Efficient Points-to Analysis for Whole-Program Analysis

Donglin Liang and Mary Jean Harrold

The Ohio State University, Columbus, OH 43210, USA
{dliang,harrold}@cis.ohio-state.edu

Abstract. To function on programs written in languages such as C that make extensive use of pointers, automated software engineering tools require safe alias information. Existing alias-analysis techniques that are sufficiently efficient for analysis on large software systems may provide alias information that is too imprecise for tools that use it: the imprecision of the alias information may (1) reduce the precision of the information provided by the tools and (2) increase the cost of the tools. This paper presents a flow-insensitive, context-sensitive points-to analysis algorithm that computes alias information that is almost as precise as that computed by Andersen's algorithm – the most precise flow- and context-insensitive algorithm – and almost as efficient as Steensgaard's algorithm – the most efficient flow- and context-insensitive algorithm. Our empirical studies show that our algorithm scales to large programs better than Andersen's algorithm and show that flow-insensitive alias analysis algorithms, such as our algorithm and Andersen's algorithm, can compute alias information that is close in precision to that computed by the more expensive flow- and context-sensitive alias analysis algorithms.

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

1 Introduction

Many automated tools have been proposed for use in software engineering. To function on programs written in languages such as C that make extensive use of pointers, these tools require alias information that determines the sets of memory locations accessed by dereferences of pointer variables. Atkinson and Griswold [2] discuss issues that must be considered when integrating alias information into whole-program analysis tools. They argue that, to effectively apply the tools to large programs, the alias-analysis algorithms must be fast. Thus, they propose an approach that uses Steensgaard's algorithm [16], a flow- and context-insensitive alias-analysis algorithm[1] that runs in near-linear time, to provide alias information for such tools. However, experiments show that, in many cases, Steensgaard's algorithm computes very imprecise alias information [13, 18]. This imprecision can adversely impact the performance of whole-program analysis.

[1] A *flow-sensitive* algorithm considers the order of statements in a program; a *flow-insensitive* algorithm does not. A *context-sensitive* algorithm considers the legal call/return sequences of procedures in a program; a *context-insensitive* algorithm does not.

Whole-program analysis can be affected by imprecise alias information in two ways. First, imprecise alias information can decrease the precision of the information provided by the whole-program analysis. Our preliminary experiments show that the sizes of slices computed using alias information provided by Steensgaard's algorithm can be almost ten percent larger than the sizes of slices computed using more precise alias information provided by Landi and Ryder's algorithm [11], a flow-sensitive, context-sensitive alias-analysis algorithm. Second, imprecise alias information can greatly increase the cost of whole-program analysis. Our empirical studies show that it can take a slicer five times longer to compute a slice using alias information provided by Steensgaard's algorithm than to compute the slice using alias information provided by Landi and Ryder's algorithm; similar results are reported in [13]. These results indicate that the extra time required to perform whole-program analysis with the less precise alias information might exceed the time saved in alias analysis with Steensgaard's algorithm.

One way to improve the efficiency of whole-program analysis tools is to use more precise alias information. The most precise alias information is provided by flow-sensitive, context-sensitive algorithms (e.g., [5, 11, 17]). The potentially large number of iterations required by these algorithms, however, makes them costly in both time and space. Thus, they are too expensive to be applicable to large programs. Andersen's algorithm [1], another flow-insensitive, context-insensitive alias-analysis algorithm, provides more precise alias information than Steensgaard's algorithm with less cost than flow-sensitive, context-sensitive algorithms. This algorithm, however, may require iteration among pointer-related assignments[2] ($O(n^3)$ 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.

Our approach to providing alias information that is sufficiently precise for use in whole-program analysis, while maintaining efficiency, is to incorporate calling-context into a flow-insensitive alias-analysis algorithm to compute, for each procedure, the alias information that holds at all statements in that procedure. Our algorithm has three phases. In the first phase, the algorithm uses an approach similar to Steensgaard's, to process pointer-related assignments and to compute alias information for each procedure in a program. In the second phase, the algorithm uses a bottom-up approach to propagate alias information from the called procedures (callees) to the calling procedures (callers). Finally, in the third phase, the algorithm uses a top-down approach to propagate alias information from callers to callees.[3]

This paper presents our alias-analysis algorithm. The main benefit of our algorithm is that it efficiently computes an alias solution with high precision. Like Steensgaard's algorithm, our algorithm efficiently provides safe alias infor-

[2] A *pointer-related assignment* is a statement that can change the value of a pointer variable.

[3] Future work includes extending our algorithm to handle function pointers using an approach similar to that discussed in Reference [2].

mation 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, our algorithm computes alias sets that are much smaller than those computed by Steensgaard's algorithm, and provides alias information that is almost as precise as that computed by Andersen's algorithm. 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.

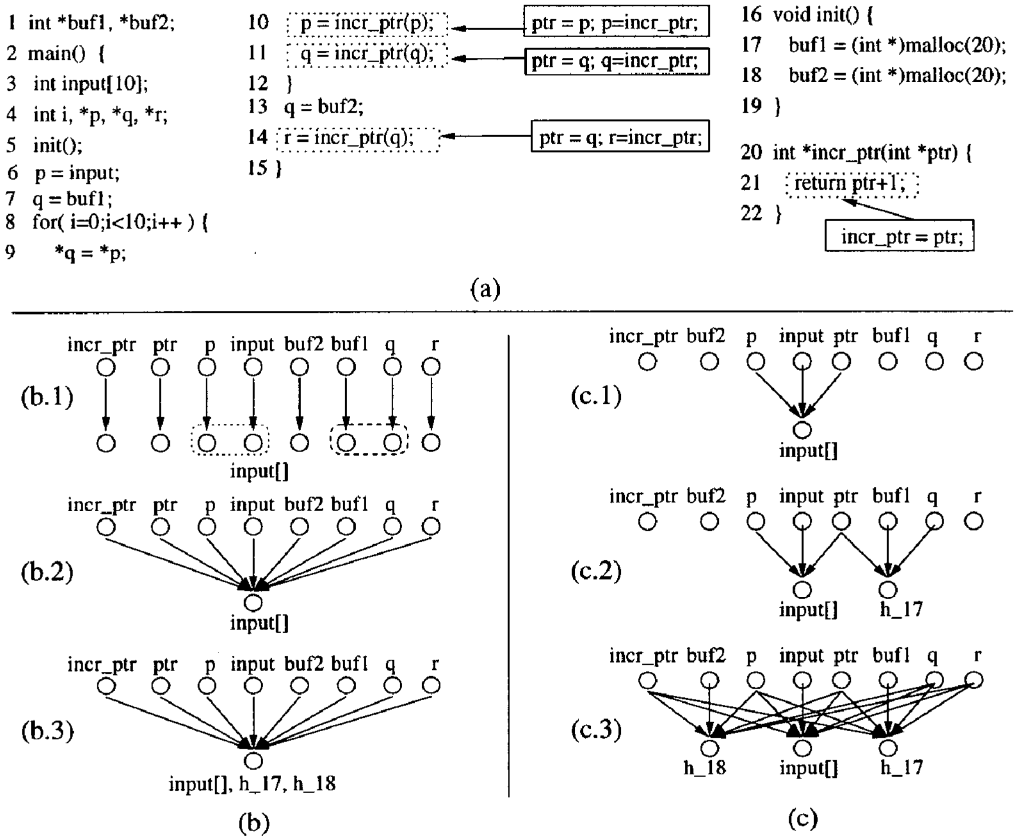
This paper also presents a set of empirical studies in which we investigate (a) the efficiency and precision of three flow-insensitive algorithms — our algorithm, Steensgaard's algorithm, Andersen's algorithm — and Landi and Ryder's flow-sensitive algorithm [11], and (b) the impact of the alias information provided by these four algorithms on whole-program analysis. These studies show a number of interesting results:

- For the programs we studied, our algorithm and Andersen's algorithm can compute a solution that is close in precision to that computed by a flow- and context-sensitive algorithm.
- For programs where Andersen's algorithm requires a large amount of time, our algorithm can compute the alias information in time close to Steensgaard's algorithm; thus, it may scale up to large programs better than Andersen's algorithm.
- The alias information provided by our algorithm, Andersen's algorithm, and Landi and Ryder's algorithm can greatly reduce the cost of constructing system-dependence graphs and of performing data-flow based slicing.
- Our algorithm is almost as effective as Andersen's algorithm and Landi and Ryder's algorithm in improving the performance of constructing system-dependence graphs and of performing data-flow based slicing.

These results indicate that our algorithm can provide sufficiently precise alias information for whole-program analysis in an efficient way. Thus, it may be the most effective algorithm, among the four, for supporting whole-program analysis on large programs.

2 Flow-Insensitive and Context-Insensitive Alias-Analysis Algorithms

Flow-insensitive, context-insensitive alias-analysis algorithms compute alias information that holds at every program point. These algorithms process pointer-related assignments in a program in an arbitrary order and replace a call statement with a set of assignments that represent the bindings of actual parameters and formal parameters. The algorithms compute *safe* alias information (points-to relations): 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.



```
1 int *buf1, *buf2;
2 main() {
3  int input[10];
4  int i, *p, *q, *r;
5  init();
6  p = input;
7  q = buf1;
8  for( i=0;i<10;i++ ) {
9     *q = *p;
10    p = incr_ptr(p);
11    q = incr_ptr(q);
12 }
13 q = buf2;
14 r = incr_ptr(q);
15 }

16 void init() {
17   buf1 = (int *)malloc(20);
18   buf2 = (int *)malloc(20);
19 }

20 int *incr_ptr(int *ptr) {
21   return ptr+1;
22 }
```

ptr = p; p=incr_ptr;
ptr = q; q=incr_ptr;
ptr = q; r=incr_ptr;
incr_ptr = ptr;
(a)
(b.1)
incr_ptr ptr p input buf2 buf1 q r
input[]
(b.2)
incr_ptr ptr p input buf2 buf1 q r
input[]
(b.3)
incr_ptr ptr p input buf2 buf1 q r
input[], h_17, h_18
(b)
(c.1)
incr_ptr buf2 p input ptr buf1 q r
input[]
(c.2)
incr_ptr buf2 p input ptr buf1 q r
input[]
h_17
(c.3)
incr_ptr buf2 p input ptr buf1 q r
h_18
input[]
h_17
(c)

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

We can view both Steensgaard's algorithm and Andersen's algorithm as building points-to graphs [14].[4] 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. Using this method, when new pointer-related assignments are processed, the points-to graph remains safe at a previously-processed pointer-related assignment. This method 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 (b.1))) 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

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

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

To represent the objects returned by **malloc()**, when Steensgaard's algorithm processes statements 17 and 18, it uses *h_⟨statement_number⟩*. The bottom graph (labeled (b.3)) shows the points-to graph after Steensgaard's algorithm processes the entire program. This graph illustrates that Steensgaard's algorithm can introduce many spurious points-to relations.

Andersen's algorithm uses a vertex to represent one memory location. This algorithm processes a pointer-related assignment by adding edges to force the left-hand side to point to the locations in the *points-to set* of the right-hand side. For example, when the algorithm processes statement 6, it adds an edge to force p to point to *input*[]. Adding edges in this way, however, may cause the alias information at a previously-processed pointer-related assignment S to be unsafe — that is, the points-to set of S's left-hand side is not a superset of the points-to set of S's right-hand side. To provide a safe solution, 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 (c.1)) shows the points-to graph constructed by Andersen's algorithm after it processes *main*(). When the algorithm processes statements 10, 11, and 14, it simulates the bindings of the parameters using the assignments shown in the solid boxes. The middle graph (labeled (c.2)) shows the points-to graph after Andersen's algorithm processes statement 17. The algorithm forces *h_*17 to point to *buf*1, which causes the alias information to be unsafe at statement 7. To provide a safe solution, Andersen's algorithm processes statement 7 again, which subsequently requires statements 11 and 14 to be reprocessed. The bottom graph (labeled (c.3)) shows the complete solution. 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.

3 A Flow-Insensitive, Context-Sensitive Points-To Analysis Algorithm

Our flow-insensitive, context-sensitive points-to analysis algorithm (FICS) computes separate alias 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 and then discuss the details of the algorithm.

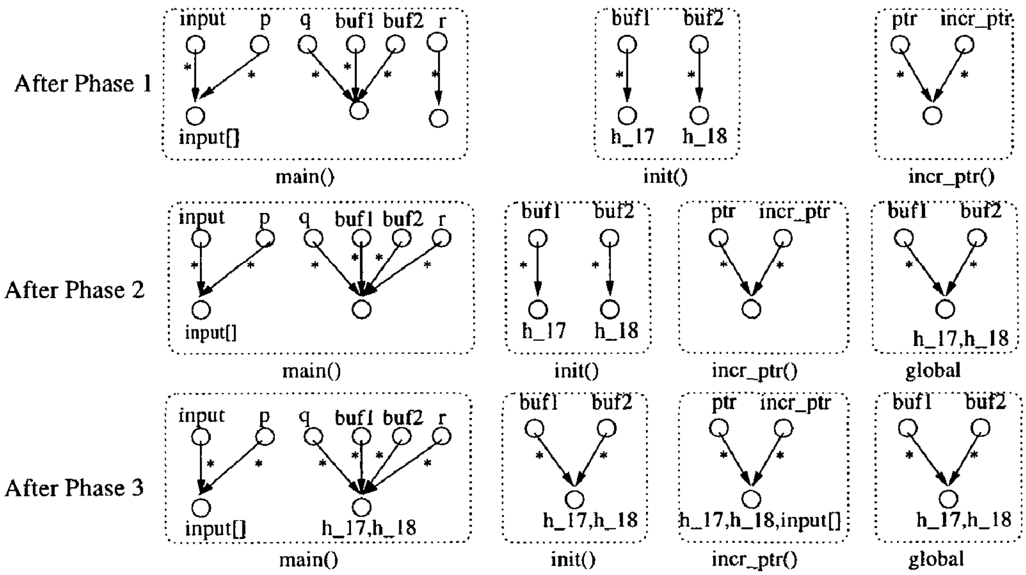


Fig. 2. Points-to graphs constructed by FICS algorithm.

3.1 Definitions

We refer to a memory location in a program 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 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 call site c, we define a function $\mathcal{A}_c(\mathcal{E}_\omega\langle N\rangle)$ that returns object name $\mathcal{E}_\omega\langle a\rangle$. If N is a global, $\mathcal{A}_c(\mathcal{E}_\omega\langle N\rangle)$ returns $\mathcal{E}_\omega\langle N\rangle$.

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

We extend points-to graphs to represent structure variables. 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 such a points-to graph, labels are unique among the edges leaving a vertex. Given an object name N, FICS can find an *access path* $\mathcal{P}\langle N, G\rangle$ 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 sequence of vertices n_i and edges e_i, $1 <= i <= k$, so that $\mathcal{P}\langle N, G\rangle = n_0, e_1, n_2, ..., e_k, n_k$ is a path in G and labels of the edges in p match the sequence of dereferences and field accesses in N. We refer to n_k, the *end vertex* of $\mathcal{P}\langle N, G\rangle$, as 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 separate alias information for each procedure using points-to graphs. FICS first computes a points-to graph G_P 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 appears in P is a global variable or a formal parameter, and none of the pointer variables is used as an actual parameter, then G_P safely estimates the alias 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 pointer-related assignments in other procedures can also introduce aliases related to these variables; G_P must be further processed to capture these aliases.

There are three cases in which pointer-related assignments in other procedures can introduce aliases related to a pointer variable that appear in P. In the first case, a pointer-related assignment in another procedure forces $\mathcal{E}_\omega\langle g\rangle$, where g is a global variable that appears in P, 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 consider such an alias pair in P. For example, in Figure 1(a), statement 17 forces $*buf1$ to be aliased to h_17; this alias pair must be propagated to $main()$ because $main()$ uses $buf1$. FICS captures this type of alias pair in G_P in two steps: (1) it computes a *global* points-to graph, G_{glob}, to estimate the memory locations that are aliased to each possible global object name in the program; (2) it updates G_P using the alias information represented by G_{glob}.

In the second case, an assignment in a procedure called by P forces $\mathcal{E}_{\omega 1}\langle f_1\rangle$ to be aliased to $\mathcal{E}_{\omega 2}\langle f_2\rangle$, 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 is viewed as a formal parameter). Alias pair $(\mathcal{E}_{\omega 1}\langle f_1\rangle,\mathcal{E}_{\omega 2}\langle f_2\rangle)$ can be propagated from the called procedure to P and can force $\mathcal{A}_c(\mathcal{E}_{\omega 1}\langle f_1\rangle)$ to be aliased to $\mathcal{A}_c(\mathcal{E}_{\omega 2}\langle f_2\rangle)$ at call site c. For example, in Figure 1(a), statement 21 in function $incr\_ptr()$ forces $*incr\_ptr$ to be aliased to $*ptr$. When this alias pair is propagated back to $main()$, it forces $*r$ to be aliased to $*q$. FICS maps the alias pairs related to the formal parameters to the alias pairs related to the actual parameters and updates G_P with the alias pairs of the actual parameters.

In the third case, an assignment in a procedure that calls P forces a location l to be aliased to $\mathcal{E}_\omega\langle a\rangle$, where a is an actual parameter bound to f at a call site 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 statement 10, and forces $(*ptr, input[])$ to be an alias pair. FICS propagates this type of alias pairs from the calling procedure to P and updates G_P.

FICS has 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 alias information from the called procedures to the calling procedures, and also builds the points-to graph for the global variables using the alias information available so far for a procedure. Phase 2 processes the procedures in a reverse topological (bottom-up) order on the strongly-connected components of the call graph. Within a strongly-connected component, Phase 2 iterates over the procedures until the points-to graphs for the procedures stabilize. Phase 3 propagates alias information from the points-to graph for global variables to each procedure. Phase 3 also propagates alias information from the calling procedures to the called procedures. Phase 3 processes the procedures in a topological (top-down) order on the strongly-connected components of the call graph. Phase 3 iterates over procedures in a component until the points-to graphs for the procedures stabilize. 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 sequences. Therefore, FICS is context-sensitive.

The bottom graphs in Figure 2 depict 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* points only 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 $\mathcal{P}$, the program to be analyzed, and outputs $\mathcal{L}$, 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 $\mathcal{P}$ to compute the points-to graph G_{P_i}. FICS first finds or creates $v_1{=}\mathcal{V}\langle\mathcal{E}_*\langle lhs\rangle, G_{P_i}\rangle$ and $v_2{=}\mathcal{V}\langle\mathcal{E}_*\langle rhs\rangle, G_{P_i}\rangle$ for each pointer-related assignment $lhs = rhs$. 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. In this phase, FICS ignores all call sites except those call sites to memory-allocation functions; for such call sites, the algorithm uses *h*_⟨*statement_number*⟩ 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 $\mathcal{L}$.

```
algorithm FICS
input      $\mathcal{P}$: program to be analyzed
output     $\mathcal{L}$: a list of points-to graphs, one for each procedure, one for global variables
declare    $P_i$, $P_j$, $P_k$, $P_l$: procedures in $\mathcal{P}$
           $G_{P_i}$: 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
 1.  foreach procedure $P_i$ in $\mathcal{P}$ do    /*phase 1 */
 2.    foreach pointer-related assignment $lhs = rhs$ do
 3.      find or create $v_1$ for $lhs$, $v_2$ for $rhs$ in $G_{P_i}$
 4.      Merge($G_{P_i}$, $v_1$, $v_2$)
 5.    endfor
 6.    Add $P_i$ to $W_1$; Add $P_i$ to $W_2$; Add $G_{P_i}$ to $\mathcal{L}$
 7.  endfor
 8.  while $W_1 \neq \phi$ do              /*phase 2 */
 9.    $P_i$ = remove procedure from head of $W_1$
10.    foreach call site $c$ to $P_j$ in $P_i$ do
11.      Bind($actuals_c$,$G_{P_i}$, $formals_{P_j}$,$G_{P_j}$)
12.    endfor
13.    BindGlobal($globals(G_{P_i})$,$G_{glob}$,$G_{P_i}$)
14.    BindLoc($globals(G_{P_i})$, $G_{glob}$, $globals(G_{P_j})$, $G_{P_i}$)
15.    if $G_{P_i}$ is updated then
16.      foreach $P_i$'s caller $P_k$ do
17.        if $P_k$ not in $W_1$ then Add $P_k$ to $W_1$ endif
18.      endfor
19.    endif
20.  endwhile
21.  while $W_2 \neq \phi$ do              /*phase 3 */
22.    $P_j$ =  remove procedure from head of $W_2$
23.    BindLoc($globals(G_{P_j})$, $G_{P_j}$, $globals(G_{P_j})$, $G_{glob}$)
24.    foreach call site $c$ from $P_i$ to $P_j$ do
25.      BindLoc($formals_{P_j}$,$G_{P_j}$, $actuals_c$,$G_{P_i}$)
26.    endfor
27.    if $G_{P_j}$ is updated then
28.      foreach $P_j$'s callee $P_l$ do
29.        if $P_l$ not in $W_2$ then Add $P_l$ to $W_2$ endif
30.      endfor
31.    endif
32.  endwhile
  end FICS
```

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

The points-to graphs on the top of Figure 2 are constructed by FICS, in the first phase, for *main*() (left), *init*() (middle), and *incr_ptr*()(right) of the example program. 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 P_i to further build G_{P_i}.

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 call sites. For each call site 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, using a depth-first search on G_{P_j}. The search begins 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 $\mathcal{E}_{\omega1}\langle f_1\rangle$ is aliased to $\mathcal{E}_{\omega2}\langle f_2\rangle$. Bind() maps this type of alias pair back to P_i and captures the alias pairs in G_{P_i} by merging the end vertices of $\mathcal{P}\langle\mathcal{A}_c(\mathcal{E}_{\omega1}\langle f_1\rangle), G_{P_i}\rangle$ and $\mathcal{P}\langle\mathcal{A}_c(\mathcal{E}_{\omega2}\langle f_2\rangle), G_{P_i}\rangle$ in G_{P_i}. For example, FICS calls Bind() to process the call site 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 $\mathcal{P}\langle\mathcal{E}_{\omega1}\langle f\rangle, G_{P_j}\rangle$ and $\mathcal{P}\langle\mathcal{E}_{\omega2}\langle g\rangle, G_{P_j}\rangle$, where f is a formal pointer parameter and g is a global variable, that end at the same vertex. Similarly, Bind() merges the end vertices of $\mathcal{P}\langle\mathcal{A}_c(\mathcal{E}_{\omega1}\langle f\rangle), G_{P_i}\rangle$ and $\mathcal{P}\langle\mathcal{E}_{\omega2}\langle g\rangle, G_{P_i}\rangle$ in G_{P_i}.

In this phase, FICS also calls BindGlobal() to compute the global points-to graph G_{glob} with the alias information of P_i. BindGlobal() finds alias pairs $(\mathcal{E}_{\omega1}\langle g_1\rangle, \mathcal{E}_{\omega2}\langle g_2\rangle)$, where g_1 and g_2 are global variables, using 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 $\mathcal{P}\langle\mathcal{E}_{\omega1}\langle g_1\rangle, G_{P_i}\rangle$ and $\mathcal{P}\langle\mathcal{E}_{\omega2}\langle g_2\rangle, G_{P_i}\rangle$ that end at one vertex. BindGlobal() then merges the end vertices of $\mathcal{P}\langle\mathcal{E}_{\omega1}\langle g_1\rangle, G_{glob}\rangle$ and $\mathcal{P}\langle\mathcal{E}_{\omega2}\langle g_2\rangle, G_{glob}\rangle$ in G_{glob}. For example, when FICS processes *main*() in this phase, 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 l is in the equivalence class represented by $\mathcal{V}\langle\mathcal{E}_{\omega}\langle g\rangle, G_{P_i}\rangle$, then $(\mathcal{E}_{\omega}\langle g\rangle, l)$ is an alias pair. FICS calls BindLoc() to look for $\mathcal{V}\langle\mathcal{E}_{\omega}\langle g\rangle, G_{P_i}\rangle$ using a depth-first search. For each location l associated with $\mathcal{V}\langle\mathcal{E}_{\omega}\langle g\rangle, G_{P_i}\rangle$, BindLoc() merges $\mathcal{V}\langle l, G_{glob}\rangle$ with $\mathcal{V}\langle\mathcal{E}_{\omega}\langle g\rangle, G_{glob}\rangle$ to capture the alias pair $(\mathcal{E}_{\omega}\langle g\rangle, l)$ 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 this phase, G_{glob} 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 completes 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 $\mathcal{V}\langle\mathcal{E}_{\omega}\langle g\rangle, G_{P_j}\rangle$ using 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, when FICS processes *main*() in the example of Figure 1, it 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 call site 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 p is bound to *ptr* at statement 11. After this phase, the set of memory locations represented by each vertex is complete.

Complexity of the FICS Algorithm.[5] Theoretically, it is possible to construct a program $\mathcal{P}$ that has $O(2^n)$ distinguishable locations [15], 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, the total distinguishable locations in $\mathcal{P}$ is $O(n)$ and a structure in $\mathcal{P}$ typically has a limited number of fields.

Let p be the number of procedures in $\mathcal{P}$ and S be the worst-case actual size of the points-to graph computed for a procedure. The space complexity of FICS is $O(p * S + n)$. In the absence of recursion, each procedure P is processed once at each phase. Thus, `Bind()`, `BindGlobal()`, and `BindLoc()` are invoked $O(NumOfCall + p)$ times. In the presence of recursion, a single change in G_P might require one propagation to each of P's callers and one propagation to each of P's callees. G_P changes $O(S)$ times, thus, `Bind()`, `BindGlobal()` and `BindLoc()` are invoked $O((NumOfCall{+}p){*}S)$ times. When the points-to graph is implemented with fast find/union structure, each invocation of `Bind()`, `BindGlobal()`, and `BindLoc()` requires $O(S)$ "find" operations on a fast find/union structure with size $O(p * S)$. Let N be $NumOfCall + p$ in the absence of recursion and N be $(NumOfCall + p) * S$ in the presence of recursion. The time complexity of FICS is $O((N{*}S{+}p{*}S)\alpha(N{*}S, p{*}S))$, where α is the inverse Ackermann function. In practice, we can expect $NumOfCall{*}S$ to be $O(n)$. Thus, we can expect to run FICS in time almost linear in the size of the program in practice.

4 Empirical Studies

To investigate the efficiency and precision of FICS and the impact on whole-program analysis of alias information of various precision levels, we performed several studies in which we compared the algorithm with Steensgaard's algorithm (ST) [16], Andersen's algorithm (AND) [1], and Landi and Ryder's algorithm (LR) [11]. 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 used in the studies.[6] 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. However, we did not create stubs for the functions that would not introduce aliases at calls to these functions because, in preliminary studies, we observed that using stubs forces Steensgaard's algorithm to introduce many additional points-to relations. For example, for `dixie`,

[5] Details of the complexity analysis for FICS can be found in [12].

[6] `T-W-MC` and `moria` are not used in Studies 2 and 3 because the slicer requires more than 10 hours, the time limit we set for slicing, to collect the data.

Table 1. Subject programs and Time in seconds to compute alias solutions.

Program	*Lines of Code*	*Number of CFG Nodes*	*Number of Procedures*	*Number of PRAs*	*Time(seconds)*			
					ST	*FICS*	*AND*	*LR*
loader	1132	819	32	42	0.05	0.14	0.16	1.45
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
unzip	4075	1892	42	144	0.06	0.16	0.19	1.75
smail	3212	2430	59	378	0.48	0.74	2.8	–
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	–
larn	9966	11796	295	642	0.37	1.2	1.2	–
mpeg_play	17263	11864	135	1782	0.92	3.18	4.92	–
espresso	12864	15351	306	2706	4.21	10.69	957.16	–
moria	25002	20316	482	785	2.34	3.68	521.82	–
T-W-MC	23922	22167	247	2228	0.83	4.41	73.31	–

using stubs for the functions that would not introduce aliases at calls, FICS computes, on average, ThruDeref Mod [18] of 29.45, whereas not using such stubs, it computes, on average, ThruDeref Mod of 22.10 (see Study 1).

Study 1. In study 1, we compare the performance and precision 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) [18].

The right-hand side of Table 1 shows the running time of the algorithms on the subject programs.[7] We collected these data by running our system on a Sun Ultra 1 workstation with 128MB of physical memory and 256 MB virtual memory. The table shows that, for our subject programs, the flow-insensitive algorithms run significantly faster than Landi and Ryder's algorithm. The table also shows that, for small programs, both FICS and Andersen's algorithm have running time close to Steensgaard's algorithm. However, for the large 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 number of ThruDeref Mod for the four algorithms. The graph shows that, for many programs, Steensgaard's algorithm computes very imprecise alias information, which might limit its applicability to other data-flow analyses. The graph also shows that, for our subject programs, FICS computes alias solutions of ThruDeref Mod that are close to that computed by Andersen's algorithm. For `smail` and `espresso`, FICS computes smaller ThruDeref Mod than Andersen's algorithm because these two programs have functions similar to *incr_ptr*() in Figure 1, on which Andersen's algorithm loses precision because it does not consider calling context. The graph further shows that the

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

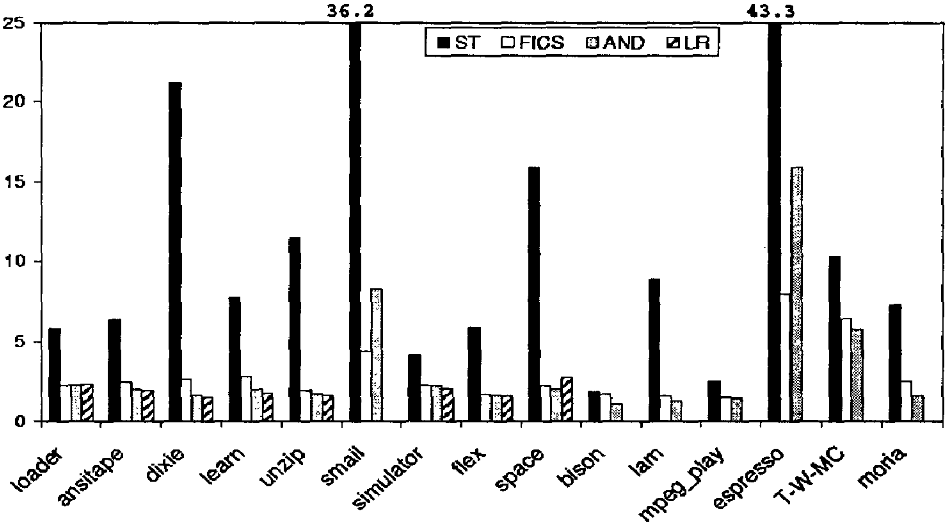


Fig. 4. ThruDeref Mod for the subject programs.

Table 2. Average number of summary edges (*S*) per call and average time (*T*) in seconds to compute the summary edges for a call in a system dependence graph.

program	*Raw Data*								*% of Steensgaard*					
	ST		*FICS*		*AND*		*LR*		*FICS*		*AND*		*LR*	
	S	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*	*T*
loader	465	2.3	195	1.1	195	1.1	199	1.1	41.9	47.4	41.9	47.8	42.8	50.0
ansitape	880	2.6	533	1.7	431	1.2	400	1.2	60.6	66.7	49.0	48.5	45.5	45.8
dixie	821	2.5	314	1.5	227	1.1	206	1.0	38.3	58.4	27.7	42.5	25.2	40.0
learn	1578	7.6	209	1.3	173	1.1	159	1.0	13.3	17.1	11.0	14.1	10.1	12.9
unzip	1979	9.4	738	4.0	687	3.4	402	2.1	37.3	42.9	34.7	36.4	20.3	22.1
smail	3518	15.8	2703	11.3	2260	8.7	–	–	76.8	71.4	64.2	54.9	–	–
simulator	979	2.0	736	1.2	736	1.2	535	1.0	75.1	62.4	75.1	62.6	54.6	50.3
flex	1156	12.1	620	8.0	579	7.5	550	7.4	53.6	66.1	50.1	61.9	47.6	61.3
space	7562	19.4	5639	10.4	5525	10.2	3839	7.5	74.6	53.4	73.1	52.7	50.8	38.5
bison	679	2.6	653	1.6	520	1.1	–	–	96.2	62.4	76.6	43.4	–	–
larn	36726	182.9	9582	38.2	8087	30.9	–	–	26.1	20.9	22.0	16.9	–	–
mpeg_play	1306	32.2	946	23.9	940	21.8	–	–	72.4	74.2	72.0	67.7	–	–
espresso	13964	121.5	8540	60.5	10518	82.9	–	–	61.2	49.8	75.3	68.3	–	–

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 using FICS or Andersen's algorithm might provide sufficient precision. Note that, because Landi and Ryder's algorithm uses a k-limiting technique, which collapses the fields of a structure, to handle recursive data structures [11], the points-to set for a pointer p computed by Landi and Ryder's algorithm may contain locations that are not in the points-to set for p computed by the three flow-insensitive algorithms. Thus, Andersen's algorithm provides a smaller alias solution than Landi and Ryder's algorithm for `loader` and `space`.

Study 2. In study 2, we investigate the impact of the alias information provided by the four algorithms on the size and the cost of the construction of one program representation — the system-dependence graph [10].[8] We study the average number of summary edges per call and the cost to compute these summary edges in a system dependence graph. The summary edges are computed by slicing through each procedure with respect to each memory location that can be modified by the procedure using Harrold and Ci's slicer [7]. Thus, the time required to compute the summary edges might differ from the time required to compute the summary edges using other methods (e.g. [10]). Nevertheless, this approach provides a fair way to compare the costs of computing summary edges using alias information of different precision levels.

Table 2 shows the results of this study. We obtained these results on a Sun Ultra 30 workstation with 640MB physical memory and 1GB virtual memory. The table shows that using more precise alias information provided by FICS, Andersen's algorithm, and Landi and Ryder's algorithm can effectively reduce both the average number of summary edges per call and the time to compute the summary edges in the construction of a system-dependence graph.[9] The table further shows that, for our subject programs, using alias information provided by FICS is almost as effective as using alias information provided by Andersen's algorithm. Our algorithm is even more effective than Andersen's algorithm on espresso because our algorithm computes a smaller points-to set for the pointer variables. These results suggest that FICS is preferable to Andersen's algorithm in building system-dependence graphs for large programs because FICS can run significantly faster than Andersen's algorithm on large programs.

Study 3. In study 3, we investigate the impact of the alias information provided by the four alias-analysis algorithms on the sizes of the slices and the cost of computing the slices. We obtained the slices by running Harrold and Ci's slicer [7] on each slicing criterion of interest, without stored reuse information.

Table 3 shows the results of this study. We obtained these results on a Sun Ultra 30 workstation with 640MB physical memory and 1GB virtual memory. The table shows that, for all the subject programs, using more precise alias information than that computed by Steensgaard's algorithm can significantly reduce the time to compute a slice. The table also shows that, for four programs, using more precise alias information can significantly ($> 10\%$) reduce the sizes of the slices. These four programs illustrate exceptions to the conclusion drawn by Shapiro and Horwitz [13] that the sizes of slices are hardly affected by the precision of the alias information. Note that for five of the programs, the slicer computes larger slices using alias information provided by Landi and Ryder's algorithm than using that provided by FICS and Andersen's algorithm because the points-to set computed by Landi and Ryder's algorithm for a pointer p contains memory locations that are not in the points-to set computed by Steensgaard's, FICS, or Andersen's algorithms for p. The table further shows that using alias information

[8] A system-dependence graph can be used to slice a program.; computing summary edges is the most expensive part of constructing such a graph.

[9] Similar results of time were reported in [13] where Steensgaard's, Shapiro's and Andersen's algorithms were compared.

Table 3. Average size of a slice (S) and average time (T) in seconds to compute a slice.

program	*Raw Data*								*% of Steensgaard*					
	ST		*FICS*		*AND*		*LR*		*FICS*		*AND*		*LR*	
	S	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*	*T*
loader†	207	5.3	192	3.4	192	3.3	194	3.5	93.0	64.1	93.0	63.4	93.8	66.5
ansitape†	290	16.6	284	9.6	277	5.3	300	4.9	98.1	58.1	95.7	32.2	103.5	29.7
dixie†	705	25.5	704	8.3	704	5.9	699	5.5	99.9	32.7	99.9	23.1	99.2	21.7
learn†	442	25.4	442	17.6	442	11.4	440	16.8	100.0	69.0	99.9	44.9	99.5	66.0
unzip†	808	37.5	807	13.1	807	10.8	805	9.3	99.9	35.0	99.8	28.9	99.6	24.9
smail†	738	176.5	637	96.1	635	75.4	–	–	86.3	54.5	86.1	42.7	–	–
simulator†	1258	54.8	1087	22.5	1087	22.7	1151	24.2	86.4	41.1	86.4	41.3	91.5	44.2
flex‡	2025	220.2	2019	167.3	2019	153.8	2002	159.8	99.7	76.0	99.7	69.9	98.9	72.6
space‡	2234	1373.9	1936	573.5	1936	569.8	2086	467.3	86.7	41.7	86.7	41.5	93.4	34.0
bison‡	2394	94.9	2394	84.1	2338	41.0	–	–	100.0	88.6	97.7	43.2	–	–
larn‡	6626	3477.3	6602	1075.6	6592	902.4	–	–	99.6	30.9	99.5	26.0	–	–
mpeg_play‡	5708	325.5	3935	134.6	3935	139.5	–	–	68.9	41.3	68.9	42.9	–	–
espresso‡	6297	8332.1	6291	3776.5	6264	5367.1	–	–	99.9	45.3	99.5	64.4	–	–

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

provided by FICS is almost as effective as using alias information provided by Andersen's algorithm in computing slices. This further supports our conclusion that FICS is preferable to Andersen's algorithm in whole-program analysis.

5 Related Work

Many data-flow analysis algorithms (e.g., [9,10]), 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 encountered, 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] use unknown initial values for parameters and global variables so that the summaries about a procedure can be computed for flow-sensitive alias analysis.[10] Then, they use the two-phase interprocedural analysis framework to compute flow- and context-sensitive alias information. Although their algorithm can improve the worst case complexity over Landi and Ryder's algorithm [11] while computing alias information with the same precision, it is still too costly in practice. Furthermore, because no comparison between these two algorithms is reported, it is not known how much Chatterjee et al.'s algorithm outperforms Landi and Ryder's algorithm.

There have been a number of attempts to design algorithms to compute alias information with efficiency close to Steensgaard's algorithm and with precision close to Andersen's algorithm. Shapiro and Horwitz [14] 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

[10] Harrold and Rothermel used a similar approach in [8].

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 [14], it appears that, on average, FICS computes alias information that is closer to Andersen's in precision than their algorithm.

6 Conclusions

We presented a flow-insensitive, context-sensitive points-to analysis algorithm and conducted several empirical studies on more than 20 C programs to compare our algorithm with other alias-analysis algorithms. The empirical results show that, although Steensgaard's algorithm is fast, the alias information computed by this algorithm is too imprecise to be used in whole-program analysis. The empirical results further show that using more precise alias information provided by our algorithm, Andersen's algorithm, and Landi and Ryder's algorithm can effectively improve the precision and reduce the cost of whole-program analysis. However, the empirical results also show that Andersen's algorithm and Landi and Ryder's algorithm could be too costly for analyzing large programs. In contrast, the empirical results show that our algorithm can compute alias information that is almost as precise as that computed by Andersen's algorithm, with running time that is within six times that of Steensgaard's algorithm. Thus, our algorithm may be more effective than the other algorithms in supporting whole-program analysis.

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

7 Acknowledgements

This work was supported in part by grants from Microsoft, Inc. and by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University. We thank the anonymous reviewers who made many helpful suggestions that improved the presentation of the paper.

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*, pages 46–55, 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. http://www.prolangs.rutgers.edu/, Rutgers University, 1998.
7. M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *The 20th International Conference on Software Engineering*, pages 74–83, April 1998.
8. M. J. Harrold and G Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):107–120, June 1996.
9. 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.
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. D. Liang and M. J. Harrold. Context-sensitive, procedure-specific points-to analysis. Technical Report OSU-CISRC-3/99-TR05, The Ohio State University, March 1999.
13. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis 4th International Symposium, SAS '97, Lecture Notes in Computer Science Vol 1302*, pages 16–34, September 1997.
14. 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.
15. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proc. of the Int. Conf. on Compiler Construction*, pages 136–150, 1996.
16. 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.
17. 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.
18. 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.