. A *field access* edge, labeled with a field name, connects a vertex representing a structure to a vertex representing a field of the structure. A *points-to* edge, labeled with "*", represents a points-to relation. In a points-to graph constructed by the FICS algorithm, labels are unique among the edges leaving a vertex. Given an object name N, FICS can find an *access path* p in a points-to graph G: first, FICS locates or creates vertex n_0 in G to which N's variable corresponds; then, FICS locates or creates a sequenece of vertices n_i and edges e_i, $1 <= i <= k$, so that $p = n_0, e_1, n_2, ..., e_k, n_k$ is a path in G and labels of the edges in p match the sequence of dereferences and field accesses in N. We denote p as $\mathcal{P}\langle N, G\rangle$. We call n_k the *associated vertex* of N in G, and denote n_k as $\mathcal{V}\langle N, G\rangle$. Note that the set of memory locations associated with $\mathcal{V}\langle N, G\rangle$ is the set of memory locations that are aliased to N.

3.2 Overview

FICS computes procedure-specific alias information for each procedure using points-to graphs. FICS first computes a points-to graph for a procedure P by processing each pointer-related assignment in P using an approach similar to Steensgaard's algorithm. If none of the pointer variables that appear in P is a global variable or a formal parameter, and none of the pointer variables is used as an actual parameter, then the points-to graph safely estimates the aliasing information for P. However, if some pointer variables that appear in P are global variables or formal parameters, or if some pointer variables are used as actual parameters, then the aliasing information computed at this point for these pointer variables is not complete. P's points-to graph must capture the aliases related to these pointer variables that are introduced at the pointer-related assignments in other procedures.

There are three cases in which pointer-related assignments in other procedures can introduce aliases related to a pointer variable appeared in procedure P. In the first case, a pointer-related assignment in another procedure can force $\mathcal{E}_\omega\langle g\rangle$, where g is a global variable, to be aliased to a memory location. Because FICS does not consider the order of the statements, it must assume that such an alias pair holds throughout the program. Thus, FICS must propagate such an alias pair to each procedure. For example, in the program of Figure 1(a), statement 17 forces $*buf1$ to be aliased to a heap-allocated object; this alias pair must be propagated to each procedure in the program. FICS captures this type of alias pairs in P's points-to graph in two steps: (1) it computes a *global* points-to graph to estimate the memory locations that can be aliased to each possible global object name in the program; (2) it updates P's points-to graph using the aliasing information represented by the global points-to graph.

In the second case, an assignment in a procedure called by procedure P can force $\mathcal{E}_{\omega1}\langle f_1\rangle$ to be aliased to $\mathcal{E}_{\omega2}\langle f_2\rangle$, where f_1 is a formal parameter and f_2 is either a formal parameter or a global variable (the return value of a function can be viewed as a special formal parameter). Alias pair $(\mathcal{E}_{\omega1}\langle f_1\rangle,\mathcal{E}_{\omega2}\langle f_2\rangle)$ can be propagated from the called procedure to P and force $\mathcal{A}_c(\mathcal{E}_{\omega1}\langle f_1\rangle)$ to be aliased $\mathcal{A}_c(\mathcal{E}_{\omega2}\langle f_2\rangle)$ at callsite c. For example, in the program of Figure 1(a), statement 21 in function $incr\_ptr()$ forces $*incr\_ptr$, the object name extended from the return value of the function, to be aliased to $*ptr$, an object name extended from the formal parameter. When this alias pair is propagated back to "main()", it forces $*r$ to be aliased to $*q$. FICS captures this type of effects by mapping the alias pairs related to formal parameters to alias pairs related to the actual parameters and updating P's points-to graph to reflect the alias pairs of the actual parameters.

In the third case, an assignment in a procedure that calls P can force a location l to be aliased to $\mathcal{E}_\omega\langle a\rangle$, where a is an actual parameter that is bound to f at a callsite C to P. Alias pair $(\mathcal{E}_\omega\langle a\rangle,\ l)$ is propagated into P and forces $\mathcal{E}_\omega\langle f\rangle$, to be aliased to l. For example, statement 6 forces $(*p, input[])$ to be an alias pair in $main()$ of Figure 1(a). $(*p, input[])$ is propagated into $incr\_ptr()$ at call statement 10 and force $(*ptr, input[])$ to be alias. FICS handles this case by propagating this type of alias pairs from the calling procedure to P and updating P's points-to graph.

FICS is organized into three phases: Phase 1 processes the pointer-related assignments in each procedure and initially builds the points-to graph for the procedure; Phase 2 and Phase 3 handle the three cases discussed above. Phase 2 propagates aliasing information from the called procedures to the calling procedures, and also builds the points-to graph for global variables using the aliasing information available so far for a procedure. Phase 2 processes the procedures in a reverse topological order (bottom-up order) on the strongly-connected components of the call graph. Within a strongly connected components, Phase 2 iterates over the procedures until the points-to graphs for the procedures become stable. Phase 3 propagates aliasing information from the points-to graph for global variables to each procedure. Phase 3 also propagates aliasing information from the calling procedures to the called procedures. Phase 3 processes the procedures in a topological order (top-down order) on the strongly-connected components of the call graph. Similar to Phase 2, Phase 3 iterates over procedures in a component until the points-to graphs for the procedures become stable. Because FICS propagates information from called procedures to calling procedures (Phase

2) before it propagates information from calling procedures to called procedures (Phase 3), it will never propagate information through invalid call/return sequence. Therefore, FICS is context-sensitive.

The graphs on the bottom in Figure 2 show the points-to graphs computed by FICS for the example program of Figure 1. The graphs show that, using FICS, variables can be divided into equivalence classes differently in the points-to graphs of different procedures. For example, in *incr_ptr*(), *h*_17, *h*_18, and *input*[] are in one equivalence class. However, in *main*(), *input*[] is in a different equivalence class than *h*_17 and *h*_18. Because FICS creates separate points-to graphs for *main*(), *init*(), and *incr_ptr*(), it computes a more precise alias solution than Steensgaard's algorithm for the example program. The graphs also show that FICS computes a smaller points-to set for p and q than Andersen's algorithm because it considers calling-context. In the solution computed by Andersen's algorithm, p must point to the locations pointed to by *incr_ptr* under any calling-context; in the solution computed by FICS, p only needs to point to the locations pointed to by *incr_ptr* when *incr_ptr*() is invoked at statement 10. Under such a calling context, *incr_ptr* points only to *input*[].

3.3 Algorithm Description

Figure 3 shows FICS, which inputs the program to be analyzed, and outputs a list of points-to graphs, one for each procedure and one for the global variables.

Phase 1: Create Points-To Graphs for Individual Procedures.

In the first phase (lines 1-7), FICS processes the pointer-related assignments in each procedure P_i in program $\mathcal{P}$ to compute the points-to graph G_{P_i}. For each pointer-related assignment $lhs = rhs$, FICS first finds or creates $v_1{=}\mathcal{V}\langle\mathcal{E}_*\langle lhs\rangle, G_{P_i}\rangle$ and $v_2{=}\mathcal{V}\langle\mathcal{E}_*\langle rhs\rangle, G_{P_i}\rangle$. Then, the algorithm uses `Merge()`, a variant of the "join" operation in Steensgaard's algorithm, to merge v_1 and v_2 into one vertex. `Merge()` also merges the successors of v_1 and v_2 properly so that the labels are unique among the edges leaving the new vertex. During the processing of the pointer-related assignments, FICS ignores all callsites except those callsites to memory-allocation functions; for such callsites, the algorithm uses $h\_\langle statement\_number\rangle$ to represent the objects returned by these functions. Finally, FICS adds P_i to W_1 and to W_2, and adds G_{P_i} to $\mathcal{L}$.

The points-to graphs on the top of Figure 2 are constructed by FICS for *main*() (left), *init*() (middle), and *incr_ptr*()(right) of the example program in the first phase. Note that the points-to relations introduced by *init*(), such as the points-to relation between $buf1$ and *h*_17, are not yet represented in *main*()'s points-to graph. In the following two phases, FICS gathers alias information from both callees and callers of a procedure to further build the points-to graph of a procedure.

Phase 2: Compute Aliases Introduced at Callsites and Create Global Points-to Graph.

In the second phase (lines 8-20), for each procedure P_i, FICS computes the aliases introduced at P_i's callsites. For each callsite c to procedure P_j in P_i, FICS calls `Bind()` to find alias pairs of $(\mathcal{E}_{\omega 1}\langle f_1\rangle,\mathcal{E}_{\omega 2}\langle f_2\rangle)$, where f_1 and f_2 are P_j's formal pointer parameters, with a depth-first search on G_{P_j}. The search begins at at the vertices associated with P_j's formal parameters of pointer type, looking for possible pairs of $\mathcal{P}\langle\mathcal{E}_{\omega 1}\langle f_1\rangle, G_{P_j}\rangle$ and $\mathcal{P}\langle\mathcal{E}_{\omega 2}\langle f_2\rangle, G_{P_j}\rangle$ that end at the same vertex. This implies that $\mathcal{E}_{\omega 1}\langle f_1\rangle$ is aliased to $\mathcal{E}_{\omega 2}\langle f_2\rangle$. `Bind()` maps this type of alias pair back to P_i and captures the alias pairs in G_{P_i} in two steps: first, `Bind()` finds the access paths p_1 and p_2 that correspond to $\mathcal{A}_c(\mathcal{E}_{\omega 1}\langle f_1\rangle)$ and $\mathcal{A}_c(\mathcal{E}_{\omega 2}\langle f_2\rangle)$, respectively; second, it merges

```
algorithm   FICS
input       𝒫: program to be analyzed
output      ℒ: a list of points-to graphs, one for each procedure, one for global variables
declare     P, P_i: procedures in 𝒫
            G_{P_i}: points-to graphs for P_i
            W_1 : a list of procedures, sorted reverse topologically on the strongly-connected components of the call graph
            W_2 : a list of procedures, sorted topologically on the strongly-connected components of the call graph
begin FICS
/* Phase 1: Create points-to graphs for individual procedures */
  foreach procedure P_i in 𝒫 do
    foreach pointer-related assignment lhs = rhs do
      find or create v_1 for lhs, v_2 for rhs in G_{P_i}
      Merge(G_{P_i}, v_1, v_2)
    endfor
    Add P_i to W_1; Add P_i to W_2; Add G_{P_i} to ℒ
  endfor
/* Phase 2: Compute aliases introduced at callsites; create global points-to graph */
  while W_1 ≠ φ do
    P_i = remove procedure from head of W_1
    foreach callsite C to P_j in P_i do
      Bind(actuals_C, G_{P_i}, formals_{P_j}, G_{P_j})
    endfor
    BindGlobal(globals(G_{P_i}), G_glob, G_{P_i})
    BindLoc(globals(G_{P_i}), G_glob, globals(G_{P_j}), G_{P_i})
    if G_{P_i} is updated then
      foreach P_i's caller P_k do
        if P_k not in W_1 then
          Add P_k to W_1        /* required when recursion is present */
        endif
      endfor
    endif
  endwhile
/* Phase 3: Compute aliases introduced by the calling environment */
  while W_2 ≠ φ do
    P_j = remove procedure from head of W_2
    BindLoc(globals(G_{P_j}), G_{P_j}, globals(G_{P_j}), G_glob)
    foreach callsite C from P_i to P_j do
      BindLoc(formals_{P_j}, G_{P_j}, actuals_C, G_{P_i})
    endfor
    if G_{P_j} is updated then
      foreach P_j's callee P_l do
        if P_l not in W_2 then
          Add P_l to W_2        /* required when recursion is present */
        endif
      endfor
    endif
  endwhile
end FICS
```

Figure 3: `FICS`: Flow-Insensitive, Context-Sensitive alias-analysis algorithm.

the end vertices of p_1 and p_2. For example, FICS calls `Bind()` to process the callsite at statement 14 in Figure 1. `Bind()` finds alias pair ($*ptr$, $*incr\_ptr$) in G_{incr_ptr}. Then, it substitutes *ptr* with q and *incr_ptr* with r, and creates an alias pair ($*q$, $*r$), and merges $\mathcal{V}\langle *q, G_{main}\rangle$ and $\mathcal{V}\langle *r, G_{main}\rangle$.

`Bind()` also searches for $\mathcal{P}\langle \mathcal{E}_{\omega 1}\langle f\rangle, G_{P_j}\rangle$ and $\mathcal{P}\langle \mathcal{E}_{\omega 2}\langle g\rangle, G_{P_j}\rangle$, where f is a formal pointer parameter and g is a global variable, that end at the same vertex. Similarly, `Bind()` finds the access path p corresponding to $\mathcal{A}_c(\mathcal{E}_{\omega 1}\langle f\rangle)$ in G_{P_i} and merges the end vertices of p and $\mathcal{P}\langle \mathcal{E}_{\omega 2}\langle g\rangle, G_{P_i}\rangle$.

In the second phase, FICS also calls `BindGlobal()` to compute the global points-to graph with aliasing information of P_i. `BindGlobal()` finds alias pairs $(\mathcal{E}_{\omega 1}\langle g_1\rangle, \mathcal{E}_{\omega 2}\langle g_2\rangle)$, where g_1 and g_2 are global variables, with a depth-first search in G_{P_i}. The search begins at the associated vertices of global variables in G_{P_i} and looks for pairs of access paths $\mathcal{P}\langle \mathcal{E}_{\omega 1}\langle g_1\rangle, G_{P_i}\rangle$ and $\mathcal{P}\langle \mathcal{E}_{\omega 2}\langle g_2\rangle, G_{P_i}\rangle$ that end at one vertex. `BindGlobal()` then finds $\mathcal{P}\langle \mathcal{E}_{\omega 1}\langle g_1\rangle, G_{glob}\rangle$ and $\mathcal{P}\langle \mathcal{E}_{\omega 2}\langle g_2\rangle, G_{glob}\rangle$ and calls `Merge()` to merge the end vertcies of these two access paths. For example, in the second phase, when FICS processes $main()$, it calls `BindGlobal()` to search G_{main} and finds that $\mathcal{P}\langle *buf1, G_{main}\rangle$ and $\mathcal{P}\langle *buf2, G_{main}\rangle$ end at the same vertex. Thus, FICS merges $\mathcal{V}\langle *buf1, G_{glob}\rangle$ and $\mathcal{V}\langle *buf2, G_{glob}\rangle$.

FICS also computes the memory locations that are aliased to $\mathcal{E}_\omega\langle g\rangle$, where g is a global. If a location loc is in the equivalence class represented by $\mathcal{V}\langle\mathcal{E}_\omega\langle g\rangle, G_{P_i}\rangle$, then $(\mathcal{E}_\omega\langle g\rangle, loc)$ is an alias pair. FICS calls `BindLoc()` to look for $\mathcal{V}\langle\mathcal{E}_\omega\langle g\rangle, G_{P_i}\rangle$ with a depth-first search. Then for each location loc associated with $\mathcal{V}\langle\mathcal{E}_\omega\langle g\rangle, G_{P_i}\rangle$, `BindLoc()` merges $\mathcal{V}\langle loc, G_{glob}\rangle$ with $\mathcal{V}\langle\mathcal{E}_\omega\langle g\rangle, G_{glob}\rangle$ to capture the alias pair $(\mathcal{E}_\omega\langle g\rangle, loc)$ in G_{glob}. For example, when FICS processes $init()$ in this phase, it merges $\mathcal{V}\langle h\_17, G_{glob}\rangle$ with $\mathcal{V}\langle *buf1, G_{glob}\rangle$ because h_17 is associated with $\mathcal{V}\langle *buf1, G_{init}\rangle$. After the second phase, the points-to graph for the global variables is complete.

Phase 3: Compute Aliases Introduced by the Calling Environment.

In the third phase (lines 21-32), FICS computes the sets of locations represented by the vertices in G_{P_j} and finishes the computation of G_{P_j}. FICS first computes the locations for vertices in G_{P_j} from G_{glob}. Let g be a global variable that appears in G_{P_j}. FICS calls `BindLoc()` to look for $\mathcal{V}\langle\mathcal{E}_\omega\langle g\rangle, G_{P_j}\rangle$ with a depth-first search. `BindLoc()` then copies the memory locations from $\mathcal{V}\langle\mathcal{E}_\omega\langle g\rangle, G_{glob}\rangle$ to $\mathcal{V}\langle\mathcal{E}_\omega\langle g\rangle, G_{P_j}\rangle$ For example, in the example of Figure 1, FICS copies h_17 and h_18 from $\mathcal{V}\langle *buf1, G_{glob}\rangle$ to $\mathcal{V}\langle *buf1, G_{main}\rangle$.

FICS also computes the locations for vertices in G_{P_j} from G_{P_i}, given that P_i calls P_j at a callsite C. Suppose a is bound to formal parameter f at C. FICS calls `BindLoc()` to copy the locations from $\mathcal{V}\langle\mathcal{E}_\omega\langle a\rangle, G_{P_i}\rangle$ to $\mathcal{V}\langle\mathcal{E}_\omega\langle f\rangle, G_{P_j}\rangle$ to capture the fact that the aliased locations of $\mathcal{E}_\omega\langle a\rangle$ are also aliased to $\mathcal{E}_\omega\langle f\rangle$. For example, FICS copies $input[]$ from $\mathcal{V}\langle *p, G_{main}\rangle$ to $\mathcal{V}\langle *ptr, G_{incr\_ptr}\rangle$ because $main()$ calls $incr\_ptr()$ at statement 11. After this phase, the set of memory locations represented by each vertex is complete.

3.4 Complexity of the FICS Algorithm

Theoretically, it is possible to construct a program $\mathcal{P}$ that has $O(2^n)$ distinguishable locations [16], where n is the size of $\mathcal{P}$. This makes any alias-analysis algorithm discussed in this paper exponential in time and space to the size of $\mathcal{P}$. In practice, however, we can assume that the total distinguishable locations in $\mathcal{P}$ is $O(n)$. We also assume that a structure in $\mathcal{P}$ has a limited number of fields.

FICS computes a separate points-to graph for each procedure in $\mathcal{P}$. Each vertex in a points-to graph represents at least one location and each vertex in a points-to graph has a limited out-degree. Therefore, the points-to graph of a procedure has $O(n)$ vertices and $O(n)$ edges. The space complexity of FICS is $O(p*n)$, where p is the number of procedures in $\mathcal{P}$. In practice, the points-to graph for a procedure typically is very small. Thus, it is more meaningful to express the complexity in the actual size of the points-to graphs. Let S be the worst-case actual size of the points-to graph computed for a procedure, then the space complexity of FICS is $O(p*S+n)$.

To facilitate a quick merge operation, we represent the set of memory locations that are associated with a vertex and the set of edges that are incident to a vertex with fast find/union structures. These representations allow `Merge()` to merge two vertices with two "union" operations and find an access path for a given object name with a constant number of "find" operations.

`Bind()`, `BindGlobal()`, or `BindLoc()` visit visit each edge in G_{P_i} at most once in each of their invocations. Thus, these functions visit vertices or edges $O(S)$ times in one invocation. `Bind()`, `BindGlobal()`, or `BindLoc()` visit an edge with a "find" operation. `Bind()` or `BindGlobal()` might need to find two access paths and merge two vertices in another points-to graph, which require constant number of "find" operations and several invocations of `Merge()`, when they visit a vertex in G_{P_i}. Thus, `Bind()` or `BindGlobal()`

take $O(S)$ time plus $O(S)$ "find" operations in each invocation, without considering the cost of `Merge()`. `BindLoc()` might copy several locations from G_{P_i} to another graph when it visits a vertex in G_{P_i}. In one invocation, `BindLoc()` copies a memory location appearing in G_{P_i} at most once to another points-to graph. Therefore, `BindLoc()` copies at most $O(S)$ locations from one graph to another in one invocation, which require $O(S)$ "find" operations and several invocations of `Merge()`. Thus, `BindLoc()` takes $O(S)$ time plus $O(S)$ "find" operations, without considering the cost of `Merge()`.

Phase 1 of the algorithm takes $O(n)$ "find" operations to find the vertices for the left-hand and right-hand sides, plus $O(n + p * log(p))$ time to manipulate the worklists, plus the time to call `Merge()`. In the absence of recursion, Phase 2 and Phase 3 need $O(NumOfCall)$ propagations to propagate the information from one points-to graph to another. In the presence of recursion, a single change in G_{P_i} requires one propagation to propagate the information to each of P_i's callers and one propagation to propagate the information to each of P_i's callees. There are two kinds of changes that can be made to a points-to graph: to add a memory location or to merge two vertices. Only $O(S)$ of these changes can be made to a points-to graph. Thus, in the presence of recursion, Phase 2 and Phase 3 need $O(NumOfCall * S)$ propagations. Let N be $NumOfCall$ in the absence of recursion, and N to be $NumOfCall * S$ in the presence of recursion, $O(N)$ propagations require $O(N)$ invocations to `Bind()`, `BindGlobal()`, or `BindLoc()`. Hence, Phase 2 and Phase 3 take $O(N * S)$ time plus the time $O(N * S)$ for find operations, without considering the cost of `Merge()`.

`Merge()` can be invoked on G_{P_i} under a number of situations in the algorithm. However, if we consider merging two vertices to be one merging action, only $O(S)$ merging actions can be applied to G_{P_i} because each merging action will delete one vertex. Thus, the cost for all calls to `Merge()` is $O(p * S)$.

From the above discussion, the time complexity of FICS is dominated by the cost of "find" operations. With fast find/union structure, $O(N * S)$ "find" operations and $O(p * S)$ "union" operations on a fast find union structures with size $O(p*S)$ takes $O((N*S+p*S)\alpha(N*S, p*S))$, where α is the inverse Ackermann function. Therefore, the time complexity for FICS is $O(NumOfCall * S\alpha(NumOfCall * S, p * S) + n + p * log(p))$ in the absence of recursion; the time complexity for FICS is $O(NumOfCall * S * S\alpha(NumOfCall * S, p * S) + n + p * log(p))$ in the presence of recursion. In practice, we can expect $NumOfCall * S$ to be $O(n)$. Thus, we can expect FICS runs in time almost linear to the size of the program in practice.

4 Empirical Studies

To investigate the efficiency and precision of FICS, and we performed several studies in which we compared the algorithm with Steensgaard's algorithm [17], Andersen's algorithm [1], and Landi and Ryder's algorithm [11].[3] We used the PROLANGS Analysis Framework (PAF) [6] to implement, with points-to graphs, FICS, Steensgaard's algorithm, and Andersen's algorithm. We used the implementation of Landi and Ryder's algorithm provided by PAF. None of these implementations handles function pointers or setjump-longjump constructs.

The left-hand side of Table 1 gives information about a subset of the subject programs we used in the studies. To allow the algorithms to capture the aliases introduced by calls to library functions, we created a set of stubs that simulate the effects of these functions on aliases. We did not, however, create stubs for the functions that have no effect on aliases because, in preliminary studies, we observed that, using stubs,

[3] We did not implement Shapiro and Horwitz's algorithm because their paper does not describe the algorithm in sufficient detail. We are in the process, however, of obtaining a copy of their implementation, which will let us perform additional experiments to compare FICS with their algorithm.

Program	*Number of Lines of Code*	*Number of CFG Nodes*	*Number of Procedures*	*Number of Pointer-Related Assignments*	*Time(seconds)* *Steensgaard*	*FICS*	*Andersen*	*LR*
pokerd	1153	956	30	35	0.04	0.15	0.14	31.93
ansitape	1596	1087	37	59	0.06	0.16	0.19	0.54
dixie	2100	1357	52	149	0.1	0.22	0.3	0.92
learn	1600	1596	50	129	0.08	0.2	0.35	1.47
triangle	1931	1763	18	121	0.08	0.18	0.25	8.78
assembler	2510	1993	58	111	0.16	0.48	0.61	5.57
patch	2648	2695	57	214	0.11	0.37	0.32	2.11
simulator	3558	2992	114	83	0.11	0.38	0.34	1.43
flex	6902	3762	93	231	0.14	0.42	0.53	410.28
space	11474	5601	137	732	0.62	1.77	4.64	113.39
bison	7893	6533	134	1170	0.33	0.78	1.27	–
spim	24322	11352	263	496	0.48	2.89	1.96	–
larn	9966	11796	295	642	0.37	1.2	1.2	–
moria	25002	20316	482	785	2.34	3.68	521.82	–
T-W-MC	23922	22167	247	2228	0.83	4.41	73.31	–

Table 1: Subject programs and Time to compute alias solutions.

forces Steensgaard's algorithm to introduce many additional points-to relations. For example, for `dixie`, using stubs for the functions that have no effect on aliases, FICS computes, on average, ThruDeref Mod [11] of 29.45 (see Study 1), whereas not using such stubs, it computes, on average, ThruDeref Mod of 22.10.

4.1 Study 1

Study 1 compares the performance and accuracy of Steensgaard's algorithm, FICS, Andersen's algorithm, and Landi and Ryder's algorithm. For each subject program, we recorded the time required to compute the alias information (Time) and the average number of locations modified through dereference (ThruDeref Mod) [19].

The right-hand side of Table 1 shows the running time of the algorithms on the subject programs.[4] We collected these data by running our system on a Sun Ultra 1 workstation with 128MB of memory. For our subject programs, the flow-insensitive algorithms run significantly faster than Landi and Ryder's algorithm. Also, for small programs, both FICS and Andersen's algorithm have running time close to Steensgaard's algorithm. However, for some programs where Andersen's algorithm takes a large amount of time, FICS still runs in time close to Steensgaard's algorithm. This result suggests that, for large programs, FICS is more efficient in time than Andersen's algorithm.

Figure 4 shows the average size of ThruDeref Mod for the four algorithms. For our subject programs, FICS computes alias solutions of ThruDeref Mod that are close to that computed by Andersen's algorithm. For many programs, Steensgaard's algorithm computes very imprecise alias solution, which might limit its applicability to other data-flow problems. However, the solutions computed by FICS and Andersen's algorithm are very close to that computed by Landi and Ryder's algorithm. This result suggests that, for many data-flow problems, aliases obtained from FICS or Andersen's algorithm might provide sufficient precision. Note that, for `pokerd`, `triangle`, `assembler`, and `space`, Andersen's algorithm provides a smaller alias solution than Landi and Ryder's algorithm. This occurs because Landi and Ryder's algorithm uses a k-limiting technique to handle recursive data structures [11], which exist in all four programs.

[4] Data on Landi and Ryder's algorithm are not available for five programs because the analysis required more than 10 hours, the limit we set for the analysis.

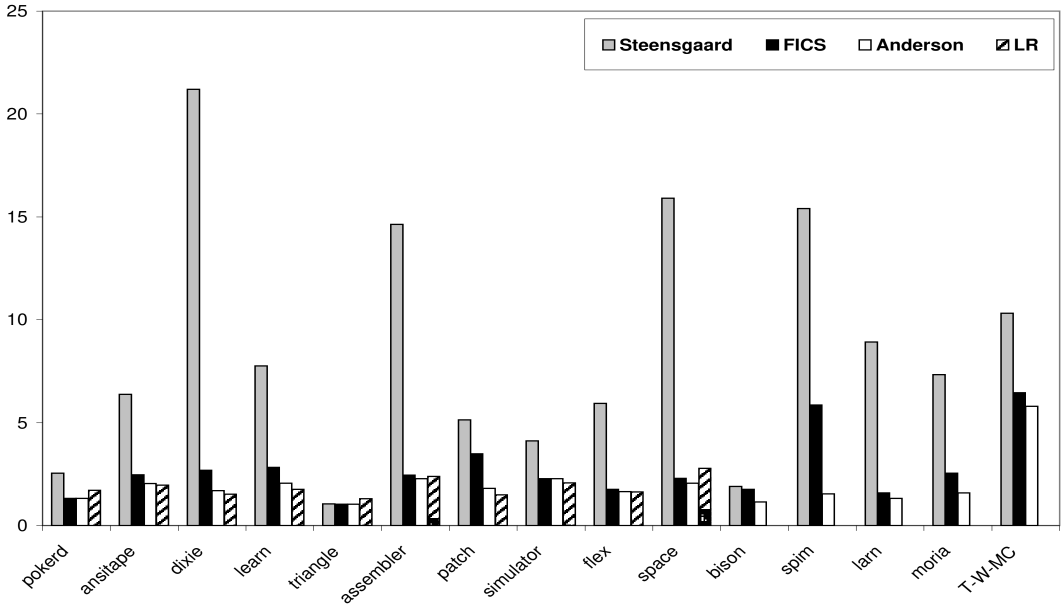

Figure 4: ThruDeref Mod for the subject programs.

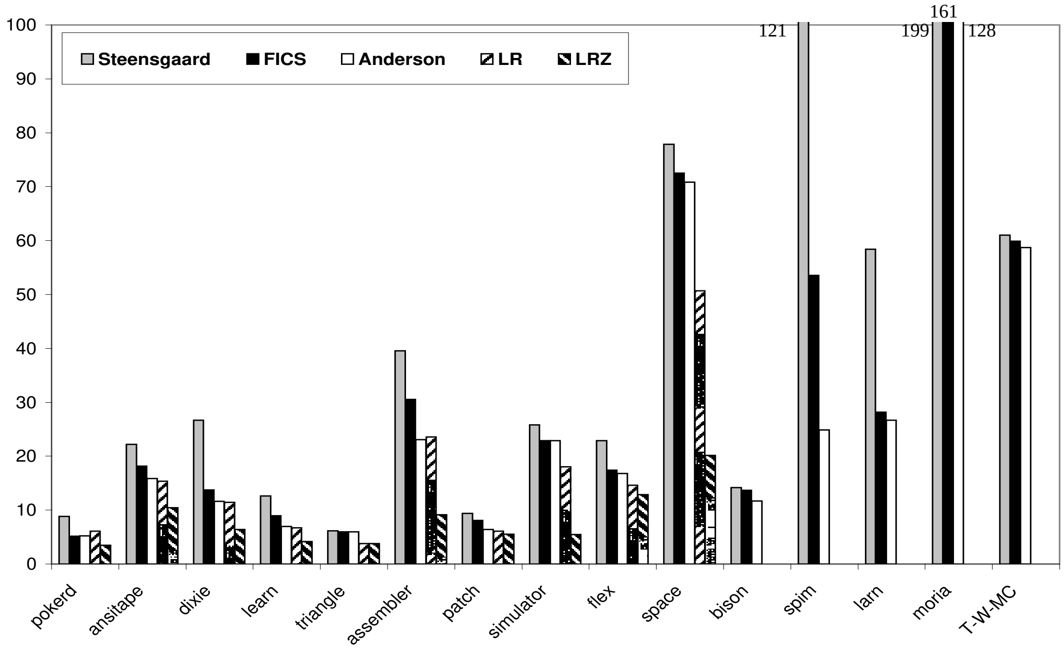

Figure 5: Average size of GMODs for the programs

4.2 Study 2

Study 2 compares the average sizes of the sets of nonlocal variables that are modified by a procedure (GMOD) when those sets are computed with the alias solutions provided by each of the four alias-analysis algorithms. We implemented context-insensitive algorithms, one for each of the four alias-analysis algorithms (Steensgaard's, FICS, Andersen's, Landi and Ryder's), to compute GMOD for each procedure. Each algorithm collects the non-local locations modified by each statement in a procedure P using the alias information computed by the underlying alias-analysis algorithm, and then propagates these locations to P's callers.

We compare the results of these four algorithms with the result computed by Landi, Ryder, and Zhang's context-sensitive side-effect analysis algorithm [12], which computes GMODs using conditional aliasing information computed by Landi and Ryder's algorithm. Figure 5 shows the results of these empirical results. The graph shows that the context-insensitive GMOD set computed with the FICS algorithm is significantly more precise than that computed with Steensgaard's algorithm. For most of the subject programs, FICS gives a result that is very close to that of Andersen's algorithm. The graph also shows that, even with the same aliasing information, context-sensitive GMOD analysis can compute a much more precise solution than context-insensitive GMOD analysis (similar results were reported in [13]).

	Steensgaard		*FICS*		*Anderson*	
program	*size*	*time(s)*	*size*	*time(s)*	*size*	*time(s)*
pokerd	100	17	66	10.62	66	10.72
ansitape	4128	1129.74	851	170.6	431	84.42
dixie	773	104.47	211	45.23	179	39.04
learn	1578	670.47	209	109.32	173	89.59
assembler	924	132.72	243	32.49	243	32.74
simulator	974	186.12	760	119.82	760	120.2
flex	1024	1338.61	602	844.77	562	789.73
space	5931	2451.68	3956	1255.81	3854	1202.46

Table 2: Average number of summary edges (*size*) per call and total time (*time*) to compute all summary edges in a SDG.

4.3 Study 3

Study 3 compares the average numbers of summary edges per call site in the *system dependence graph* [10] computed with aliasing information provided by Steensgaard's algorithm, FICS, and Andersen's algorithm. The study also compares the time to compute all summary edges in a system-dependence graph with aliasing information provided by the three aliasing analysis algorithm. The summary edges are computed by slicing through each procedure with respect to each memory location in the GMOD set of the procedure. Thus, the time required to compute the summary edges might differ from the time required to compute the summary edges using other methods. However, this approach gives us a fair way to compare the costs to compute summary edges with aliasing information at different precision levels.

Table 2 shows the results of this study. From the table, we can see that an output memory location for a procedure in a program that uses pointer variables can depend on many input memory locations of the procedure. One reason for this dependence on a large number of memory locations is because a global pointer or a parameter pointer can point to a large set of memory locations that are not local to the procedure. Another reason for this dependence is that, in the presence of pointer variables, an assignment to a dereference of a pointer cannot kill any definition of memory locations if the dereference accesses more than one memory location. From the table, we can also see that the aliasing information computed by Steensgaard's algorithm can greatly increase the average number of summary edges, which will greatly increase the size of the system-dependence graph for a program. The table also shows that the aliasing information computed by Steensgaard's algorithm can greatly increase the time to compute the summary edges for a program. These results suggest that aliasing information computed by Steensgaard's algorithm can be too imprecise to be used with whole-program analysis tools, such as a system-dependence-graph-based slicer.

5 Related Work

Many data-flow analysis algorithms (e.g., [8, 10]), including FICS, use a two-phase interprocedural analysis framework: in the first phase, information is propagated from the called procedures to the calling procedures, and when a call statement is encoutered, summaries about the called procedure are used to avoid propagating information into the called procedure; in the second phase, information is propagated from the calling procedures to the called procedures. Recently, Chatterjee et al. [4] proposed a method of using unknown initial values for parameters and global variables so that the summaries about a procedure can be computed for flow-sensitive aliasing analysis.[5] Then, they use the two-phase interprocedural analysis framework to compute flow- and context-sensitive aliasing information. Although their algorithm can improve the worst case complexity over Landi and Ryder's algorithm [11] while computing aliasing information with the same precision, it is still too costly in practice. Furthermore, they did not report how their algorithm compares with Landi and Ryder's algorithm in practice.

There have been a number of attempts to design algorithms to compute aliasing information with efficiency close to Steensgaard's algorithm and with precision close to Andersen's algorithm. Shapiro and Horwitz [15] propose a method that divides the program variables into k categories, and allows only variables belonging to the same category to be in an equivalence class. Thus, similar to FICS, this method computes smaller equivalence classes, and provides a smaller points-to set for each pointer variable, than Steensgaard's algorithm. FICS differs from this method, however, in that it uses an independent set of equivalence classes for each procedure. Thus, FICS can benefit from the fact that a procedure references only a small set of program variables. FICS also differs from this method in that FICS is context-sensitive (information is not propagated through invalid call/return sequences). Finally, FICS differs from Shapiro and Horwitz's algorithm in that FICS can handle the fields of structures, whereas in their algorithm, assignments to a field of a structure are treated as assignments to the entire structure. Because of this last difference, it is difficult to compare our experimental results with theirs. However, from the experimental results reported in Reference [15], it seems that, on average, FICS computes alias information that are closer to Andersen's in precision than their algorithm.

Several other researchers consider calling context with a flow-insensitive algorithm. Burke et al. [3] propose an algorithm that interleaves flow-insensitive intraprocedural pointer analysis and context-sensitive interprocedural pointer analysis to compute alias information. This algorithm differs from FICS in that, intraprocedurally, it uses a method similar to Andersen's to process a procedure, which requires iterations over the pointer-related assignments. Interprocedurally, their algorithm may require many iterations over the program's call graph, even for a program without recursion ($O(A_{stmt} * P)$ – A_{stmt} is the maximum number of aliases that hold at any statement, and P is the number of procedures).

Andersen [1] proposes a technique that uses constraints to compute the alias information for variants of a procedure under different calling contexts. This technique uses procedure cloning and represents variants of a variable in different procedure clones with a vector. Thus, in the worst case, it suffers from exponential growth in time and space. Furthermore, no empirical results have reported on its expense in practice.

There have been many works on comparing the performance and precision of flow-insensitive and flow-sensitive aliasing analysis algorithms. Zhang et al. [19, 20] compare a flow-insensitive aliasing analysis algo-

[5] Harrold and Rothermel used a similar approach in [7].

rithm (a variant of Steensgaard's algorithm), an algorithm that combines flow-sensitive and flow-insensitive aliasing analysis, and Landi and Ryder's algorithm. Their result agrees with ours: average number of locations modified or referenced through dereference computed with Steensgaard's algorithm can be much larger than that computed with Landi and Ryder's algorithm. However, their result does not show how the combined algorithm compares with Andersen's algorithm, which also gives a result that is close to Landi and Ryder's algorithm in the average number of locations modified or referenced through dereference. Hind and Pioli [9] compare their flow-insensitive aliasing analysis algorithm [3] with their flow-sensitive aliasing analysis algorithm. Their result shows that flow-insensitive algorithm runs faster than flow-sensitive algorithm and the precisions are close to each other. However, their result also show that their flow-insensitive algorithm does not scale up to large program in time. It is not known that how their flow-insensitive algorithm compare with Andersen's algorithm in precision. Shapiro and Horwitz [14] study how the aliasing information provided by Steensgaard's algorithm, Shapiro's algorithm and Andersen's algorithm can affect other data-flow analyses. Their results also agree with ours: Steensgaard's algorithm can compute much more spurious data-flow information than Andersen's algorithm. Their results also show that data-flow information computed with Shapiro's algorithm is very close to that computed by Steensgaard's algorithm for many data-flow analyses. Because their implementation of these algorithm does not distinguish the fields of a strcture, it is hard to justify their conclusions about the effect of the precision of aliasing information on different data-flow analyses.

6 Conclusions

We presented a context-sensitive, procedure-specific points-to analysis algorithm that computes a separate points-to graph for each procedure. Our algorithm first processes the pointer-related assignments in each procedure using an approach similar to that used by Steensgaard's algorithm, and then it completes the building of points-to graphs by propagating information among procedures. Our algorithm utilizes the fact that a procedure references to only a small set of variables, thus can compute smaller points-to set for each pointer variable than Steensgaard's algorithm.

We conducted several empirical studies on more than 20 C programs to compare our algorithm with other aliasing analysis algorithms. The empirical results show that our algorithm computes aliasing information that is close to Andersen's algorithm in precision, and is close to Steensgaard's algorithm in running time. The empirical results also show that for large programs where Andersen's algorithm takes a large amount of time, our algorithm can compute the aliasing information in time close to Steensgaard's algorithm. This indicates our algorithm can be scaled up to large programs better than Andersen's algorithm. Our studies also show that for many subject programs, Andersen's algorithm computes aliasing information that is close to Landi and Ryder's algorithm, with much less cost. This indicates that flow-sensitive aliasing analysis may provide sufficient precision for many data-flow analyses.

Our future work includes performing additional empirical studies, especially on large subject programs, to further compare our algorithm with other aliasing analysis algorithms. We will also conduct more studies to see how the imprecision in the aliasing information computed by our algorithm can affect various program analyses.

References

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

[2] D. Atkinson and W. Griswold. Effective whole=program analysis in the presence of pointers. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundation of Software Engineering*, November 1998.

[3] M. Burke, P. Carini, J. D. Choi, and M. Hind. Flow-insensitive interprocedrual alias analysis in the presence of pointers. In *Language and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, pages 234–250, 1994.

[4] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of 26th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1999.

[5] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[6] Programming Languages Research Group. PROLANGS Analysis Framework, 1998.

[7] M. J. Harrold and G Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):107–120, June 1996.

[8] M. J. Harrold and M. L. Soffa. Efficient Computation of Interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[9] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *5th International Static Analysis Symposium*. ACM Press, September 1998.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

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

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

[13] W. A. Landi, B. G. Ryder, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. Technical Report DCS-TR-336, Rutgers University, May 1998.

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

[15] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[16] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proc. of the Int. Conf. on Compiler Construction*, 1996.

[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, 1996.

[18] 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.

[19] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer analysis: A step toward practical analyses. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 81–92, November 1996.

[20] S. Zhang, B. G. Ryder, and W. Landi. Experiments with combined analysis for pointer aliasing. In *Program Analysis for Software Tools and Engineering '98*, pages 11–18, 1998.