Reducing the Cost of Data Flow Analysis By Congruence Partitioning [†]

Evelyn Duesterwald, Rajiv Gupta, Mary Lou Soffa

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260

Abstract. Data flow analysis expresses the solution of an information gathering problem as the fixed point of a system of monotone equations. This paper presents a technique to improve the performance of data flow analysis by systematically reducing the size of the equation system in any monotone data flow problem. Reductions result from partitioning the equations in the system according to congruence relations. We present a fast $O(n \log n)$ partitioning algorithm, where n is the size of the program, that exploits known algebraic properties in equation systems. From the resulting partition a reduced equation system is constructed that is minimized with respect to the computed congruence relation while still providing the data flow solution at all program points.

1 Introduction

Along with the growing importance of static data flow analysis in current optimizing and parallelizing compilers comes an increased concern about the high time and space requirements of solving data flow problems. Experimental studies show that performing sophisticated analyses over even small to medium-sized programs can take several hours [Lan92]. Phrased in the traditional data flow framework [KU77], the solution of a data flow problem is the greatest fixed point of a system of monotone equations. Each equation expresses the solution at one program point in terms of the solutions at immediately preceding (or succeeding) points. This formulation may result in overly large equation systems, limiting both the time and space efficiency of even the fastest fixed point evaluation algorithm.

A closer inspection of equation systems reveals that their sizes are unnecessarily enlarged due to the inherent inclusion of redundant equations. The structure of data flow equation systems requires the propagation of intermediate results throughout the program, including the propagation to program points where these results are of no relevance. As a consequence, multiple equations in the system carry identical information. Equations that duplicate information already expressed by other equations are redundant and their repeated evaluation during the fixed point iteration is clearly undesirable. If equivalent but smaller equation

[†] Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh.

systems without redundancies were constructed, fixed point computations would be faster, independent of the evaluation algorithm used.

We present in this paper a systematic approach to minimize data flow equation systems by discovering congruence relationships among equations. Two equations are congruent only if their fixed points are equal. Thus, at least one of two congruent equations is redundant and can therefore be eliminated. Given a congruence relation an equivalent but reduced equation system is constructed by including only a single equation from each class of congruent equations. Our approach is general in that it is applicable to all monotone data flow analysis problems.

Previous approaches to avoid unnecessary evaluations of data flow equations include the methods based on *static single assignment form* [WZ85, AWZ88, RWZ88, CLZ86], *sparse evaluation graphs* [CCF90] and *dependence flow graphs* [JP93]. The idea behind these approaches is to by-pass some of the unnecessary equation evaluations by manipulating the underlying graphical program representation. We show that, by viewing the problem as an algebraic problem of congruence relations, our approach allows for conceptually simple algorithms that are both more general and powerful than previous graph-oriented methods.

The results of this paper are summarized as follows. We define a *congruence relation* among data flow equations that is based on exploiting the known idempotence property of the *meet* operator in the system. No assumptions are made on the sequence of intermediate values an equation may take during the fixed point iteration. These sequences of intermediate values are highly dependent on the particular iteration strategy that is used to compute the fixed point, but the notion of congruence is a valid relation for any such strategy. A fast partitioning algorithm is presented to compute the idempotence congruence relation in $O(n \log n)$ time and $O(n)$ space, where n is the size of the program. Using the computed congruence relation, a reduced equation system is constructed that only contains a single equation from each congruence class. By the definition of congruence, it is sufficient to compute the fixed point over only the reduced system using any of the standard evaluation strategies.

The approach of reducing equation systems by computing congruence relations can easily be extended to include other notions of congruence. The congruence relations discussed in [DST80, NO80] are based on common subexpressions. Alpern et al. [AWZ88] used a fast $O(n \log n)$ algorithm due to Hopcroft for minimizing finite automata to compute congruences by common subexpression for program optimization. We show that Hopcroft's algorithm can equally well be applied to disover common subexpression in data flow equations systems in order to enable further reductions.

The asymptotic performance of congruence partitioning to reduce a data flow equation system only depends on the size of the equation system. The complexity of the data flow problem, i.e., the cost of actually evaluating the equations, does not impact on the performance of the partitioning algorithm. The complexity of data flow problems varies dramatically, ranging from simple problems, such as live variable analysis, that can be implemented efficiently using bit vectors, to sophisticated time- and space-intensive analyses, such as alias analysis. Naturally, the benefits of congruence partitioning increase with the complexity of the data flow problem.

We present the pertinent background in data flow analysis in Section 2. Section 3 introduces congruence relations among data flow equations. The idempotence congruence relation along with our fast partitioning algorithm is presented in Section 4. Section 5 discusses congruence computations based on common subexpressions. We compare congruence partitioning with previous work and discuss other related work in Section 6. Conclusions are given in Section 7.

2 Data Flow Equation Systems

A data flow analysis is defined over a graphical representation of a program, usually the control flow graph $G = (N, E, n_0)$. The nodes N represent basic blocks [ASU86] in the program with a unique entry node n_0. The edges E represent transfer of control among basic blocks. We assume that $|E| = O(|N|)$. Given a node $n \in N$, *pred*(n) (*succ*(n)) denotes the set of immediate predecessors (successors) of node n in G.

Data flow analyses are modeled in a *data flow framework* $D = (L, F, G, m)$, where:

- $(L, \leq, \bot, \top, \wedge)$ is a semi-lattice with a set L, a partial order $\leq$, a least element $\bot$ (bottom), a greatest element $\top$ (top) and a meet operator $\wedge$, such that for all $x, y, z \in L$: $x \wedge x = x$ (idempotence), $x \wedge y = y \wedge x$ (commutativity), and $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (associativity).
- $F \subseteq \{f : L \mapsto L\}$ is a space of monotone flow functions over L.
- $G = (N, E, n_0)$ is a control flow graph
- $m : N \mapsto F$ is a mapping of program nodes to functions in F.

The function $m(n)$ mapped to a node n (also denoted f_n) models the data flow when execution passes through node n. If $x \in L$ holds on entry of a node n then $f_n(x) \in L$ holds on exit from node n. [2].

A data flow framework induces a *system of data flow equations* parameterized by the nodes in the control flow graph:

$$\begin{aligned} x[n_0] &= f_{n_0}(\bot) \\ x[n] &= f_n\Big(\bigwedge_{p \in pred(n)} x[p]\Big) \text{ for } n \neq n_0 \end{aligned}$$

The solution of a data flow framework is the *greatest fixed point* assignment *gfp* : $N \mapsto L$ of the equation system based on the initial value $\top$. The monotonicity of F ensures that the greatest fixed point *gfp*(n) of each equation $x[n]$ exists and is unique. For each node $n \in N$, *gfp*(n) describes the data flow solution that holds on exit of node n.

[2] The framework models both forward and backward analyses by assuming that in a backward analysis the transposed control flow graph $G^t = (N, E^t)$ is used, where $E^t = \{(n, m) \mid (m, n) \in G\}$.

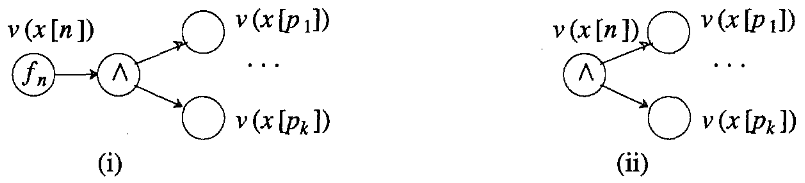


Fig. 1. The translation of equations into graphs.

The equation system X can be represented by a labeled directed graph $G = (V, E)$. The vertices in V represent equation variables and the operations of the right hand side of equations. An edge (v, w) in E describes that the expression represented by vertex v depends on the input represented by vertex w. We refer to this graph as an *equation graph.*

An equation $x[n] = f_n(\bigwedge_{p \in pred(n)} x[p])$ is translated into the graph shown in Fig. 1 (i). Corresponding to the function symbol f_n is a vertex $v(x[n])$ with $label(v(x[n]) = f_n$ that has a single successor vertex with label $\wedge$. The vertex labeled $\wedge$ has successors $v(x[p])$ for each predecessor p of node n. If the function f_n is the identity function, the equation reduces to $x[n] = \bigwedge_{p \in pred(n)} x[p]$. In this case no vertex for the function symbol is created, and the vertex $v(x[n])$ is the vertex labeled $\wedge$ as shown in Fig. 1 (ii). We partition the vertex set V into a set $V_\wedge$ of vertices labeled $\wedge$ (meet vertices) and a set V_f of vertices with a label denoting any other function symbol (function vertices).

Due to the direct correspondence between the graphical and textual representations of an equation system we will not always explicitly distinguish between the two. In discussing equation systems we assume that their graphs are transformed into graphs whose vertices have an indegree and outdegree of at most 2. This transformation is analogous to transforming the textual representation of the equation system into some form of three-address-code. The associativity of the meet operator ensures that a graph can always be transformed into this form by adding some additional vertices for each vertex whose indegree or outdegree is greater than 2. At most a constant number of vertices is added per edge in this process and the number of vertices remains $O(n)$ [DST80], where $n = |N|$ is the number of nodes in the control flow graph.

As the running example in this paper we consider alias analysis performed over procedure `Insert`, shown below. Alias analysis computes pairs of aliased variables. To simplify the representation, we consider a simple alias analysis that assumes that if a variable q is aliased to a variable p then any variable that q points to is aliased by any variable p points to. The lattice elements are collections of alias relations. A collection could be simply a set of alias pairs or, alternatively, a partition of the variables into sets of aliased variables. We omit showing the control flow graph for procedure `Insert`. The relevant program points at which data flow information is computed are numbered in curly braces in procedure `Insert`.

```
procedure Insert(x, val) /* insert a value val in a binary tree x */
begin
   val:=h(val); { 1 }
   repeat { 2 }
      p:=x; { 3 }
      if (val < x->key) then x=x->left; { 4 }
                        else x=x->right; { 5 }
   ;{ 6 }
   until (x = NULL); { 7 }
   new(x); x->key:=val; x->left:=NULL; x->right:=NULL; { 8 }
   if (val < p->key) then p->left:=x { 9 }
      else p->right:=x; { 10 }
   ;{ 11 }
end
```

The equation system that expresses the analysis over procedure Insert is shown in Fig 2(i) along with its equation graph in Fig. 2 (ii). Each equation $x[n]$ refers to the alias information that holds at the program point n marked in procedure Insert. The meet operator $\wedge$ represents the union of two collections of alias relations into a single collection. The data flow equations are also based on a function $kill[y]$ that takes as an argument a collection of alias relations C and eliminates all alias relations for variable y from C. For more details of the analysis we refer the reader to [CC77]. With respect to congruence partitioning, the meet $\wedge$ and other functions like $kill[y]$ are merely uninterpreted symbols.

3 Congruence Relations

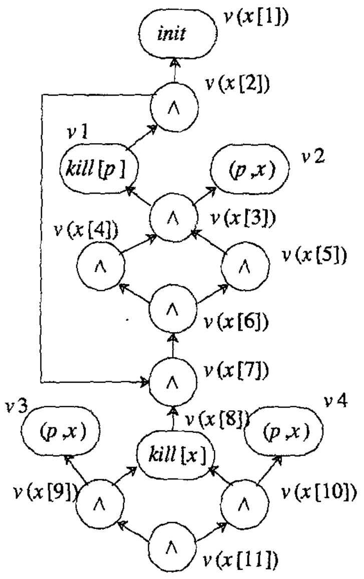
Given an equation system, our objective is to minimize the size of the system without evaluating any equation. Unfortunately, even the following restricted version of this minimization problem is NP-complete [GJ79]: Given a set of expressions constructed from uninterpreted constants and only the single commutative and associative operator, determine the minimum number of operations needed to evaluate all expressions. Thus, in general, we cannot expect an efficient algorithm to be able to eliminate all redundancies from X. However, we show that it is possible to minimize X with respect to certain well-defined classes of redundancies using a fast algorithm.

Redundancies are eliminated by discovering congruence relationships among equations. We only consider relationships among the final fixed point values of equations; two equations $x[n]$ and $x[m]$ in a system X are called *congruent* only if $gfp(n) = gfp(m)$.

Congruence is an equivalence relation (symmetric, reflexive and transitive) and therefore induces a partition π of the equations into *congruence classes*. All equations that are contained in the same congruence class in π have an identical fixed point. Given π we can reduce the original equation system by eliminating all but one equation from each congruence class. By the definition of congruence, the resulting reduced system is guaranteed to provide the same fixed point solution as the original system, independent of the particular evaluation strategy used. If

$$
\begin{aligned}
x[1] &= init \\
x[2] &= x[1] \wedge x[7] \\
x[3] &= kill[p](x[2]) \wedge (p, x) \\
x[4] &= x[3] \\
x[5] &= x[3] \\
x[6] &= x[4] \wedge x[5] \\
x[7] &= x[6] \\
x[8] &= kill[x](x[7]) \\
x[9] &= (p, x) \wedge X[8] \\
x[10] &= x[8] \wedge (p, x) \\
x[11] &= x[9] \wedge x[10]
\end{aligned}
$$

(i)



(ii)

Fig. 2. The data flow equation system (i) for a simple alias analysis over procedure Insert and its graphical representation (ii).

needed, the solution of the reduced system can later be expanded to the solution of all original equations using the computed partition π.

The following sections discuss how congruence relationships among the equations can be discovered by exploiting properties in the equation graph. We first present a partitioning algorithm that discovers congruence based on the idempotence property of the meet operator. We then show how an algorithm due to Hopcroft for minimizing finite automata can be adapted to discover additional congruences that result from common subexpressions. Fig. 3 shows the reductions in the equation system for the alias analysis of procedure Insert that are achieved by congruence partitioning explained in the next sections.

4 Congruence by Idempotence

This section describe the detection of congruences among data flow equations that result from the idempotence of the meet operator $\wedge$. Recall that a data flow equation is of the form:

$$x[n] = f_n(\bigwedge_{p \in pred(n)} x[p])$$

Trivial congruences result from a special case of equations, where the function f_n is the identity function and node n has only a single predecessor p. In this

$$\begin{aligned}
x[1] &= \mathit{init}\\
x[2] &= x[1] \wedge x[7]\\
x[3] &= \mathit{kill}[p](x[2]) \wedge (p,x)\\
x[4] &= x[3]\\
x[5] &= x[3]\\
x[6] &= x[4] \wedge x[5]\\
x[7] &= x[6]\\
x[8] &= \mathit{kill}[x](x[7])\\
x[9] &= x[8] \wedge (p,x)\\
x[10] &= x[8] \wedge (p,x)\\
x[11] &= x[9] \wedge x[10]
\end{aligned}$$

(i)

$$\begin{aligned}
x[1] &= \mathit{init}\\
x[2] &= x[1] \wedge x[3]\\
x[3] &= \mathit{kill}[p](x[2]) \wedge (p,x)\\
x[8] &= \mathit{kill}[x](x[3])\\
x[9] &= x[8] \wedge (p,x)\\
x[10] &= x[8] \wedge (p,x)\\
x[11] &= x[9] \wedge x[10]
\end{aligned}$$

(ii)

$$\begin{aligned}
x[1] &= \mathit{init}\\
x[2] &= x[1] \wedge x[3]\\
x[3] &= \mathit{kill}[p](x[2]) \wedge (p,x)\\
x[8] &= \mathit{kill}[x](x[3])\\
x[9] &= x[8] \wedge (p,x)
\end{aligned}$$

(iii)

Fig. 3. The original equation system for the alias analysis of procedure `Insert` (i), the reduced system after partitioning by idempotence (ii), and the reduced system after a combined partitioning by common subexpression and idempotence and (iii).

case the equation reduces to a simple copy equation $x[n] = x[p]$. Clearly, the fixed points of $x[n]$ and $x[p]$ are identical and $x[n]$ and $x[p]$ are congruent.

The congruence relation based on copies can easily be computed in a single pass over the equation system. Initially, we assume each equation $x[n]$ is in a separate congruence class. For each copy equation $x[n] = x[m]$ that is encountered, the congruence class of $x[m]$ is merged into class of $x[n]$ creating a single class. A reduced equation system without copies is constructed by including from each congruence class only a single representative equation. Each operand that occurs in an included equation is replaced by the representative of its congruence class.

Idempotence congruence extends this trivial notion of copy congruences by also covering *hidden copies.* A hidden copy is an equation of the form $x = y \wedge z$ such that y and z are congruent. By the idempotence of the meet operator, the congruence of y and z implies that $\mathit{gfp}(y) \wedge \mathit{gfp}(z)$ reduces to $\mathit{gfp}(y)$ and equation x is essentially a copy. Thus, it can be determined that all three variables x, y, and z are congruent. Over an equation graph G, we obtain the following definition with respect to the idempotent meet operation $\wedge$ in G.

Definition 1 (Congruence by idempotence). *Let $G = (V, E)$ be an equation graph. A relation C on V is called an idempotence congruence relation, if $(v, w) \in C$ implies one of the following conditions:*
(1) *$v = w$ (the vertices v and w are identical), or*
(2) *one of the vertices, say v, is labeled $\wedge$ and $(v, u) \in E$ implies $(u, w) \in C$*

To verify that C is indeed a congruence relation we have to ensure that the base case of the recursive rule (2) as well as the application of rule (2) can only yield congruent pairs of vertices. The base case of rule (2) declares $(v, w) \in C$ if w is the sole destination of edges leaving v. In this case v represents a copy equation and v and w are congruent. If all destinations of edges leaving v are congruent to a vertex w then v reduces to w by idempotence and v and w are congruent (application of rule 2).

By its recursive definition, the idempotence congruence relation is not unique if G contains cycles. Consider the equations in (a):

$$\begin{array}{lll} x[1] = f(x[0]) & x[1] = f(x[0]) & x[1] = f(x[0]) \\ x[2] = x[1] \wedge x[3] & x[2] = x[1] \wedge x[2] & \\ x[3] = x[2] & & \\ \text{(a)} & \text{(b)} & \text{(c)} \end{array}$$

The partition $\pi_1 = \{c_1 = \{x[1]\}, c_2 = \{x[2], x[3]\}\}$ with the corresponding system (b) describes an idempotence congruence relation. However, the partition $\pi_2 = \{c_1 = \{x[1], x[2], x[3]\}\}$ also describes an idempotence congruence relation that provides the reduced system (c)[3]. We are interested in the maximal idempotence congruence relation (fewest number of congruence classes) for an equation graph. For the remainder of this paper, we use the symbol $C^\star$ to refer to the maximal idempotence congruence relation according to Definition 1. The relation $C^\star$ provides the *coarsest* partition $\pi^\star$ of the vertices in an equation graph such that two vertices are in the same partition only if they are congruent according to Definition 1.

We present a fast partitioning algorithm to compute $\pi^\star$ that starts with an initial partition π that places all possibly congruent pairs of equations in the same class. The partition π is iteratively refined until a stable partition $\pi^\star$ is reached that is consistent with the definition of $C^\star$. Given partition $\pi^\star$ we construct the equation system that is minimized with respect to idempotence congruence in the same way as previously described. That is, from each congruence class in $\pi^\star$ only one representative equation is included. The resulting equation system contains no copy equations and no hidden copies due to idempotence.

4.1 The Partitioning Algorithm

Computing the partition $\pi^\star$ by iterative refinement requires first determining an appropriate initial partition. If two vertices are initially placed in different congruence classes they can never discovered to be congruent. Thus, the initial partition must *overestimate* the congruence relation $C^\star$. A partition π overestimates $C^\star$, if $(v, w) \in C^\star$ implies that the vertices v and w are placed in the same congruence class in π. In order to enable the partitioning algorithm to converge quickly to $\pi^\star$, we are interested in finding the *finest* initial partition that overestimates $C^\star$.

Standard graph partitioning algorithms [AHU74] are based on an initial partition of the vertices by their label. Unfortunately, we cannot follow this approach for computing $C^\star$. Although function vertices with a different label cannot be congruent by idempotence, meet vertices may be congruent to any function ver-

[3] Note that the congruence between $x[1]$ and $x[2]$ only holds with respect to the *greatest* fixed point defined with the initial value $\top$ at each equation.

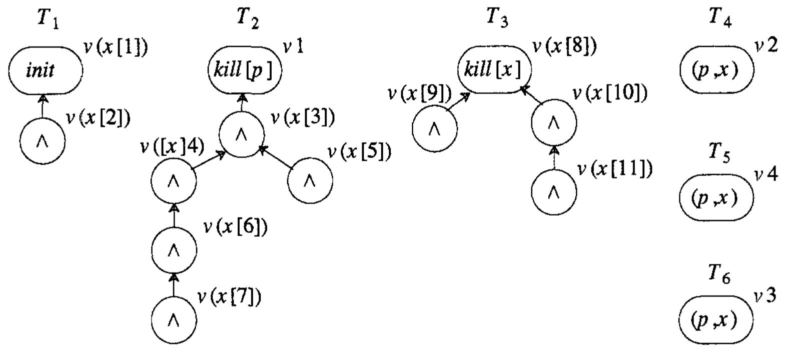


Fig. 4. A reverse DFST partition $\pi = T_1, \ldots T_6$ of the equation graph from Fig. 2(ii).

tex. We present a new partitioning algorithm and show how an overestimating initial partition of the vertices can be constructed in a canonical way.

Congruence classes in a partition are represented as *reverse trees* of vertices in an equation graph G. A reverse tree is a tree in which edges are directed from children to parent vertices. Thus, $\pi = T_1, \ldots, T_k$ is a collection of disjoint reverse trees and each tree T_i is a subgraph of the equation graph G. We will often refer to the reverse trees in a partition simply as trees and use the following notation for a given partition forest π. The root vertex of a tree T in π is denoted *root(T)*. For a given vertex v in a tree T, $parent(v)$ is the unique predecessor of v in G that is contained in T.

We construct an initial partition of the vertices in an equation graph G during a single reverse depth-first traversal of G, i.e., a depth-first traversal of the transposed graph of G. The resulting partition contains one tree (congruence class) for each function vertex in G. The tree T_v for a function vertex v is constructed by traversing each reachable edge in reverse direction, such that T_v is a reverse depth-first spanning tree (DFST) that is rooted at v and that does not include any other function vertex. The resulting forest of reverse DFSTs is called a *reverse DFST partition.* A reverse DFST partition for the equation graph from Fig. 2(ii) is shown in Fig. 4.

A reverse DFST partition for an equation graph is not unique since selections among multiple candidates to visit next are made arbitrarily. We show in the following lemma that any reverse DFST partition π safely overestimates $C^\star$.

Lemma 1 *Let π be a reverse DFST partition for a graph G and let v and w be vertices in G. If $(v, w) \in C^\star$ then v and w are in the same tree in π.*

Proof. For a vertex v in a tree T in π we use the notation $level(v)$ to denote the length of the path from v to $root(T)$. Given two distinct trees T_1 and T_2 in π, we first show that if v is a vertex in T_1 then $(v, root(T_2)) \notin C^\star$ by induction on $l = level(v)$. (Basis $l = 0$) Clearly, $(root(T_1), root(T_2)) \notin C^\star$ since two distinct function vertices cannot be congruent by idempotence. (Ind. $l > 0$) By hypothesis $(w, root(T_2)) \notin C^\star$ if $level(w) < l$. Assume $(v, root(T_2)) \in C^\star$ and $level(v) = l$.

Then by rule (2) of Def. 1 also $(parent(v), root(T_1)) \in C^\star$ which contradicts the hypothesis since $level(parent(v)) < l$.

Consider now two vertices v and w that are in distinct trees T_1 and T_2 and neither v nor w are the root vertex in their tree. If $(v,w) \in C^\star$ then it follows by rule (2) of Def. 1 that for the parent of at least one of the vertices, say v, we obtain $(parent(v), w) \in C^\star$. By repeatedly applying this argument, we eventually derive that the root vertex of one the trees must be congruent under $C^\star$ to a vertex in the other tree, which was however shown not to be possible. Hence, $(v,w) \notin C^\star$. □

Our algorithm *Partition* operates on an initial reverse DFST partition π by subsequently refining π until the current partition is consistent with the definition of $C^\star$. In the resulting partition $\pi^\star$ two vertices v and w are left in the same tree only if $(v,w) \in C^\star$.

Algorithm *Partition*, shown below, maintains two lists of vertices, *worklist* and *splitlist*. *Worklist* is a list of current partition trees to be examined. Each tree T in *worklist* is examined in line (5) to determine whether it contains an interior vertex v that has a successor not in T. In this case, the vertices v and *parent*(v) in T cannot be congruent under idempotence. To ensure that the two vertices do not remain in the same tree, vertex v is placed in *splitlist*. During the inner loop the tree of each vertex u in *splitlist* is split by disconnecting the subtree rooted at u. After the split one of the two resulting subtrees is placed in *worklist* to ensure that vertices that may trigger a subsequent split will be examined. *Partition* terminates when *worklist* is exhausted with the final partition $\pi^\star$.

Algorithm *Partition* performs the following operation on partition trees:
split(v): disconnects and returns the subtree rooted at v of the tree containing vertex v.

Algorithm *Partition (Partitioning by idempotence)*
Input: Equation graph $G = (V = V_f \cup V_\wedge, E)$
Output: Partition $\pi^\star = T_1, \ldots, T_k$ of V according to $C^\star$
Method:

```
1.  create an initial reverse DFST partition $\pi = T_1, \ldots, T_l$ of the vertices in $V$;
2.  worklist ← $\{T_1, \ldots, T_l\}$;
3.  while worklist $\neq \emptyset$ do
4.      select and remove a tree $T$ from worklist;
5.      splitlist ← $\{v \in V_\wedge \mid v$ has one successor in $T$ and one successor not in $T\}$ ;
6.      for each $u \in$ splitlist such that $u$ is not a root vertex in $\pi$ do
7.          let $T_1$ be the tree containing vertex $u$;
8.          add $T_2$ ← split($u$) as a new tree to $\pi$;
9.          if $T_1 \in$ worklist then add $T_2$ to worklist
10.         else add the smaller of $T_1$ and $T_2$ to worklist;
11.     endfor;
12. endwhile;
```

We apply *Partition* to the initial reverse DFST partition from Fig. 4. The initial partition π corresponding to Fig. 4 and the final partition $\pi^\star$ after algorithm

Partition terminates are shown below, where congruence classes are displayed in columns. The original complete equation system was shown in Fig. 3 (i). The final partition $\pi^\star$ describes the congruences in that system that result from the copy equations $x[4], x[5], x[7]$ and from the hidden copy equation $x[6]$. Specifically, all equations in the column for $x[3]$ in $\pi^\star$ are found to have the same fixed point as equation $x[3]$. The reduced equation systems in which the four redundant (hidden) copy equations are eliminated is shown in Fig. 3 (ii).

π

$x[1]$	$x[3]$	$x[8]$
$x[2]$	$x[4]$	$x[9]$
	$x[5]$	$x[10]$
	$x[6]$	$x[11]$
	$x[7]$	

$\pi^\star$

$x[1]$	$x[2]$	$x[3]$	$x[8]$	$x[9]$	$x[10]$	$x[11]$
		$x[4]$				
		$x[5]$				
		$x[6]$				
		$x[7]$				

4.2 Analysis

We show that algorithm *Partition* computes the congruence relation $C^\star$, that is, the output partition $\pi^\star$ is the coarsest partition, such two vertices v and w are contained in the same tree in $\pi^\star$ only if $(v, w) \in C^\star$. We proceed with the proof by first showing in Lemma 2 that $\pi^\star$ is consistent with the definition of $C^\star$, that is, $\pi^\star$ is not too coarse. We then show in Lemma 3 that algorithm *Partition* is optimal in that $\pi^\star$ is the coarsest consistent partition.

Lemma 2 (Consistency). *Partition $\pi^\star$ is consistent with the definition of $C^\star$, for if v is a vertex in a tree T in $\pi^\star$ and v is not the root vertex of T then all successors of v are also in T.*

Proof. Assume v is a vertex in a tree T in $\pi^\star$ that is not the root vertex of T. Then v has one successor *parent*(v) in T. Assume on the contrary to the claim that v has another successor w not in T. In the initial partition π, vertex v is in some tree $T_1 \supseteq T$ and all trees are initially placed in *worklist.* The construction of *splitlist* in line (5) implies that w must also be in T_1 since otherwise a split during the first iteration would have separated vertex v from *parent*(v) contradicting the assumptions. Now, consider the point during the algorithm at which vertex w is separated from the vertices v and *parent*(v) and the vertices are placed in two different trees $T_2 \subseteq T_1$ containing w and $T_2' \subseteq T_1$ containing v and *parent*(v). After this separation at least one of T_2 and T_2' will be in *worklist*, which implies that vertex v will be separated from *parent*(v) after the new contents of *worklist* are exhausted, which again contradicts the assumptions. Hence, all successors of v must be in T. □

Lemma 3 (Optimality). *Partition $\pi^\star$ is as coarse as possible, that is, if $(v, w) \in C^\star$ then v and w are in the same tree in partition $\pi^\star$.*

Proof. We show by induction on the number i of *split* operations performed in algorithm *Partition* that two vertices v and w are in two distinct trees only if $(v, w) \notin C^\star$. (Basis $i = 0$) The claims holds for the initial partition by Lemma

1. (Ind. $i > 0$) Let π be the partition resulting after $i-1$ split operations. The i-th split operation splits an edge (v, w) in some tree T only if v has another successor u in a different tree and by induction hypothesis: $(u, w) \notin C^\star$ and $(u, root(T)) \notin C^\star$. Hence, by rule (2) of Definition 1: $(v, w) \notin C^\star$ and also $(v, root(T)) \notin C^\star$. Let T_1 be the subtree of T rooted at v and let T_2 be the remaining portion of T after disconnecting T_1. Since the root vertices of the two trees, v and $root(T)$, are not congruent under $C^\star$, an analogous induction argument to the one in the proof of Lemma 1 shows that no vertex in T_1 can be congruent to a vertex in T_2 under $C^\star$. Thus, two vertices are in different trees in the new partition only if they are not congruent under $C^\star$. □

Corollary 1. *Algorithm Partition correctly computes the idempotence congruence relation $C^\star$ (by lemmas 2 and 3).*

Theorem 1 (Complexity). *Algorithm Partition can be implemented in $O(n \log n)$ time and $O(n)$ space, where n is the number of vertices in the equation graph G.*

Proof. Constructing the initial partition takes $O(n)$ time. To calculate the total time spent in the while loop, we consider the number of times the tree of each vertex can be placed in *worklist*. Each time the current tree of a vertex w is added to *worklist* the tree's size is at most half the size of the previous tree containing w. Hence, a vertex' tree can be at most $\log n + 1$ times in *worklist*. *Splitlist* is constructed by a scan of the vertices whose tree was removed from *worklist* and the total number of vertices scanned is $O(n \log n)$. Operation *split* is executed at most n times, since there can be at most n partitions. Each call to *split* is implemented in $O(1)$ time by maintaining for each vertex a pointer to its position in the partition forest. To find the smaller of the two subtrees after a split in time proportional to the smaller tree (i.e., in total time $O(n \log n)$), the vertices in the two trees are counted by alternating between the trees after each vertex. The algorithm also requires a pointer for each vertex to its current partition tree, which is updated after each split only for the vertices of the smaller resulting tree. In summary, the total time spent in executing algorithm *Partition* is $O(n \log n)$. The size of no auxiliary data structure is more than $O(n)$ and $O(n)$ space is used to store the partition. □

If the equation graph is constructed as described in Section 2, the size n of the graph is linear in the size of the program. In data flow problems that are based on a product lattice L^V, such as constant propagation, the equation at each program point is a vector $x = (x_1, \ldots, x_V)$. In constant propagation there is a component x_i for each of V program variables. In general, it will be beneficial to break the vector equation x into a set of V components equations $x_1, \ldots, x_V$ in order to expose additional congruences. In this granularity, the size of the equation graph increases to $V \times n$.

5 Congruence by Common Subexpression

Additional reductions in an equation system can be achieved by extending our definition of congruence to capture redundancies that result from sources other than idempotence. In [DST80, NO80] congruence relations are defined based on common subexpressions. For example, in Fig. 3 (ii), the term $x[8] \wedge (p, x)$ is a common subexpression in equations $x[9]$ and $x[10]$. The congruence relation by common subexpression is defined below by observing the commutativity of the meet operator.

Definition 2 (Congruence by common subexpression). *Let $G = (V, E)$ be an equation graph. A relation S on V is called common subexpression congruence relation if for vertices v and w with successors $v_1, \dots, v_k$ and $w_1, \dots, w_k$, $(v, w) \in S$ implies label(v) = label(w) and $\forall\ 1 \leq i \leq k$:*

$$\begin{cases} (v_i, w_{p(i)}) \in S \text{ for some permutation } p \text{ on } \{1, \dots k\} & \text{if } label(v) = \wedge \\ (v_i, w_i) \in S & \text{otherwise} \end{cases}$$

Partitioning a graph by common subexpression is a well known problem and a fast $O(n \log n)$ algorithm is due to Hopcroft's algorithm for minimizing finite automata [Hop71]. Among other applications, Hopcroft's algorithm was used to eliminate common subexpression in program optimization [AWZ88]. We present a different application by employing the algorithm to reduce data flow equation systems.

Hopcroft's algorithm starts with an initial partition π in which all vertices with an identical label are placed in the same congruence class in π. The algorithm iterates over the congruence classes to subsequently refine the current partition until it is consistent with Definition 2. The algorithm terminates with the coarsest partition in which two equations are in the same class only if they are congruent under S. An adaptation of Hopcroft's partitioning algorithm to partition equation graphs is shown below.

```
Algorithm An adaptation of Hopcroft's partitioning algorithm
Input:  Equation graph $G = (V = V_f \cup V_\wedge, E)$
Output: Partition $\pi^* = C_1, \dots, C_k$, where $C_i$ is a collection of vertices in $G$
1.  create an initial partition $\pi = C_1, \dots, C_l$ of the vertices in $V$ by their label;
2.  worklist $\leftarrow \{C_1, \dots, C_l\}$;
3.  while worklist $\neq \emptyset$ do
4.      select and remove $C_i$ from worklist;
5.      for $n \leftarrow 1$ to 2 do
6.          splitlist $\leftarrow$ $\{v \in V_f \mid$ the $n$-th succ. of $v$ is in $C_i\}$
7.                   $\cup$ $\{v \in V_\wedge \mid v$ has exactly $n$ succ. in $C_i\}$; /* commut. of $\wedge$ */
8.          for each $C_j$ such that $(splitlist \cap C_j) \neq \emptyset$ and $(C_j \not\subseteq splitlist)$ do
9.              create a new tree collection $C$ in $\pi$;
10.             move each $u \in (splitlist \cap C_j)$ to $C$;
12.             if $C_j \in$ worklist then add $C$ to worklist
13.                                   else add the smaller of $C_j$ and $C$ to worklist;
14.         endfor; endfor;
15. endwhile
```

If Hopcroft's algorithm is applied over the equation graph for the alias analysis of procedure Insert, the two equations $x[9] = (p, x) \wedge x[8]$ and $x[10] = x[8] \wedge (p, x)$ in Fig. 3 (i) are discovered to be congruent. The discovery of congruences due to common subexpressions may enable the detection of additional congruences by idempotence. For example, once we know that the two equations $x[9]$ and $x[10]$ are congruent, it can in turn be determined that equation $x[11] = x[9] \wedge x[10]$ is actually a hidden copy and in fact all three equations $x[9]$, $x[10]$ and $x[11]$ are congruent. To enable these second order effects, we can incorporate the results of common subexpression partitioning into the initial partition for idempotence partitioning. This is achieved by applying algorithm *Partition* to the equation graph that results if all vertices that were already found to be congruent are merged into a single vertex. The reductions in the equation system from Fig. 3 (i) that are enabled in this process are shown in Fig. 3 (iii). The additional improvements over the equations system that results from only partitioning by idempotence (Fig. 3 (ii)) are due to the discovery of the congruence among equations $x[9], x[10]$ and $x[11]$.

Unfortunately, applying each partitioning algorithm once may not provide optimal results. In general, congruences that are found based on idempotence may enable the discovery of additional common subexpressions and vice versa. Thus, to find the maximal number of congruences requires computing the transitive closure of the union of the two congruence relations. This closure can be computed by iterating over the two partitioning algorithms until no more congruence can be discovered. Each time a new iteration is started the size of the equation graph is reduced resulting in a bound of $O(n^2 \log n)$. In practice, the number of common subexpressions in an equation graph may be small, in which case, it may be sufficient to compute each congruence partitioning only once. While this may sacrifice optimality, equation system reduction remains fast. Experimentation is needed to determine the benefits of computing the iterated congruence closure.

6 Related Work

A number of previous methods has focused on suppressing some of the unnecessary equation evaluations by manipulating the underlying graphical program representation. The sparse evaluation graph (SEG) approach [CCF90], achieves reductions in data flow equation systems indirectly by specializing a program's control flow graph G with respect to each analysis problem such that smaller equation systems will be generated. The SEG is obtained from a control flow graph G by eliminating some of the nodes in G that have an identity flow function. The construction of a SEG requires $O(e+n^2)$ time using dominance frontiers [CCF90] and $O(e \times \alpha(e))$ time using a recent more complicated algorithm [CF93], where e is the number of edges in a program's control flow graph G and n is the number of nodes in G. The SEG approach compares directly with our idempotence congruence partitioning algorithm in that the removal of control flow graph nodes with identity flow functions results in the elimination of redundant (hidden) copy equations. However, there are important problems for which the SEG approach fails to eliminate all (hidden) copies and algorithm *Partition* would construct strictly smaller equation systems. Constant propagation is an example of such a problem. It is likely in constant propagation that no flow graph nodes have an identity flow function, in which case the SEG would be identical to the original flow graph graph. However, even flow graph nodes with a non-identity flow function generate copy and hidden copy equations for all program variables that are not assigned a new value within that node. As our partitioning approach operates on the level of individual equation operations, these redundancies are exposed and can therefore be eliminated. In addition, congruence partitioning is, unlike the SEG approach, extensible to discover redundancies due to common subexpressions enabling further reductions in an equation system.

Other methods that improve data flow analysis by building specialized program graphs are applicable to only certain data flow problems. The *partitioned variable technique* [Zad84] constructs for each variable a simplified flow graph that enables a fast evaluation of the solution. However, this method is restricted to *partitionable* data flow problems that permit the analysis of each variable partition in isolation. The *global value graph* [RL77, RT82], *static single assignment form* (SSA) [SS70] and *dependence flow graphs* [JP93] are graphical representations that provide connections between definitions and uses of program variables. SSA form is constructed in $O(e + n^2)$ time based on dominance frontiers [CFR+91] and in $O(e \times \alpha(e))$ time based on a recent algorithm [CF93]. The benefits of using SSA for data flow analysis are limited to problems that are based on definition-use connections, such as constant propagation [WZ85]. A problem like available expressions does not benefit from SSA. The same limitation applies to the related *dependence flow graphs* that are constructed in $O(V \times e)$ time, where V is the number of program variables.

Computing congruence relations based on common subexpressions is a well known problem and efficient algorithms have been developed [NO80, DST80, Hop71]. Hopcroft's partitioning algorithm for minimizing finite automata was used in program optimization to detect equalities among variables based on common subexpressions over an extended SSA form of the program [AWZ88]. The authors describe a strategy to manipulate the SSA representation in order to combine congruent (equal) variables values from different branches of a structured if-statement. This treatment can be viewed as handling a special case of detecting idempotence congruences. Other methods to eliminate redundant program computations include value numbering [CS70], global value numbering based on SSA form [RWZ88] and methods based on the global value graph [RT82].

7 Conclusion

We presented a new and efficient approach to improve the performance of any monotone data flow analysis by reducing the size of data flow equation systems through congruence partitioning. The presented partitioning algorithms discover congruences among data flow equations by exploiting the algebraic properties of idempotence and commutativity of the meet operator. A remaining property of the meet that we have not discussed is associativity. Unfortunately, discovering congruences that are due to associativity is a much harder problem. The difficulty of discovering congruences by associativity results from the fact that an exponential number of different sequences of meet operations can yield congruent values by associativity. This problem with associative operators also arises in program optimizations, where *reassociation* techniques have been used as a heuristic to discover certain equalities by associativity [CM80]. We are currently considering whether reassociation would be a suitable approach to enable further reductions in data flow equation systems.

Our approach of congruence partitioning demonstrates the feasibility of applying principles of program optimization and analysis, such as common subexpression elimination, to optimize the analyzers themselves. We expect to investigate this issue further as part of our future work.

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, principles, techniques, and tools.* Addison-Wesley Publishing Company, Massachusetts, 1986.

[AWZ88] B. Alpern, M. Wegman, and F.K. Zadeck. Detecting equality of values in programs. In *Proc. 15th Annual ACM Symp. on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[CC77] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *Proc. ACM Conf. on Language Design for Reliable Software*, pages 77–93, Raleigh, North Carolina, March 1977.

[CCF90] J.D. Choi, R.K. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conf. Rec. 18th Annual ACM Symp. on Principles of Programming Languages*, pages 55–66, Orlando, Florida, January 1990.

[CF93] R.K. Cytron and J. Ferrante. Efficiently computing ϕ-nodes on-the-fly. *Proc. 1993 Workshop on Languages and Compilers for Parallelism*, 1993.

[CFR$^+$91] R.K. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions of Programming Languages and Systems*, 13(4):451–490, October 1991.

[CLZ86] R.K. Cytron, A. Lowry, and F.K. Zadeck. Code motion of control structures in high-level languages. In *Conf. Rec. 13th Annual ACM Symp. on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, Florida, January 1986.

[CM80] J. Cocke and P.W. Markstein. Measurement of program improvement algorithms. In *Proc. Information Processing 80.* North Holland Publishing Company, 1980.

[CS70] J. Cocke and J.T. Schwartz. Programming languages and their compilers; preliminary notes. Courant Institute of Mathematical Sciences, New York University, April 1970.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* Freeman and Company, New York, 1979.

[Hop71] J.E. Hopcroft. An n log n algorithm for minimizing states in finite automata. In *Theory of Machines and Computations.* Academic Press, New York, 1971.

[JP93] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proc. ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 78–89, Albuquerque, New Mexico, June 1993.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, July 1977.

[Lan92] W.A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, New Brunswick, New Jersey, 1992.

[NO80] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closures. *Journal of the ACM*, 27(2), April 1980.

[RL77] J. Reif and J. Lewis. Symbolic evaluation and the global value graph. In *Conf. Rec. 4th Annual ACM Symp. on Principles of Programming Languages*, pages 104–118, January 1977.

[RT82] J. Reif and R.E. Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, 11(1):81–93, February 1982.

[RWZ88] B. Rosen, M. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. 15th Annual ACM Symp. on Principles of Programming Languages*, pages 12–27, San Diego, California, January 1988.

[SS70] R.M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, 1970.

[WZ85] M. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *Conf. Rec. 12th Annual ACM Symp. on Principles of Programming Languages*, pages 291–299, New Orleans, Louisiana, January 1985.

[Zad84] F.K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proc. ACM SIGPLAN Symp. on Compiler Construction*, pages 132–143, June 1984.