

Equivalence Analysis: A General Technique to Improve the Efficiency of Data-flow Analyses in the Presence of Pointers

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

Abstract

Existing methods to handle pointer variables during data-flow analyses can make such analyses inefficient both in time and space because the data-flow analyses must store and propagate large sets of data facts that are introduced by dereferences of pointer variable. This paper presents *equivalence analysis*, a general technique to improve the efficiency of data-flow analyses in the presence of pointers. The technique identifies equivalence relations among the memory locations accessed by a procedure and ensures that two equivalent memory locations share the same set of data facts in a procedure and in the procedures that are called by that procedure. Thus, a data-flow analysis needs to compute the data-flow information only for a representative memory location in an equivalence class. The data-flow information for other memory locations in the equivalence class can be derived from that of the representative memory location. Our empirical studies indicate that equivalence analysis may effectively improve the efficiency of many data-flow analyses.

Keywords: Alias analysis, data-flow analysis.

1 Introduction

Data-flow analyses have been widely used to provide information required for tasks such as compiler optimization, program understanding, debugging, testing, and maintenance. To provide safe data-flow information for programs written in languages such as C, these analyses must account for the effects of aliasing introduced by pointer variables.¹ Data-flow analyses use alias-analysis algorithms to compute alias information. Existing alias-analysis algorithms can be classified into two categories: *flow-sensitive* algorithms (e.g., [6, 11]) provide precise alias information at each program point but are expensive; *flow-insensitive* algorithms (e.g., [1, 13, 15, 16]) sacrifice precision for efficiency, and quickly compute less precise alias information that holds throughout a procedure or the entire program.

Many researchers report ways to use alias information in data-flow analyses to ensure safety (e.g., [2, 12, 14]). The method used

¹Aliasing occurs when two different names access the same memory location at a point in a program.

by all these techniques to handle pointer variables in a program can be described as follows: when the data facts, such as definitions and uses, are computed for a statement, a dereference of a pointer variable is replaced with the set of *memory locations* (variables or heap-allocated objects) accessed by the dereference of the pointer variable. For example, to analyze a C program, if $*p$ can access x , y , and z at statement “ $*p=3$;”, then these techniques will replace $*p$ with x , y , and z , create definitions for x , y , and z , and use these definitions in the analysis.

Researchers have studied the effects of the alias information computed by different alias analysis algorithms, with varying levels of precision, on the precision and efficiency of data-flow analyses when the pointer variables are handled using the method discussed in the previous paragraph [12, 14, 19]. The empirical results of these studies show that, when the size of the subject program is large, the side-effects of a statement or a procedure can be greatly increased through dereferences of pointer variables. For example, Zhang, Ryder, and Landi [19] report that, when Landi and Ryder’s algorithm [11], a flow-sensitive alias analysis algorithm, is used to resolve the dereference of pointers, a statement in a C program that contains approximately 13,000 lines of code can modify, on average, 20 memory locations through dereference of pointer variables. These empirical results also show that the side-effects of a statement or a procedure can be further increased if less precise alias information is used. For example, Landi et al. [12] report that, when alias information provided by their algorithm is used to analyze the modification side-effects, a procedure in a C program with 28,735 lines of code can modify, on average, about 13 memory locations. However, if alias information provided by a variant of Steensgaard’s algorithm [16], a fast flow-insensitive alias analysis algorithm, is used, a procedure in this program can modify, on average, 115 memory locations.

These empirical results imply that, without better techniques to handle pointer variables, many data-flow analyses may be too inefficient for use in analyzing programs that use pointer variables, because these techniques may require storage and propagation of large sets of data-flow information. These empirical results also imply that, without better techniques to handle pointer variables, the additional time required to perform data-flow analyses with the less precise alias information might exceed the time saved using fast flow-insensitive alias analyses.

This paper presents *equivalence analysis*, a new technique that may improve the efficiency of many data-flow analyses in the presence of pointer variables. Equivalence analysis is based on the observation that memory locations pointed to by a pointer variable are often analyzed in the same way by the data-flow analyses of a procedure. Equivalence analysis identifies the equivalence relations among the memory locations for each procedure, and uses these equivalence relations to partition the memory locations that

```

1. f1(int *p) {      7.  int g, g1;      12. main() {
2.  *p = g+1;        8.  f2(int *q) {      13.  int x,y,z;
3.  if( g>0 )        9.    *q = g;        14.  f1(&x);
4.  p = &g;          10.   g1++;          15.  f1(&y);
5.  f2(p);           11. }                16.  f2(&z);
6.  *p++;            17. }
}

```

procedure	equivalence classes
f2()	{x, y, z}, {g}, {g1}, {q}
f1()	{x, y}, {z}, {g}, {g1}, {p}
main()	{x}, {y}, {z}, {g}, {g1}

Figure 1: Example Program and equivalence classes for each procedure when dereferences are resolved with Landi and Ryder’s algorithm [11].

are accessed in the procedure into equivalence classes. A data-flow analysis computes data-flow information for only one representative memory location in an equivalence class. The data-flow analysis can derive data-flow information about other memory locations in an equivalence class from that computed for the representative memory location. Therefore, equivalence analysis can be used to improve the efficiency of data-flow analyses.

This paper also presents a set of empirical studies. These studies show that equivalence analysis can effectively reduce the sizes of sets of data facts and data-flow information for a procedure. For example, our studies show that for several subject programs, the data-flow information for more than 60 percent of the nonlocal memory locations modified by a procedure can be derived from the data-flow information of the representative memory locations when the pointer dereferences are resolved using alias information computed by Andersen’s algorithm [1]. The studies indicate that using equivalence analysis may significantly improve the efficiency of many data-flow analyses in the presence of pointer variables.

In the next section, we define the equivalence relations, and present an algorithm to compute the equivalence classes. In Section 3, we discuss the empirical results of the studies that we performed to investigate the efficiency of our algorithm in computing equivalence classes and the effectiveness of using equivalence relations to reduce the amount of data-flow information. Finally, in Section 4, we discuss the related work, and in Section 5, we give a conclusion and discuss the future work.

2 Equivalence Analysis

In this section, we first define equivalence among memory locations and discuss applications of the equivalence relation. We then present our algorithm to compute the equivalence classes of memory locations for each procedure. We also analyze the complexity of the algorithm and discuss extensions to the equivalence analysis.

2.1 Equivalence of Memory Locations

A memory location is accessed through an *object name* [11], which consists of a variable and a sequence of dereferences and field accesses. An object name is *indirect* if the object name consists of at least one dereference; otherwise, the object name is *direct*. Given an object name obj and a statement s , the *accessed set of obj at s* , denoted as $ASet(obj, s)$, is the set of memory locations that may be aliased to obj at s . For example, in the program in Figure 1, if the alias information is provided by Landi and Ryder’s algorithm [11], $ASet(*p, 2) = \{x, y\}$. If l is in $ASet(obj, s)$ where obj is an indirect name, then l is *indirectly accessed* at s .

The mode of access of two memory locations can induce an equivalence relation between these two memory locations that holds in a procedure P . A memory location is always *equivalent* to itself in P . Memory location l_1 is *equivalent* to memory location l_2 in

P if, at any statement s in P or in any procedure directly or indirectly called by P , given an object name obj that is referenced at s , $l_1 \in ASet(obj, s)$ implies $l_2 \in ASet(obj, s)$ and $l_2 \in ASet(obj, s)$ implies $l_1 \in ASet(obj, s)$. For example, in procedure $f2()$ of Figure 1, variables x , y , and z are equivalent: x , y , and z are in the accessed set of $*q$ at statement 9, the only statement in $f2()$ that accesses to any of the three memory locations. Note that, under this definition, if memory location l is accessed directly in P or in any procedure directly or indirectly called by P , then l is equivalent only to itself. For example, g in $f2()$ is equivalent only to itself because it is directly accessed at statement 9.

The efficiency of many intraprocedural data-flow analyses can be improved using equivalence analysis. An intraprocedural data-flow analysis computes the same data-flow information in procedure P for two equivalent memory locations because these equivalent memory locations share the same set of data facts at each statement in P . The data-flow analysis can be improved by first partitioning the memory locations accessed in P into equivalence classes, and then computing the data-flow information only for a representative memory location in each equivalence class. Using the information computed for the representative memory location, the data-flow analysis can then derive data-flow information about other memory locations in an equivalence class. For example, the memory locations referenced by $f2()$ in Figure 1 can be partitioned into three equivalence classes: $\{x, y, z\}$, $\{q\}$, and $\{g\}$. To compute reaching definitions for $f2()$, the analysis computes the reaching definitions only for x , q , and g . The analysis can derive the reaching definitions for y and z from that of x . In $f2()$, x is defined at statement 9, and this definition can reach statements 10 and 11; thus, the analysis concludes that x , y , and z are defined at statement 9 and these definitions can reach statements 10 and 11.

The efficiency of interprocedural data-flow analyses also can be improved by computing and storing information only for the representative memory locations in equivalence classes. Many interprocedural data-flow analyses (e.g., [8, 10]) summarize the results computed by an intraprocedural phase for nonlocal memory locations accessed by a procedure P . In such interprocedural analyses, when a callsite to P is processed, the summary information can be used to compute the effect of P on the callsite. Because memory locations in an equivalence class share the same data facts in P , they share similar summary information. Thus, these types of data-flow analyses can be improved by computing and storing summary information only for the representative memory locations of the equivalence classes. At the callsite to P , a data-flow analysis can derive the summary information for a memory location l from that computed for the representative memory location of the equivalence class that contains l . For example, to compute the set of statements in $f2()$ that can affect y at statement 5 in Figure 1, the reuse-driven slicer [8] can use the set of statements that was computed for x . This approach gives correct results because x and y are in one equivalence class in $f2()$ and share the same set of data facts at each statement in $f2()$.

2.2 Equivalence Analysis Algorithm

Our algorithm computes a set of equivalence classes for each procedure in a program in two phases: an intraprocedural phase and an interprocedural phase.

During the intraprocedural phase, the algorithm partitions the memory locations accessed in each procedure P_i into a set of equivalence classes using object names that are referenced at statements in P_i . The algorithm first initializes, \mathcal{E}_i , the set of equivalence classes for P_i with a single equivalence class that contains all memory locations that are accessed in P_i . The algorithm then updates \mathcal{E}_i by considering each object name obj that is referenced at a statement s in P_i . For any two memory locations l_1 and l_2 , if l_1 and l_2

are in the same equivalence class E in \mathcal{E}_i at this point in the computation and $l_1 \in ASet(obj, s)$, $l_2 \notin ASet(obj, s)$, then l_1 and l_2 are not equivalent and, thus, should be in different equivalence classes. The algorithm updates \mathcal{E}_i by dividing E into E_1 and E_2 , in which $E_1 = E \cap ASet(obj, s)$ and $E_2 = E - ASet(obj, s)$. For example, the set of equivalence classes for $f2()$ in Figure 1 is initialized to contain only $\{x, y, z, g, g1, q\}$. When the algorithm processes $*q$ at statement 9, because $ASet(*q, 9) = \{x, y, z\}$, the algorithm splits $\{x, y, z, g, g1, q\}$ into $\{x, y, z\}$ and $\{g, g1, q\}$. Table 1 illustrates the changes of equivalence classes when the algorithm processes $f2()$ during the intraprocedural phase.

Table 1: Steps to process $f2()$.

Object Name	Access Set	Equivalence Classes
$*p$	$\{x, y, z\}$	$\{x, y, z, g, g1, q\}^\dagger$
g	$\{g\}$	$\{x, y, z\} \{g, g1, q\}^\dagger$
q	$\{q\}$	$\{x, y, z\} \{g\} \{g1, q\}^\dagger$
$g1$	$\{g1\}$	$\{x, y, z\} \{g\} \{q\} \{g1\}^\dagger$

† Implicit equivalence class.

During the interprocedural phase, the algorithm further builds \mathcal{E}_i by considering the equivalence relations among the nonlocal memory locations accessed in each of the procedures, P_j , called by P_i . The algorithm processes each call statement C in P_i that calls P_j and identifies the set of nonlocal memory locations accessed by P_j . The algorithm partitions this set of memory locations into a set of equivalence classes, S_j , according to the equivalence relations computed for P_j up to this point. Each equivalence class E_j in S_j is used to divide an equivalence class E in \mathcal{E}_i into two sets E_1 and E_2 in which $E_1 = E \cap E_j$ and $E_2 = E - E_j$. The algorithm updates \mathcal{E}_i by replacing E with E_1 and E_2 if neither E_1 nor E_2 is empty.

Figure 2 shows the algorithm `EquivAnalysis` that computes equivalence classes for each procedure in a program. To save space, `EquivAnalysis` explicitly stores only the equivalence classes that contain memory locations that have been processed by the algorithm (i.e., the memory locations that have been processed by lines 19–26 in `Update`). For example, initially, all memory locations that are accessed by procedure P_i are in one equivalence class. However, because none of these memory locations has been processed, such an equivalence class is not stored in \mathcal{E}_i . In the rest of this section, we refer to the equivalence class that contains unprocessed memory locations as the *implicit* equivalence class. Table 1 shows the implicit equivalence class at each step when the algorithm processes $f2()$ during the intraprocedural phase. Because unprocessed memory locations are equivalent in P_i , there is only one implicit equivalence class in \mathcal{E}_i .

During the intraprocedural phase (lines 1–8), `EquivAnalysis` processes each object name obj referenced at a statement s in procedure P_i . `EquivAnalysis` calls `Update` to update \mathcal{E}_i , the set of equivalence classes for P_i , with $ASet(obj, s)$. `Update` (lines 19–33) inputs a set of equivalence classes \mathcal{E} and a set of memory locations E . For each memory location l in E , if l is in any equivalence class in \mathcal{E} (line 20), then l has been previously processed. `Update` marks E_c , the equivalence class that contains l , as being *touched* (line 21). `Update` also puts l in $M[E_c]$ to indicate that l is in both E and E_c (line 22). If l is in the implicit equivalence class, then l is processed for the first time. In this case, `Update` removes l from the implicit equivalence class and puts l in a new equivalence class E_{new} (line 24). After the loop is finished, E_{new} is the intersection of E and the implicit equivalence class and for each equivalence class E_c in \mathcal{E} , $M[E_c] = E \cap E_c$. `Update` adds E_{new} in \mathcal{E} (line 27). Then, for each equivalence class E_c that is touched, `Update` checks to see whether $|M[E_c]| < |E_c|$ is true. If so, E_c must be split into two equivalence classes. `Update` replaces E_c with $E_c - E$ by removing from E_c memory locations in

```

algorithm   EquivAnalysis
input       $\mathcal{P}$ : program to be analyzed
output     a set of equivalence classes for each procedure
declare     $P_i, P_j$ : procedures in  $\mathcal{P}$ 
              $\mathcal{E}_i$ : a list of equivalence classes of memory-location sets for  $P_i$ 
              $W$ : a list of procedures, sorted reverse topologically on
                 the strongly-connected components of the call graph

begin EquivAnalysis
1. foreach procedure  $P_i$  in  $\mathcal{P}$  do /* Intraprocedural phase */
2.   put  $P_i$  on  $W$ 
3.   foreach statement  $s$  in  $P_i$  do
4.     foreach object name  $obj$  at  $s$  do
5.       Update( $\mathcal{E}_i, ASet(obj, s)$ )
6.     endfor
7.   endfor
8. endfor
9. while  $W \neq \phi$  do /* Interprocedural phase */
10.   $P_i =$  remove procedure from head of  $W$ 
11.  foreach callsite  $C$  to  $P_j$  in  $P_i$  do
12.     $S_j :=$  {equivalence classes of  $P_j$ 's nonlocal memory locations};
13.    foreach equivalence class  $E$  in  $S_j$  do
14.      Update( $\mathcal{E}_i, E$ );
15.    endfor
16.    if  $\mathcal{E}_i$  changes then put  $P_i$ 's callers in  $W$ ; endif
17.  endfor
18. endwhile
end EquivAnalysis

```

```

procedure   Update( $\mathcal{E}, E$ )
input       $\mathcal{E}$ : a list of equivalence classes
              $E$ : a set of memory locations
output     updated  $\mathcal{E}$ 
declare     $M[E_c]$ : the intersection of equivalence class  $E_c$  and  $E$ 
begin Update
19. foreach memory location  $l$  in  $E$  do
20.   if  $l$  in any equivalence class  $E_c$  in  $\mathcal{E}$  then
21.     mark  $E_c$  touched
22.     put  $l$  in  $M[E_c]$ 
23.   else
24.     put  $l$  in  $E_{new}$ 
25.   endif
26. endfor
27. add  $E_{new}$  in  $\mathcal{E}$ 
28. foreach touched equivalence class  $E_c$  in  $\mathcal{E}$  do
29.   if  $|M[E_c]| < |E_c|$  then
30.     remove elements in  $M[E_c]$  from  $E_c$ 
31.     add  $M[E_c]$  to  $\mathcal{E}$ 
32.   endif
33. endfor
end Update

```

Figure 2: `EquivAnalysis`: Equivalence analysis algorithm.

$M[E_c]$ and then adds $M[E_c]$, which is $E_c \cap E$, to \mathcal{E} .

During the interprocedural phase (lines 9–18), `EquivAnalysis` iterates over the procedures in a reverse topological (bottom up) order on the strongly-connected components of the call graph. In each procedure P_i , `EquivAnalysis` checks each call C to procedure P_j (lines 11–17). `EquivAnalysis` computes the set of nonlocal memory locations that are accessed by P_j and partitions these memory locations into S_j , a set of equivalence classes according to the equivalence relations computed for P_j (line 12). An equivalence class in S_j can be computed by copying the nonlocal memory locations from an equivalence class in \mathcal{E}_j . `EquivAnalysis` then calls `Update` to update \mathcal{E}_i with each equivalence class in S_j (line 14). Finally, if \mathcal{E}_i changes, `EquivAnalysis` puts the procedures that call P_i on W for further updates (line 16). The iteration terminates when the sets of equivalence classes become stabilized.

2.3 Complexity

Using an array to map a location l to the equivalence class that contains l , line 20 in `Update` takes constant time. If we use a doubly-linked list to represent a memory location set, then adding an element to the set or removing an element from the set take constant time. Thus, lines 20–25 takes constant time. The complexity of lines 19–26 is $O(|E|)$. When `Update` adds E_{new} into \mathcal{E} , it must update the mapping array to map locations in E_{new} to E_{new} . Thus, the time taken by line 27 is proportional to $|E_{new}|$. Similarly, the time taken by line 31 is proportional $|M[E_c]|$. Line 30 takes $|M[E_c]|$ steps to remove elements from E_c . Therefore, the complexity of lines 28–32 is $O(|E|)$. The complexity of `Update` is $O(|E|)$.

Let n be the size of the program and s be the maximum size of an access set. In the intraprocedural phase of `EquivalenceAnalysis`, line 5 is executed at most n times because the total number of object names appearing in P_i is bounded by the size of P_i . Thus, the intraprocedural phase requires $O(n * s)$ time. In the absence of recursion, the information is propagated through a procedure call only once in the interprocedural phase, thus, at most c propagations are needed, where c is the number of calls in the program. In the presence of recursion, information is propagated through a call when there is a change in the set of equivalence classes of the called procedure. According to the algorithm, the set of equivalence classes changes when an equivalence class is divided into two equivalence classes. Thus, at most n changes can occur to the equivalence classes of a procedure. At most $n * c$ propagations are needed in the presence of recursion. Each propagation (lines 12–16) requires $|S_j|$ calls to `Update()` and the total cost of these calls is $O(n)$. Therefore, in the absence of recursion, the interprocedural phase requires $O(c * n)$ time; in the presence of recursion, the interprocedural phase requires $O(c * n^2)$ time.

The time complexity of `EquivalenceAnalysis` is $O((c + s) * n)$ in the absence of recursion; the time complexity of the algorithm is $O(c * n^2)$ in the presence of recursion.

2.4 Discussions

The definition of the equivalence relation in Subsection 2.1 can be tailored to a specific data-flow analysis. For example, the reuse-driven slicing algorithm [8] is interested in how the statements within a procedure can affect a nonlocal memory location that is modified in a procedure. To improve the performance of such an algorithm, we are interested the equivalence of two nonlocal memory locations when the memory locations are modified. We ignore the accesses that reference but do not modify these two memory locations. Under such a consideration, we define *modification-equivalence*. A memory location is always modification-equivalent to itself in a procedure P . Memory location l_1 is modification-equivalent to memory location l_2 in P if, at any statement s in P or in any procedure directly or indirectly called by P , given an object name obj that is modified at s , $l_1 \in ASet(obj, s)$ implies $l_2 \in ASet(obj, s)$ and $l_2 \in ASet(obj, s)$ implies $l_1 \in ASet(obj, s)$. For example, under this definition, x and g are modification-equivalent in `f2()`. Modification-equivalence is different than the equivalence relationship defined in 2.1. For example, according to the definition in 2.1, x and g are not equivalent because g is referenced by a direct object name at statement 9. For other specific data-flow analyses, the definition of equivalence of memory locations can be tailored in a similar way.

The algorithm discussed in Subsection 2.2 can be used with the alias information that is provided by any alias analysis algorithm. However, by investigating the properties of the representation of the alias information that is provided by a specific alias analysis algorithm, we can develop a more efficient algorithm to compute the equivalence classes.

Alias information provided by Steensgaard’s algorithm [17] can be represented as a *points-to* graph. In such a graph, vertices represent sets of memory locations and edges represent points-to relations or field accesses. For example, if p points to x , then there is a *points-to edge* from the vertex that represents p to the vertex that represents x . For another example, if vertex v_1 represents structure a and vertex v_2 represents field $a.f$, then there is a *field-access edge* labeled with “f” from v_1 to v_2 . In the points-to graph constructed by Steensgaard’s algorithm, the sets of memory locations represented by the vertices are disjoint and there is at most one points-to edge leaving a vertex and at most one field access edge leaving a vertex labelled with a specific field name. Given an indirect object name obj and a statement s , $ASet(obj, s)$ can be computed by first locating, in the points-to graph, the vertex that represents the variable in obj , and then, from this vertex, searching for a path in which the labels of the edges match the sequence of dereferences and field accesses in obj . We call the last vertex of the path as the *accessed vertex* of obj . $ASet(obj, s)$ is the set of memory locations represented by obj ’s accessed vertex.

Given such a points-to graph, if two memory locations l_1 and l_2 are represented by the same vertex and neither l_1 nor l_2 is accessed by direct names in procedure P or any procedure directly or indirectly called by P , then l_1 and l_2 are equivalent in P . According to this observation, we can develop an algorithm that is more efficient than the algorithm in Figure 2. The new algorithm first computes $S[P]$, the set of accessed vertices for the indirect names referenced in P or procedures directly or indirectly called by P . The new algorithm also computes the set of nonlocal memory locations referenced in P or procedures directly or indirectly called by P . The equivalence classes for P are derived by the following rules:

1. A directly accessed memory location forms an equivalence class.
2. Memory locations represented by a vertex in $S[P]$, except those memory locations directly accessed, form an equivalence class.

For example, according to the new algorithm, p , g and $g1$ are directly accessed in `f1()` and the vertex representing $\{x, y, z, g\}$ is accessed in `f1()` when the alias information is represented as a points-to graph. Therefore, the algorithm derives equivalence classes $\{x, y, z\}$, $\{g\}$, $\{g1\}$ and $\{p\}$ for `f1()`. Note that the equivalence classes for `f1()` computed here are different from the equivalence classes for `f1()` listed in Figure 1 because the equivalence classes listed in Figure 1 are computed with alias information provided by Landi and Ryder’s algorithm.

3 Empirical Studies

We performed several studies to evaluate the effectiveness of equivalence analysis in reducing the amount of data-flow information that is computed using alias information provided by each of the following alias analysis algorithms: Steensgaard’s algorithm (ST) [16], Liang and Harrold’s algorithm (LH) [13], Andersen’s algorithm (AND) [1], and Landi and Ryder’s algorithm (LR) [11]. We implemented the equivalence analysis algorithm, Steensgaard’s algorithm, Liang and Harrold’s algorithm, and Andersen’s algorithm using PROLANGS Analysis Framework (PAF) [7], which parses a C program and provides an interface to access the intermediate representation of the program. We used the implementation of Landi and Ryder’s algorithm provided together with PAF. We collected the data for the studies on a Sun Ultra 30 workstation with 640MB of physical memory.

Table 2 gives information about the subject programs we used for the studies. The subjects are ordered by the third column, the number of CFG (Control Flow Graph) nodes that are generated

by PAF to represent a subject program. This set of subjects contains various types of C programs. `unzip`, `lharc`, and `arc` are compression/decompression utilities. `flex` and `bison` are compiler generators. `assem` is an assembler, and `spim` is an assembly language simulator. `space` is an interface to a satellite controlling system. `larn` is a computer game. `mpegplay` is a program to display MPEG format image. `espresso` is an integer benchmark program. Most of these subjects have been used in other studies (e.g., [13, 15, 18, 19]). The fifth column in Table 2 shows that only a small number of procedures are involved in recursions in each of the subjects. The sixth column in the table shows that the alias information for Landi and Ryder’s algorithm is not available for `bison`, `spim`, `larn`, `mpegplay`, and `espresso` because the time required to run Landi and Ryder’s algorithm on these programs exceeded our limit (10 hours).

Table 2: Subject programs.

Program	LOC	CFG Nodes	Procedures	Recur Procs	LR
unzip	4075	1892	42	2	Yes
assem	2510	1993	58	0	Yes
lharc	3235	2539	89	3	Yes
flex	6902	3762	93	5	Yes
arc	7325	3955	116	2	Yes
space	11474	5601	137	0	Yes
bison	7893	6533	134	3	No
spim	24322	11352	263	5	No
larn	9966	11796	295	3	No
mpegplay	17263	11864	135	3	No
espresso	12864	15351	306	27	No

3.1 Study 1

In this study, we evaluate the efficiency of our algorithm in computing equivalence classes. Table 3 shows T , the time required to compute the set of nonlocal memory locations modified by each procedure (GMOD set), and T' , the time required to compute the equivalence classes for each procedure. From the table, we can see that the time required to compute the equivalence classes is close to the time required to compute the GMOD sets. Because many data-flow analyses are much more expensive than the computation of GMOD sets, those data-flow analyses might also be much more expensive than the computation of equivalence analyses. Thus, even if we can cut down a portion of the cost for those data-flow analyses, we can expect that the time required to compute the equivalence classes will be compensated by the time saved by using equivalence classes to improve those data-flow analyses. We are performing additional experiments to determine the effectiveness of using equivalence analysis to improve various data-flow analyses.

Table 3: Time in seconds to compute GMOD sets and equivalence classes.

program	ST		LH		AND		LR	
	T	T'	T	T'	T	T'	T	T'
unzip	0.17	0.17	0.14	0.13	0.14	0.15	0.14	0.15
assem	0.34	0.42	0.26	0.34	0.23	0.28	0.25	0.29
lharc	0.22	0.23	0.19	0.23	0.22	0.23	0.22	0.22
flex	0.43	0.51	0.35	0.40	0.35	0.40	0.45	0.49
arc	0.64	0.72	0.38	0.41	0.38	0.41	0.39	0.41
space	1.51	1.95	1.48	1.98	1.86	2.42	1.55	1.89
bison	0.58	0.62	0.56	0.56	0.62	0.62	–	–
spim	5.04	6.37	2.40	3.01	1.95	2.03	–	–
larn	3.80	4.59	2.18	2.41	2.11	2.29	–	–
mpegplay	1.19	1.21	1.13	1.14	2.09	2.06	–	–
espresso	9.09	11.91	7.34	9.28	8.62	10.52	–	–

From Table 3, we can also see that, the time required to compute the equivalence classes or GMOD sets increases when the precision of the alias analysis algorithm used to resolve dereferences

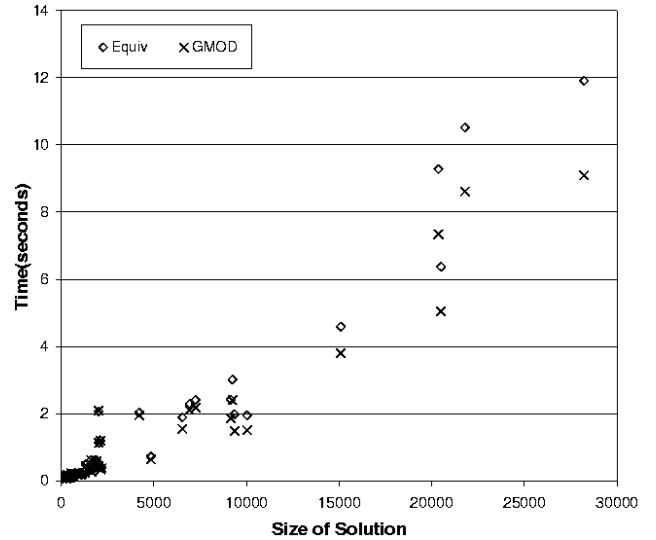


Figure 3: Correlation between time and size of solution.

of pointer variables decreases. This is because the time of computing the equivalence classes or GMOD sets is mainly determined by the number of memory locations that need to be propagated from one procedure to another. Figure 3 shows the relationship between the running time of computing equivalence classes and the summation of sizes of equivalence classes for the procedures in each of the subject programs (i.e., the size of the equivalence analysis solution). The figure also shows the relationship between the running time of computing GMOD sets and the summation of sizes of GMOD sets for the procedures (i.e., the size of GMOD analysis solution). In the figure, a point is drawn to show the time to compute a solution and the size of the solution for a subject program when the solution is computed using alias information provided by one of the four alias analysis algorithms. The figure shows a strong correlation between the computation time and the size of the solution. Given a subject program, if the dereferences of pointer variables are resolved using less precise alias analysis algorithm, a procedure might access more nonlocal memory locations, and thus, the size of the solution might increase. Therefore, it takes more time to compute the solution using alias information provided by less precise alias analysis algorithm.

3.2 Study 2

In this study, we evaluate the effectiveness of the equivalence analysis in reducing the data facts at each statement. For each statement that modifies memory locations through pointer dereference, we recorded the number of memory locations that are modified by the statement (ThruDeref Mod) and the number of equivalence classes that contain these memory locations. Figure 4 shows the results of this study when the alias information is provided by each of the four alias analysis algorithms. The total length of each bar represents the average size of ThruDeref Mod and the length of the segment filled with slanted lines represents the average number of equivalence classes that contain the memory locations in the ThruDeref Mod set. For example, for `unzip`, the average size of ThruDeref Mod when alias analysis is performed using Steensgaard’s algorithm is 11.6 whereas the number of equivalence classes of memory locations for ThruDeref Mod is 1.9. From the graph, we can see that the memory locations modified by a statement through dereferences almost always fall into the same equivalence class regardless of the precision levels of the aliasing information. This result re-

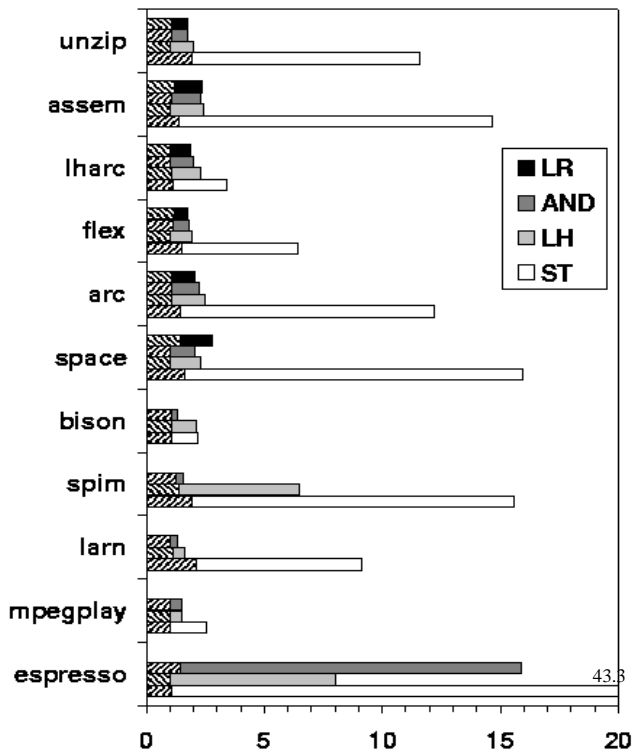


Figure 4: Average size of ThruDeref Mod.

flects the common way a pointer is used: one pointer is used to point to one or two conceptual objects in a procedure. The result also indicates that equivalence analysis can effectively reduce the data facts, such as definitions, at a statement because a data-flow analysis creates the data facts only for a representative memory location of each equivalence class. Thus, we can expect that using equivalence analysis may effectively improve the efficiency of many intraprocedural data-flow analyses.

3.3 Study 3

Table 4: Average size of GMOD sets.

program	ST		LH		AND		LR	
	S	S'	S	S'	S	S'	S	S'
unzip	24.2	9.1	14.4	9.6	13.8	9.7	10.9	8.7
assem	38.9	8.0	30.0	11.3	23.1	10.6	23.6	11.6
lharc	12.3	7.5	11.1	7.6	10.3	7.6	7.4	6.1
flex	22.5	13.6	17.0	14.0	16.2	13.9	14.6	13.5
arc	45.7	13.7	20.7	14.3	19.3	14.2	18.8	13.8
space	77.8	14.2	72.6	17.5	70.9	19.1	50.7	19.5
bison	14.8	10.0	14.4	10.0	12.4	10.6	-	-
spim	121	19.1	54.7	19.5	24.9	19.8	-	-
larn	58.1	20.4	27.9	21.0	26.7	21.0	-	-
mpegplay	15.7	13.0	15.1	13.6	15.0	13.6	-	-
espresso	92.2	17.1	66.5	25.0	71.3	20.3	-	-

In this study, we investigate the effectiveness of equivalence analysis in reducing the summary information for each procedure. We compare the average size of the GMOD set, the set of memory locations modified by a procedure, and the average size of the *reduced GMOD set*, the set of nonlocal memory locations obtained by removing from a GMOD set all memory locations except the representative memory locations of equivalence classes. Because an interprocedural data-flow analysis computes and stores summary

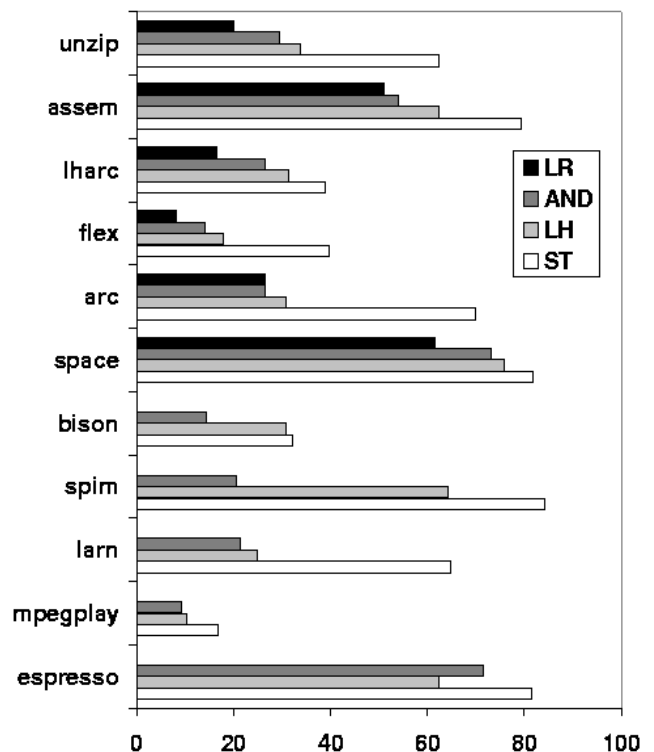


Figure 5: Reduction (%) of the GMOD size.

information for each memory location in the GMOD set, the reduction of the average sizes of GMOD sets may indicate the effectiveness of using equivalence analysis to improve the performance of the interprocedural data-flow analysis.

Table 4 shows S , the average size of the GMOD sets, and S' , the average size of the reduced GMOD sets. The results show that the average size of the reduced GMOD set is about the same regardless of the precision of the alias information. For example, the average size of the reduced GMOD sets for program `larn` is about 21 when we compute the reduced GMOD sets with alias information provided by any of the four algorithms. This result indicates that an interprocedural data-flow analysis might have the same performance regardless of the alias information it uses. Thus, equivalence analysis may provide a way to solve the performance problem of using aliasing information provided by Steensgaard's algorithm in interprocedural data-flow analyses. Figure 5 shows the percentage reduction in the size of GMOD sets using equivalence analysis. The results show that using equivalence classes can effectively reduce the size of the GMOD set. Thus, we can expect that using equivalence classes may effectively improve the performance of many interprocedural data-flow analyses in the presence of pointer variables.

4 Related Work

There are several approaches to reduce the cost of data-flow analyses using equivalence relations. Duesterwald, Gupta, and Soffa [5] propose *congruence partitioning*, a technique that identifies *congruence relations*, a kind of equivalence relations, among data-flow equations for a program and uses the congruence relation to reduce the number of data-flow equations. Two congruent data-flow equations have the same fixed points. Thus, one of the equations is redundant and can be eliminated. Duesterwald, Gupta, and Soffa

further propose algorithms to compute the congruence relations induced by idempotence and common subexpressions. Like congruence partitioning, our technique can also reduce the number of data-flow equations in the presence of pointers. Our technique differs from congruence partitioning in that our technique computes the equivalence relations among memory locations before the data-flow equations are created. Therefore, our technique can be used together with congruence partitioning to further reduce the number of data-flow equations. Further more, unlike congruence partitioning, our technique not only can improve the performance of data-flow analyses that use data-flow equations, our technique also can improve the performance of data-flow analyses that use other representations.

There are other techniques that can improve the efficiency of data-flow analysis in the presence of pointer variables. Choi, Cytron, and Ferrante [3] propose a technique that builds data-flow representation linear in the number of definitions and uses in the program in the presence of preserving definitions. Preserving definitions can be induced by assignments to dereference of pointer variables or by assignments to the elements of arrays. Without such a technique, the preserving definitions can cause the data-flow chains to grow quadratically in the number of definitions and uses in a program. Choi, Cytron, and Ferrante's technique builds both definition-use chains and definition-definition chains for the program variables. However, the technique builds only the *transitive reduction* of these chains. Together with *static single assignment* forms [4], this technique guarantees that the number of chains built for a program is linear in the number of definitions and uses in a program. Choi, Cytron, and Ferrante also discuss how this compact representation can be used for data-flow analyses.

Our technique differs from Choi, Cytron, and Ferrante's technique in that our technique addresses a problem that is orthogonal to the preserving definition problem in the presence of pointer variables. Our technique can be used to reduce the size of data facts, such as definitions and uses, that are introduced at the statements where dereferences of pointer variables are referenced. Thus, our technique can be used to further reduce the size of the compact representation built by Choi, Cytron, and Ferrante's technique. In our preliminary studies, we observed that in many small procedures in a program, a formal parameter pointer can point to a large set of memory locations and these memory locations are referenced but not modified in a procedure. In this case, Choi, Cytron, and Ferrante's technique will not be able to reduce the size of the representation of the procedures. However, our technique can reduce the size of the representation because we can use one representative memory locations to represent all the memory locations pointed to by that formal parameter pointer.

Our algorithm for computing equivalence classes is one of the *partitioning algorithms* (e.g., [5, 9]) that compute a partition of some items according to some criteria. A partitioning algorithm first initializes the partition with an overestimate. The algorithm then iteratively refines the partition until the partition becomes *safe*, that is, if two items t_1 and t_2 are in the same set, then t_1 and t_2 should be indistinguishable according to the criteria. The details of the efficient implementation and complexity analysis of the refinement step (`Update` in our algorithm) can be found in [9].

5 Conclusions

We presented equivalence analysis, a technique that can be used to improve many data-flow analyses in the presence of pointers. We also presented an efficient algorithm to compute the equivalence classes. We conducted several studies to evaluate the performance and effectiveness of our technique. Our studies show that this technique can reduce the amount of data-flow information generated in a program, and thus, may effectively improve the performance of many data-flow analyses on programs that use pointer variables.

We are applying our technique to various data-flow analyses and conducting studies to investigate the effectiveness of equivalence analysis in improving the performance of the data-flow analyses.

6 Acknowledgments

This work was supported in part by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University. The Programming Languages Research Group at Rutgers University developed, and freely distributes, the PROLANGS Analysis Framework. Alberto Pasquini provided the source code for the `space`. The anonymous reviewers provided 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 Symposium on the Foundation of Software Engineering*, pages 46–55, November 1998.
- [3] J. D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, February 1994.
- [4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.
- [5] E. Duesterwald, R. Gupta, and M. L. Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *5th International Conference on Compiler Construction*, pages 357–373, April 1994.
- [6] 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.
- [7] Programming Languages Research Group. PROLANGS Analysis Framework. <http://www.prolangs.rutgers.edu/>, Rutgers University, 1998.
- [8] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *The 20th International Conference on Software Engineering*, pages 74–83, April 1998.
- [9] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in finite automata. In *Theory of Machines and Computations*. Academic Press, 1971.
- [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. 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.
- [13] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Joint 7th European Software Engineering Conference and 7th ACM Symposium on Foundations of Software Engineering*, September 1999.
- [14] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis 4th International Symposium, Lecture Notes in Computer Science Vol 1302*, pages 16–34, 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 *Proceedings of the International Conference on Compiler Construction*, pages 136–150, 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] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer analysis: A step toward practical analyses. In *Proceedings of the Fourth Symposium on the Foundation of Software Engineering*, pages 81–92, November 1996.
- [19] 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.