Abstract

As recent studies show, state-of-the-art parallelizing compilers produce no noticeable speedup for 9 out of 12 PERFECT benchmark codes, while the speedup that was reached by manually applying certain automatable techniques ranges from 10 to 50. In this paper we introduce the Global Value Propagation algorithm that unifies several of these techniques.

Global propagation is performed using program abstraction called Value Flow Graph (VFG). VFG is an acyclic graph in which vertices and arcs are parametrically specified using F-relations. The distinctive features of our propagation algorithm are: (1) It propagates not only values carried by scalar variables, but also values carried by individual array elements. (2) We do not have to transform a program in order to use propagation results in program analysis.

In this paper we focus on use of the VFG and global value propagation in array dataflow analysis. F-relations are used to represent values produced by uninterpreted function symbols that appear in dependence problems for non-affine program fragments. Global value propagation helps us to discover that some of these functions are in fact affine.

This work is supported by an NSF PYI grant CCR-9157834 and by a Packard Fellowship.
Global Value Propagation Through Value Flow Graph and Its Use in Dependence Analysis

Vadim Maslov
Computer Science Department
University of Maryland, College Park, MD 20742
vadik@cs.umd.edu, (301) 405-2726, fax (301) 405-6707

May 5, 1994

Abstract

As recent studies show, state-of-the-art parallelizing compilers produce no noticeable speedup for 9 out of 12 PERFECT benchmark codes, while the speedup that was reached by manually applying certain automatable techniques ranges from 10 to 50. In this paper we introduce the Global Value Propagation algorithm that unifies several of these techniques.

Global propagation is performed using program abstraction called Value Flow Graph (VFG). VFG is an acyclic graph in which vertices and arcs are parametrically specified using F-relations. The distinctive features of our propagation algorithm are: (1) It propagates not only values carried by scalar variables, but also values carried by individual array elements. (2) We do not have to transform a program in order to use propagation results in program analysis.

In this paper we focus on use of the VFG and global value propagation in array dataflow analysis. F-relations are used to represent values produced by uninterpreted function symbols that appear in dependence problems for non-affine program fragments. Global value propagation helps us to discover that some of these functions are in fact affine.

1 Introduction

Automatic parallelization of the real Fortran programs does not live up to user expectations yet. As the recent studies [Blu92, BE94] show, state-of-the-art parallelizing compilers produce no noticeable speedup for 9 out of 12 PERFECT benchmark codes, while the speedup that was reached by manually applying certain automatable techniques (techniques that can be implemented in a compiler) ranges from 10 to 50.

Several of these important techniques are special cases of the Global Value Propagation that we introduce in this paper. The basic idea of our approach is to compute Value Flow Graph (VFG) for a given program fragment and then to perform global value propagation using this graph. Each vertex of VFG is a single statement instance and set of all vertices forms iteration space of the program. There is an arc from vertex $a$ to vertex $b$ if statement instance $a$ directly passes value to statement instance $b$.

The main distinctive features of the Value Flow Graph are: (1) It is acyclic graph because every statement instance is executed only once, (2) It is parametrized graph, because number of vertices and arcs in VFG is not known statically, it is a parameter of a program, (3) For affine program fragments (fragments in which all subscript functions, IF conditions and loop bounds are affine functions of loop variables and symbolic constants) we can compute exact VFG, that is, VFG in
DO i = 1, Mb
  DO j = 1, i
  S1:  Lmin = j
  S2:  Lmax = i
  DO k = i, Mb
  S3:  DO l = Lmin, Lmax
  S4:  XKL(l) = ...
      END DO
      ...
  S5:  Lmin = 1
  S6:  Lmax = k + 1
  END DO
      ...
  END DO
  END DO

Figure 1: Fragment of OLDA of TRFD

which for every argument of every statement instance we know coordinates of just one statement instance that supplies the values used by this argument. Value Flow Graph for non-affine program fragments is only approximate.

We represent VFG using $F$-relations (functional relations). The formal definition of $F$-relation is given in Section 2. In Section 3 we present algorithm that computes $F$-relations for given program fragment. Since existing graph algorithms do not work on parametrized graphs directly, we introduce the Characteristic Graph (CG) that serves as compact representation of VFG. Characteristic graph has a fixed number of vertices, but there’s a price to pay — CG may have cycles.

The three algorithms presented in Section 4 perform global value propagation with varying degree of aggressiveness and performance. The main distinctive features of these algorithms are: (1) They propagate not only values carried by scalar variables, but also values carried by individual array elements. (2) They change VFG and characteristic graph of a program but we do not have to transform a program in order to use global propagation results in program analysis.

In this paper we focus on use of $F$-relations and global value propagation in array dataflow analysis. In Section 5 we introduce an extension of the Lazy Array Dataflow Analysis Algorithm [Mas94] that uses $F$-relations to represent values produced by uninterpreted function symbols that appear in dependence problems for non-affine program fragments. Global value propagation helps us to discover that many of these uninterpreted functions are in fact affine functions of loop variables. This makes dependence problem affine and therefore increases precision of dependence analysis for non-affine program fragments.

In the remaining part of introduction we consider examples of the real Fortran programs from the PERFECT benchmark suite [B+89] and show how use of $F$-relations and value propagation makes it possible to compute exact dependence information for them.

1.1 Propagating values of scalar variables

In the program fragment in Figure 1 the variables $L_{min}$ and $L_{max}$ are assigned within the $k$ loop. Since these variables are used as lower and upper bounds for loop $1$, the existing systems cannot determine statically what elements of the array $XKL$ are being written and therefore they cannot parallelize loops $i$, $j$ and $k$.

However, it is not difficult to see that $L_{min}$ and $L_{max}$ are piecewise-affine functions of the loop
variables \(i, j\) and \(k\) that can be expressed as \(F\)-relations (1) and (2). Substituting these functions to the non-affine set of constraints (3) that describes execution conditions for the statement \(S_4\), we discover that these constraints become affine constraints (4). Having affine execution conditions we can parallelize loop \(l\) if other criteria are satisfied.

The generalized induction variable recognition techniques [Wol92, HP92] can recognize that \(L_{\text{min}}\) and \(L_{\text{max}}\) are wrap-around variables. However, the existing systems have to transform the program in order to use this information in dependence analysis. In this example they have to peel off the first iteration of loop \(k\). We think that program analysis techniques that need to transform the program should be avoided for several reasons:

- User of parallelizing environment expects that when the environment is asked to analyze a program, it does not change the program.
- Analysis-enabling transformations are not justified by anything except needs of the dependence analyzer. As a result, in a number of cases they can cause program slowdown and increase program size considerably.
- These transformations are not always possible. For example, in Figure 2 no transformation can make dependence analyzer to believe that periodic variables \(I_1\) and \(I_2\) are affine functions of loop variable \(l\). However, using \(F\)-relations (5) and (6) that describe values of \(I_1\) and \(I_2\) as affine functions of \(l\) we can compute exact \(F\)-relation (7) for statement \(S_3\).

So we use techniques [Wol92, HP92] to compute closed form of generalized scalar induction variables. However, we do not substitute these closed forms to the places in program where these variables are used. Instead we substitute \(F\)-relation that expresses their closed form to the dependence problem involving references to these variables.

To compute closed form of array references we need to employ more general techniques of [RF93]. However, none of the benchmark studies yet claimed that recognizing array reductions may be useful for dependence analysis. So we restrict us to recognizing scalar reductions only.

1.2 Propagating values of array elements

Consider a program fragment in Figure 3. Suppose we should run the loop nest \(k\) (statements from \(S_8\) to \(S_{12}\)) on distributed memory machine and the memory distribution is such that \(XY(j,k,4)\) is aligned with \(X(j,k)\). For statement \(S_{10}\) the existing systems cannot generate efficient communications code, because they do not know at compile time that the value of scalar variable \(J_m\) (and therefore, value of array element \(J\text{MINU}(j)\)) used in subscript of array \(X\) is affine function of \(j\).

Using the global value propagation, we compute \(F\)-relation (9) that makes it clear that for all
The smallest unit of computation we consider in this paper is a program fragment (APF). An affine program fragment (APF) is a body of procedure or a body of one of the loops of procedure. APF consists of assignment statements, structured IF statements and DO loop statements (GOTO statements are not allowed) such that in every statement all (1) subscript functions, (2) conditions in IF statements, and (3) loop bounds should be explicit affine functions of loop variables and symbolic constants assigned outside the fragment (that is, they have form $c_0 + \sum_{i=1}^{n} c_i x_i$, where $c_i$ are integer constants and $c_i$ are variables). If either of these requirements is not satisfied, the program fragment is called non-affine.

The smaller the program fragment, the more likely it will be affine. For example, in program in Figure 1 the body of the loop 1 is affine program fragment, but the body of the loop k is non-affine fragment, because lower and upper bounds of the loop 1 are assigned within the fragment and therefore they are not explicit affine functions of $k$ and $l$.

**2 Definitions**

**Affine program fragment.** An affine program fragment (APF) is a body of procedure or a body of one of the loops of procedure. APF consists of assignment statements, structured IF statements and DO loop statements (GOTO statements are not allowed) such that in every statement all (1) subscript functions, (2) conditions in IF statements, and (3) loop bounds should be explicit affine functions of loop variables and symbolic constants assigned outside the fragment (that is, they have form $c_0 + \sum_{i=1}^{n} c_i x_i$, where $c_i$ are integer constants and $c_i$ are variables). If either of these requirements is not satisfied, the program fragment is called non-affine.

The smaller the program fragment, the more likely it will be affine. For example, in program in Figure 1 the body of the loop 1 is affine program fragment, but the body of the loop k is non-affine fragment, because lower and upper bounds of the loop 1 are assigned within the fragment and therefore they are not explicit affine functions of $k$ and $l$.

**Tuples and Statement instances.** Tuple is simply an ordered set of integers. Tuples are denoted with bold letters, such as $w, r, s$. Tuple of length $n$ represents a point in $n$-dimensional space.

The smallest unit of computation we consider in this paper is statement instance. The statement instance $S[v]$ is specified by $S$ — statement of the program, and $v$ — tuple of loop variables values (loops that surround the statement $S$ are included). For example, in Figure 1 statement $S_i$ has
F/-relation instance

<table>
<thead>
<tr>
<th>S[i, j]</th>
<th>= 2i−j−2</th>
<th>1 ≤ i, j ≤ N</th>
<th>S[i]</th>
<th>= In(A(2 N−i))</th>
<th>1 ≤ i ≤ N</th>
<th>S[i, j]</th>
<th>= S[i] + 2S3[i, j]</th>
<th>1 ≤ i ≤ j ≤ N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out.B(i, i)</td>
<td>= S2[i]</td>
<td>1 ≤ i ≤ N</td>
<td>S3[5, 7]</td>
<td>= 1</td>
<td></td>
<td>S2(1)</td>
<td>= In.A(2)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Examples of F-relations and their instances

In affine fragments that we consider DO loops and IF statements are used to shape the iteration space, and actual computations are performed exclusively by assignment statements. Therefore we call assignment statement instances simply “statement instances”.

**F-relations.**  Functional relation (or F-relation) is an extension of the concept of dependence relation introduced in [Pug91]. F-relation is a union of one or more simple F-relations. Mathematically, simple F-relation is a parametrized set of equalities that define statement instances in the left hand side of relation as a function $F$ of statement instances, initial memory reads and constants in the right hand side of the relation. General form of simple F-relation is:

$$S[w] = F(S[i_1][r_1], ..., S[i_m][r_m]; \text{In} A_{j_1}(s_1), ..., \text{In} A_{j_n}(s_n); r) \mid \ p(r_1, ..., r_m; s_1, ..., s_n; r; w)$$

where $p$ is a conjunction of affine constraints. Examples of F-relations and their instances are given in Figure 4. The semantic meaning of this simple F-relation is:

for all $r_1, ..., r_m; s_1, ..., s_n; r; w$: if $p(r_1, ..., r_m; s_1, ..., s_n; r; w)$ is True then statement instance $S[w]$ is a result of application of function $F$ to the values of statement instances $S[i_1][r_1], ..., S[i_m][r_m]$, values of initial memory cells $\text{In} A_{j_1}(s_1), ..., \text{In} A_{j_n}(s_n)$ and value of tuple $r$.

Terms in the right hand side of F-relation are called arguments of F-relation. Arguments specify the sources of values consumed by F-relation. There are three types of arguments:

- $r$: constant. Consume value of a constant.
- $\text{In} A_{j_i}(s_i)$: memory read. Consume value of memory cell that it had before execution of the affine program fragment represented by F-relation.
- $S[i][r_i]$: statement instance. Consume value computed by another statement instance.

Values computed by F-relation can be used in two ways:

- $S[w] = x$: Value of $x$ becomes a value of statement instance $S[w]$ (to be consumed by another statement instance).
- $\text{Out} A_{k_i}(s_i) = x$: Value of $x$ is written to a memory cell that is used after execution of the affine program fragment represented by F-relation.

We can represent input and output statements as reads from and writes to global file memory, so F-relations are sophisticated enough to represent real programs.

**Correctness properties of F-relations.** F-relation $R$ should satisfy certain criteria to be correct:

\[
\frac{Mb(Mb+1)}{2} \text{ instances } S[i, j] \mid 1 \leq j \leq Mb.
\]

Correctness properties of F-relations.
1. There should be no contradictory definitions for every statement instance. That is, if some statement instance $S[v]$ is defined twice: $S[v] = F(ArgList_1)$ and $S[v] = G(ArgList_2)$, then $F = G$ and $ArgList_1 = ArgList_2$.

2. There should be no cycles. That is, for every statement instance $S[v]$ there should not exist a chain of simple F-relation instances that defines $S[v]$ as a function of itself.

3. In F-relation $R$ that represents a complete affine program fragment, then all statement instances should be defined. That is, for every statement instance $S[v]$ that appears as an argument of some simple F-relation from $R$ there should exist a definition of $S[v]$ in $R$.

**Operations over F-relations.**

**Domain($R$):** Domain of F-relation $R$.

This is a set of statement instances consumed by F-relation $R$:

$$\text{Domain}(R) = \{S_i[r_1] \cup \cdots \cup S_m[r_m] | \pi_{r_1,\ldots,r_m}(p(r_1,\ldots,r_m; s_1,\ldots,s_n; r; w))\}$$

**Domain($R, S_i[r_i]$):** Domain of F-relation $R$ with respect to its argument $S_i[r_i]$.

This is a set of statement instances consumed by reference $S_i[r_i]$ of F-relation $R$:

$$\text{Domain}(R, S_i[r_i]) = \{S_i[r_i] | \pi_r(p(r_1,\ldots,r_m; s_1,\ldots,s_n; r; w))\}$$

**Range($R$):** Range of F-relation $R$.

This is a set of statement instances produced by relation $R$:

$$\text{Range}(R) = \{S[w] | \pi_w(p(r_1,\ldots,r_m; s_1,\ldots,s_n; r; w))\}$$

**Value Flow Graph (VFG): geometric interpretation of F-relation.** F-relation that represents an affine program fragment has an elegant geometric interpretation called Value Flow Graph. We build VFG for the F-relation $R$ in the following way:

- For every statement instance $S[v] \in \text{Range}(R)$, we create a vertex $S[v]$ in VFG.
- If statement instance $S_1[v_1]$ is an argument of F-relation instance that computes $S_2[v_2]$, (that is, if $S_1[v_1]$ directly passes value to $S_2[v_2]$) we create a directed arc $(S_1[v_1], S_2[v_2])$ in VFG.

Value Flow Graph is a directed acyclic parametrized graph that exactly describes flow of values in affine program fragment. Actually, VFG of program fragment $P$ or corresponding F-relation can be viewed as a parametrically specified function from memory before execution of $P$ to memory after execution of $P$.

Since every statement instance stores only one value and it does only once in execution of program $P$, Value Flow Graph is memoryless program representation. Indeed, VFG specifies what statement instances pass values to what statement instances, but it does not specify what memory cells are used for intermediate storing of these values.

**Machinery used: Presburger arithmetic solver.** In definition of F-relation the problem $p(r_1,\ldots,r_m; s_1,\ldots,s_n; r; w)$ is a conjunction of affine equalities and inequalities over integers. Our algorithms require the following operations to be performed on problems like $p$: $p_1 \land p_2$, $p_1 \lor p_2$, $\neg p$, $\exists v s.t. p \lor v : p$. These operations produce Presburger arithmetic formulas. We simplify these formulas to *Disjunctive Normal Form (DNF)* using the Omega test [Pug92, PW93].
Often instead of \( \exists \) we use more convenient projection operator introduced in [Pug92]:

\[
\pi_v(P(v, w)) = \exists w \text{ s.t. } P(v, w)
\]

### 3 Computing VFG for affine fragments

In this section we present algorithm to compute Value Flow Graph for a given affine program fragment:

1. For every assignment statement

   \[ S : a = F(b_1, b_2, ..., b_n; v) \]

   where \( v \) is a tuple of loop variables surrounding \( S \) and \( b_1, ..., b_n \) are array or scalar references, create simple \( F \)-relation

   \[ S[v] = F(S_{i_1}[v], S_{i_2}[v], ..., S_{i_m}[v]; v) \mid S.IsExecuted(v) \]

   Please note that in addition to instances of statement \( S \) we create statement instances for each argument of \( F \) that is a reference to scalar variable or array.

2. Compute exact value-based dependences for each read reference using algorithm [Mas94]. These dependences are expressed as dependence relations [Pug91]:

   \[ S_x[w] \rightarrow S_{i_j}[v] \mid p(w, v) \]

   Convert each dependence relation to equivalent \( F \)-relation:

   \[ S_{i_j}[v] = S_x[w] \mid p(w, v) \]

3. Propagate all assignments for statement instances \( S_{i_j}[v] \) to the statement where they are used, that is, to the statement \( S \). Since each \( S_{i_j}[v] \) is used only in \( F \)-relation for \( S \), this propagation is simple.

This simple algorithm just translates dependence relations computed for affine program fragments to \( F \)-relations. The more sophisticated dependence analysis algorithm introduced in Section 5 uses global propagation to compute exact dependence information for certain non-affine programs. For example, for program in Figure 5 [RF93] \( F \)-relation (11) is computed.

### 4 Global Value Propagation

#### 4.1 Characteristic Graph

Since number of vertices in Value Flow Graph is not known at compile time, regular graph manipulation techniques do not readily apply to VFGs. To make Value Flow Graph more manageable we build Characteristic Graph (CG) that represents VFG. \( F \)-relation \( R \) that consists of \( n \) simple \( F \)-relations \( R_1, ..., R_n \) is represented by CG constructed in the following way:

\( ^1 \)Number of vertices in VFG is finite, but since this number is not known at compile time, it can be arbitrarily large. However, we think that parametrized graphs that we consider are not infinite graphs mentioned in graph theory.

---

---
Single act of propagation

Therefore CG of this program has 5 vertices. Characteristic graph arcs are listed in (1).

There are 2 cycles in the CG: (R₃, R₅) and (R₃, R₅, R₅).

4.2 Single act of propagation

Function Propagate presented in Figure 6 propagates statement instances computed by simple F-relation R₁ to argument S₁[v₂] of simple F-relation R₂. The function checks that transfer set from
SafePropagation(F-relation $R$) Begin

DoWhile (exists simple F-relations $R_1, R_2 \in R$ and argument $S[v]$ of $R_2$ such that $\text{Transfer}(R_1, R_2, S[v]) = \text{Domain}(R_2, S[v])$)

Propagate($R_1, R_2, S[v]$)

EndWhile

End

Figure 7: Safe propagation algorithm that avoids splintering

$R_1$ to $R_2, S_1[v_2]$ is not empty. Otherwise there is nothing to propagate.

Basically this function replaces instances of $S_1[v_1]$ with expressions from $R_1$ that compute their values. Since $R_1$ does not necessarily compute values for all instances of $S_1[v_2]$ used in $R_2$, some instances of unpropagated $R_2$ (denoted as $R''_2$) are left intact. Since the computation of $R''_2$ involves negation, $R''_2$ can contain more than one simple F-relations.

Let’s consider example in which the algorithm is asked to propagate simple F-relation $R_2$ to the argument $S_2[j_2]$ of simple F-relation $R_5$ (both simple F-relations are part of (11)):

$$R_2: S_2[i_1] = \text{In} \cdot X(i_2) \quad |1 \leq i_1 \leq N \land i_2 = 2N - i_1 + 1$$

$$R_5: S_3[j_1] = S_3[j_3] + S_2[j_2] \quad |2 \leq j_1 = j_2 \leq 2N \land j_3 = j_2 - 1$$

We find that $\text{Transfer}(R_2, R_5, S_2[j_2]) = \{S_2[i] | 2 \leq i \leq N\}$ is not empty and then the result is:

$$R'_2: S_3[j_1] = \text{In} \cdot X(i_2) \quad |1 \leq i_1 \leq N \land i_2 = 2N - i_1 + 1$$

$$R''_2: S_3[j_1] = \text{In} \cdot X(i_2) + S_3[j_3] \quad |2 \leq j_1 \leq 2N \land i_2 = 2N - i_1 + 1 \land j_3 = j_2 - 1$$

$$R''''_2: S_3[j_1] = S_2[j_2] + S_3[j_3] \quad |N + 1 \leq j_1 = j_2 \leq 2N \land j_3 = j_2 - 1$$

4.3 Value Propagation algorithms

In this section we present several algorithms that perform value propagation for the whole program fragment. The algorithms differ in performance and aggressiveness. The propagation is done as a series of invocations of Propagate function.

Propagating values in VFG is not simple. Simply invoking Propagate for every arc in the characteristic graph may result in infinite sequence of substitutions. For example, it happens in the following sequence of substitutions:

$$R_1: S[1] = x_0$$
$$R_2: S[i] = F(S[i-1]) \quad |2 \leq i \leq N \Rightarrow R'_1: S[1] = x_0$$
$$R_3: S[i] = F(S[i-1]) \quad |3 \leq i \leq N \Rightarrow R''_1: S[2] = F(x_0)$$
$$R_4: S[i] = F(S[i-1]) \quad |4 \leq i \leq N \Rightarrow R'''_1: S[3] = F(F(x_0))$$

Our algorithms avoid this problem.

The safe propagation algorithm presented in Figure 7 propagates values only along the characteristic graph arcs that alone carry all the value instances consumed by argument of F-relation. This guarantees that consumer F-relation will not splinter as a result of propagation. Therefore every single act of propagation never increases number of vertices in CG. Actually the number of vertices may decrease if producer F-relation becomes unused after propagation. Since with every propagation values move closer to the place where they are consumed and the number of vertices in CG does not increase, the safe algorithm is guaranteed to terminate. The safe algorithm can be used in any application, however it can miss some important propagations.
ConstantPropagation(F-relation $R$) Begin
    Find Strongly Connected Components (SCCs) in CG using Tarjan algorithm [Tar72]
    For (component $C$ in SCCs of characteristic graph in topological order)
        If ($C$ is a single vertex $x$ such that no $(x,x)$ self-arcs exist) then
            For ($y$ in immediate successors of $x$)
                If ($y$ is not involved in a cycle) Propagate($x,y$)
            EndFor
        EndIf
    EndFor
End

Figure 8: Aggressive constant propagation algorithm

HeuristicPropagation(F-relation $R$) Begin
    ArcSet WorkSet := all arcs of characteristic graph for F-relation $R$
    While (WorkSet is not empty)
        Arc $(x,y)$ := remove an arc from WorkSet
        If (Transfer($R_1$, $R_2$, $S[v]$) = Domain($R_2$, $S[v]$)) then
            Propagate($x,y$)
            Add to WorkSet new arcs that appeared in CG due to propagation
        ElseIf ($(x,y)$ is not self-arc) then
            Try to Propagate($x,y$)
            If (cost function of $R$ decreases with this propagation) then
                Commit this propagation
                Add to WorkSet new arcs that appeared in CG due to propagation
            Else
                Undo this propagation
            EndIf
        EndIf
    EndWhile
End

Figure 9: Heuristic propagation algorithm

The safe algorithm completely avoids splintering. However, if splintering does not lead to infinite sequence of substitutions, it may be useful, because it allows deeper propagation. Since this paper is focused on making dependence analysis more precise and propagation of parametrized constant values makes this happen, we developed aggressive constant propagation algorithm presented in Figure 8. This algorithm does not propagate values to the vertices inside strongly connected components to avoid infinite splintering, it only propagates values from the characteristic graph constant leaves to the consumer vertices not involved in cycles. Since vertices involved in cycles never splinter, number of vertices in CG can increase only by a finite number.

Heuristic propagation algorithm presented in Figure 9 combines some aggressiveness of constant propagation algorithm and cautiousness of safe propagation algorithm. It always performs safe propagations first. Then it uses heuristic cost function to decide whether unsafe propagation makes the characteristic graph better. Since every single act of propagation is allowed only to decrease the cost of graph, the algorithm is guaranteed to terminate and it will produce CG that is better
than original characteristic graph. Currently the cost function is “number of vertices in CG plus maximum length of a cycle in CG”.

We think that the heuristic algorithm should not be used in dependence analysis, but it is suitable for applications that require “deep” propagation. For instance, it can be used in generalized recurrences recognition [RF93].

**Heuristic Propagation example.** The heuristic algorithm performs the following steps when working on program in Figure 5. First, in (11) we try safe propagations: we propagate $R_1$ to $R_4, S_1[1]$ and $R_2$ to $R_4, S_2[1]$. We also try to propagate $R_2$ to $R_5, S_2[i]$. This propagation proves to be beneficial because while number of simple F-relations stays the same, the cycle $(R_3, R_5, R_3)$ of length 3 is broken into two cycles $(R_5, R_3)$ and $(R_2', R_3')$, each of unit length. The result of these propagations is F-relation (13).

On next iteration we perform safe propagation of $R_3$ to $R_3', S_2[i]$. We also try to propagate $R_4' = R_3', S_3[-1]$ but this results only in increase of number of CG nodes, so we undo this propagation. Finally we get F-relation (14) which is simpler than the original F-relation (11).

### 5 Array dataflow dependence analysis using VFG

In Figure 10 we present an extension of Lazy Array Dataflow Dependence Analysis Algorithm [Mas94]. The basic idea of this extension is to substitute to non-affine constraint values of non-affine references obtained by global value propagation. This substitution often makes the constraint affine, and therefore we stay in domain of exact dependence analysis. The numbered lines of algorithm constitute the extension proposed in this paper and details of the rest of the algorithm are given in [Mas94].

So, let’s imagine that we build a dependence problem $DepProb$ that has references to scalar and array references $a_1, ..., a_q$ and these references are not explicit affine functions of loop variables and symbolic constants (line 10). We break this problem into two parts: first part ($F$) contains only non-affine constraints, second part ($L$) contains only affine constraints:

$$DepProb(v) = F(a_1(v), ..., a_q(v); v) \geq 0 \land L(v)$$

Then we compute F-relations for each non-affine reference (line 11). That is, we call the dependence analysis routine recursively and ask it to compute source function for references $a_1, ..., a_q$. To avoid recursive cycling, we memorize in stack all read references for which source function is being computed (line 3) and if the given reference is already in stack (line 2), it means that dependence problem for the reference includes the reference itself (like in Example 2 below). In this case we give up on propagation and compute an affine approximation of the non-affine dependence relation.

After we computed F-relations for references $a_1, ..., a_q$ and they are all affine, we create F-relation $R_D$ that represents values computed by function $F$:

$$R_D : S_D[v] = F(S_{a_1}[v], ..., S_{a_q}[v]) \mid L(v)$$

and propagate F-relations for $S_{a_1}[v], ..., S_{a_q}[v]$ to $R_D$. As a result of propagation $R_D$ can splinter.

If the resulting relation arguments are constants, then the original non-affine constraint can be converted to affine form. That is, when each simple F-relation $R_D'$ that is a member of $R_D$ after propagation has a form

$$R_D' : S_D[v] = F'(v, w) \mid L'(v, w)$$
Relation $\text{SourceFunction}(R.A)$ Begin

\textbf{Input:} $R.A$ is a read reference surrounded by $n$ loops with variables $r = (r_1, ..., r_n)$. 

(* Compute dependence relation that represents source function for $R.A$ *)

1: Static stack $\text{AlreadyInAnalysis}$
2: If ($\text{AlreadyInAnalysis}$ contains $R.A$) Return ($\text{NonAffine}$)
3: Push $R.A$ to stack $\text{AlreadyInAnalysis}$

Relation $\text{DepRel} := \{\emptyset\}$

$\text{Dnf NotCovered}(r) := \text{IsExecuted}(R[r])$

Statement $W := R$

While ($\text{NotCovered}$ is feasible) do

\begin{itemize}
  \item $W := \text{statement preceding statement } W$
  \item Statement $W$ is surrounded by $m$ loops with variables $w = (w_1, ..., w_m)$
  \item If ($W$ is assignment statement that writes to array of $R.A$) then
    \begin{itemize}
      \item Build dependence problem $\text{DepProb}(w, r)$ for dependence from $W$ to $R.A$
    \end{itemize}
  \end{itemize}

10: (at $i = 1$ to $q$) $R_i := \text{SourceFunction}(a_i)$

11: If (all of $R_i$ are affine) then

12: Create F-relation $R_F$ that represents values computed by $\text{DepProb}(w, r)$

13: Propagate to $R_D$ values carried by relations $R_1, ..., R_q$

14: If $\text{Domain}(R_D) = \{\emptyset\}$ then convert $\text{DepProb}$ to affine form

15: Else

16: Source function for $R.A$ is non-affine. Compute its affine approximation.

17: EndIf

18: $\text{DepRel} := \text{DepRel} \cup \text{Cmax}$

19: $\text{Cmax} := \text{RelMax}_{\leq}(W[w] \to R.A[r] | \text{DepProb}(w, r))$

20: Remove $R.A$ from top of stack $\text{AlreadyInAnalysis}$

Return ($\text{DepRel}$)

Figure 10: Lazy dependence analysis combined with global value propagation

where $F'$ is an affine function and $w$ is a tuple of variables added by propagation, then for each $R_D'$ we generate affine constraint

$$F'(v, w) \geq 0 \land L'(v, w)$$

The sum of generated constraints is equivalent to the original non-affine constraint.

**Example 1: constraint affinization using propagation.** Computing the source function for reference $S_3.IF(11)$ in Figure 2 we build the following dependence problem:

$$p(w, r) = (S_3.I1[w] = S_3.I2[r] \land 1 \leq w, r \leq \mathbb{N})$$

Since equality constraint is not affine, we compute F-relations (5) and (6) for references $S_3.I1$ and $S_3.I2$. Then we build F-relation for non-affine constraint:

$$S_D[w, r] = S_3.I1[w] - S_3.I2[r] \mid 1 \leq w, r \leq \mathbb{N}$$
\[
\text{DO } i = 1, N \\
S1: \quad A(A(i)) = x \\
\text{END DO}
\]

\[1 \leq i_w < i_r \leq N \wedge S_{i_r} . A(i))[i_w] = i_r \quad (16)\]

Figure 11: Propagation cycle example

and propagate F-relations (5) and (6) to this F-relation:

\[
\begin{align*}
S_D[l_w, l_r] &= -1 & 1 \leq l_w \leq l_r \leq N1 \wedge l_w = 2\alpha+1 \wedge l_r = 2\beta+1 \\
S_D[l_w, l_r] &= 0 & 1 \leq l_w \leq l_r \leq N1 \wedge l_w = 2\alpha \wedge l_r = 2\beta+1 \\
S_D[l_w, l_r] &= 0 & 1 \leq l_w \leq l_r \leq N1 \wedge l_w = 2\alpha+1 \wedge l_r = 2\beta+1 \\
S_D[l_w, l_r] &= 1 & 1 \leq l_w \leq l_r \leq N1 \wedge l_w = 2\alpha \wedge l_r = 2\beta
\end{align*}
\]

Since now \(S_D[l_w, l_r]\) has constant values only, we convert it back to constraint form \(S_D[l_w, l_r] = 0\), simplify and get:

\[
p(l_w, l_r) = (1 \leq l_w \leq l_r \leq N1 \wedge l_w = 2\alpha \wedge l_r = 2\beta+1) \lor \\
(1 \leq l_w \leq l_r \leq N1 \wedge l_w = 2\alpha+1 \wedge l_r = 2\beta+1)
\]

Computing lexicographical maximum \(\max_{\subseteq}(l_w \mid p(l_w, l_r))\) and simplifying we get dependence relation (7).

**Example 2: when propagation can cycle.** In a program fragment in Figure 11 dependence problem (16) constructed when computing source function for the read reference \(A(i)\) is non-affine. Moreover, the dependence problem refers to the source function it is computing. In this case we do not perform propagation, we just compute affine approximation of the source function.

### 6 Related Work

**Scalar program graph representations.** In recent years there has been a flurry of research activity in graph program representations. Static Single Assignment (SSA) form [CFR+91] and Program Dependence Graph (PDG) [FOW87] were introduced. They were followed by Program Dependence Web [BMO90], Dependence Flow Graph [JP93] and Value Dependence Graph [WCE94].

We call these graphs scalar program abstractions, because they are oriented towards representing data flow carried by scalar variables in program fragments without loops. When it comes to representing data flow in programs that have array references and loops, scalar abstractions essentially cease to be dataflow representations, because the array load and store operations appear in them and individual value path is not followed. Also when representing loops, scalar abstractions have to distinguish between data arcs and control arcs. Contrary to the scalar program abstractions, VFG has only one type of arcs, it does not have cycles, and it does not use memory for storing intermediate results.

**Exact array dataflow analysis techniques.** The concept of F-relation is based on a concept of dependence relation introduced in [Pug91]. The most difficult part of computing VFG is computing exact value-based dependence relations between statements. This part is done by array dataflow dependence analysis algorithms [F91, PW93, M94].
Voevodins work. Valentine and Vladimir Voevodin [Voe92a, Voe92b] use Algorithm Graph (AG) to represent data flow in affine programs. The Algorithm Graph is essentially equivalent to VFG and notation used for specifying Algorithm Graphs seems to be close to F-relations. However, authors do not formalize their notation. Also they do not discuss using the Algorithm Graph for global value propagation.

What’s interesting, they mention review by Yershov [Yer73] in which he writes about Program Implementation Dataflow Graph, not discussing, however, its properties and applications. This graph seems to be equivalent to both VFG and AG.

Feautrier and Redon work. Systems of Linear Recurrence Equations (SLRE) [RF93] are close to F-relations. However important details in definitions and algorithms differ.

First, [RF93] uses Quasi-Affine Search Trees (quasts) to represent SLREs while we use F-relations to represent VFGs. We refer reader to comparison of dependence relations and quasts in [PW93, Mas94], because this comparison is appropriate for F-relations and SLREs.

Second, in [RF93] the system graph that is analogue of our characteristic graph has an arc $(R_1, R_2)$ if some reference $x$ appears both in the left hand side of $R_1$ and in the right hand side of $R_2$, while we also require transfer set Transfer($R_1, R_2$) to be not empty. This additional requirement makes our characteristic graph more precise.

Our characteristic graph is more refined than system graph of [RF93] in other respects too. Consider example in Figure 12. Characteristic graph of the F-relation (17) has no cycles, while system graph of the equivalent SLRE (18) has a cycle $(Q_1, Q_1)$ that creates a false impression that there is an iterative computation going on at $Q_1$. We have more refined characteristic graph that allows deeper propagation, because we require conditions at the simple F-relation to be single conjunct, while in [RF93] conditions at one SLRE equation can be arbitrary disjunction of conjuncts.

Third, we think that [RF93] propagation algorithm is excessively cautious, because they do not allow splintering of the system graph nodes at all and their propagation condition (SLRE to be propagated should be used only in one other SLRE) is too stringent. As even a relatively small set of our examples (Figures 1, 2, 3) shows, in dependence analysis we need to perform propagation even if SLRE is used in two or more places, and [RF93] cannot do it.

7 Conclusion

In this paper we introduced Value Flow Graph that exactly represents flow of values in affine program fragment. We presented algorithms that (1) compute VFG, (2) propagate values through VFG and are not embarrassed by values carried across the loop iterations by array elements, (3) use results of global propagation to compute exact dependence information for many important
cases of non-affine programs.

We believe that Value Flow Graph can be used not only for enhancing dependence analysis, but also for (1) generalized recurrence recognition a la [RF93], (2) global dead code elimination, (3) global common subexpression elimination. Also we think that more experimentation is needed to measure the performance of the algorithms introduced in this paper and to find new areas of their applicability.

References


