UMIACS-TR-94-80 July, 1994
CS-TR-3310

Global Value Propagation Through Value Flow Graph and Its Use in Dependence Analysis

Vadim Maslov
vadik@cs.umd.edu

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, College Park, MD 20742

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-9157384 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 teh *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
```

$$\left[\begin{array}{lcl} S_3.\mathtt{Lmin}[i,j,k] = j & | & 1 \le j \le i = k \le \mathtt{Mb} \\ S_3.\mathtt{Lmin}[i,j,k] = 1 & | & 1 \le j \le i < k \le \mathtt{Mb} \end{array}\right. \qquad (1)$$

$$\left[\begin{array}{lcl} S_3.\mathtt{Lmax}[i,j,k] = i & | & 1 \le j \le i = k \le \mathtt{Mb} \\ S_3.\mathtt{Lmax}[i,j,k] = k{+}1 & | & 1 \le j \le i < k \le \mathtt{Mb} \end{array}\right. \qquad (2)$$

$$\begin{array}{l} 1 \le j \le i \le k \le \mathtt{Mb} \wedge \\ S_3.\mathtt{Lmin}[i,j,k] \le l \le S_3.\mathtt{Lmax}[i,j,k] \end{array} \qquad (3)$$

$$\begin{array}{l} 1 \le j \le l \le i = k \le \mathtt{Mb} \vee \\ 1 \le j \le i < k \le \mathtt{Mb} \ \wedge \ 1 \le l \le k{+}1 \end{array} \qquad (4)$$

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 `Lmin` and `Lmax` are assigned within the `k` loop. Since these variables are used as lower and upper bounds for loop `l`, 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 `Lmin` and `Lmax` are piecewise-affine functions of the loop

```
S0: I1 = 1
S1: IP(1) = X
    DO l = 1, N1
S2:   I2 = 3 - I1
      ...
S3:   IP(I2) = IP(I1)
      ...
S4:   I1 = I2
    END DO
```

$$\left[\begin{array}{ll} S_3.\mathtt{I1}[l] = 1 & \mid\ \exists\alpha\ s.t.\ l=2\alpha{+}1\ \wedge\ 1\le l\le \mathtt{N1} \\ S_3.\mathtt{I1}[l] = 2 & \mid\ \exists\alpha\ s.t.\ l=2\alpha\ \wedge\ 1\le l\le \mathtt{N1} \end{array}\right. \tag{5}$$

$$\left[\begin{array}{ll} S_3.\mathtt{I2}[l] = 2 & \mid\ \exists\alpha\ s.t.\ l=2\alpha{+}1\ \wedge\ 1\le l\le \mathtt{N1} \\ S_3.\mathtt{I2}[l] = 1 & \mid\ \exists\alpha\ s.t.\ l=2\alpha\ \wedge\ 1\le l\le \mathtt{N1} \end{array}\right. \tag{6}$$

$$\left[\begin{array}{ll} S_3.\mathtt{IP(I1)}[l] = S_3[l{-}1] & \mid\ 2\le l\le \mathtt{N1} \\ S_3.\mathtt{IP(I1)}[l] = S_1[] & \mid\ l=1 \end{array}\right. \tag{7}$$

Figure 2: Fragment of CFFT2D1 (simplified)

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 `Lmin` and `Lmax` 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 `I1` and `I2` are affine functions of loop variable `l`. However, using F-relations (5) and (6) that describe values of `I1` and `I2` 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 $\mathtt{XY}(j,k,4)$ is aligned with $\mathtt{X}(j,k)$. 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 `Jm` (and therefore, value of array element `JMINU(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

```
C--- subroutine INITIA ---
    DO j = 1, Jmax
S1:   JMINU(j) = j - 1
    END DO
    IF (.NOT.p) THEN
S2:   JMINU(1) = 1
S3:   Jlow = 2
S4:   Jup = Jmax - 1
    ELSE
S5:   JMINU(1) = Jmax
S6:   Jlow = 1
S7:   Jup = Jmax
    ENDIF
```

```
C--- subroutine XIDIF ---
    DO k = Kb, Ke
S8:   DO j = Jlow, Jup
S9:     Jm = JMINU(j)
S10:    XY(j,k,4) = X(Jm,k) + ...
      END DO
      IF (.NOT.p) THEN
S11:    XY(1,    k,4) = X(1,k) + ...
S12:    XY(Jmax,k,4) = X(Jmax,k) + ...
      ENDIF
    END DO
```

$$j_w = 1 \;\wedge\; k_w = k_r \;\wedge\; \mathtt{Kb} \le k_w, k_r \le \mathtt{Ke} \;\wedge\; S_8.\mathtt{Jlow}[k_w] \le j_w \le S_8.\mathtt{Jup}[k_w] \;\wedge\; \neg p \quad (8)$$

$$\left[\begin{array}{ll} S_{10}.\mathtt{Jm}[k,j] = j{-}1 & |\ \mathtt{Kb} \le k \le \mathtt{Ke} \;\wedge\; 2 \le j \le \mathtt{Jmax} \\ S_{10}.\mathtt{Jm}[k,j] = \mathtt{Jmax} & |\ \mathtt{Kb} \le k \le \mathtt{Ke} \;\wedge\; p \;\wedge\; j = 1 \\ S_{10}.\mathtt{Jm}[k,j] = 1 & |\ \mathtt{Kb} \le k \le \mathtt{Ke} \;\wedge\; \neg p \;\wedge\; j = 1 \end{array}\right. \quad (9)$$

$$\left[\begin{array}{ll} S_8.\mathtt{Jlow}[k] = 2 & |\ \mathtt{Kb} \le k \le \mathtt{Ke} \;\wedge\; \neg p \\ S_8.\mathtt{Jlow}[k] = 1 & |\ \mathtt{Kb} \le k \le \mathtt{Ke} \;\wedge\; p \\ S_8.\mathtt{Jup}[k] = \mathtt{Jmax}{-}1 & |\ \mathtt{Kb} \le k \le \mathtt{Ke} \;\wedge\; \neg p \\ S_8.\mathtt{Jup}[k] = \mathtt{Jmax} & |\ \mathtt{Kb} \le k \le \mathtt{Ke} \;\wedge\; p \end{array}\right. \quad (10)$$

Figure 3: Fragment of XIDIF of ARC2D

$j \geq 2$ the statement instance $S_{10}[k,j]$ reads value from the array cell $X(j-1,k)$. This fact makes generating the efficient communication possible.

Also we are able to prove that output dependence $S_{10}\ \delta^O\ S_{11}$ does not exist. To compute this dependence we build dependence problem (8) and substitute into it F-relations (10) that describe values of variables `Jlow` and `Jup`. Simplifying we find that (8) has no solutions.

2 Definitions

Affine program fragment. *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 v_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 `l` is affine program fragment, but the body of the loop `k` is non-affine fragment, because lower and upper bounds of the loop `l` 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 $\mathbf{w}, \mathbf{r}, \mathbf{s}$. 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 $S[\mathbf{v}]$ is specified by S — statement of the program, and $\mathbf{v}$ — tuple of loop variables values (loops that surround the statement S are included). For example, in Figure 1 statement S_1 has

F-relation			F-relation instance	
$S_3[i,j]$	$= 2i–j–2$	$\mid 1\leq i\leq j\leq N$	$S_3[5,7]$	$= 1$
$S_2[i]$	$= \mathtt{In}.A(2N–i)$	$\mid 1\leq i\leq N$	$S_2[1]$	$= \mathtt{In}.A(2)$
$S_2[i,j]$	$= S_1[i] + 2S_3[i,j]$	$\mid 1\leq i\leq j\leq N$	$S_2[7,8]$	$= S_1[7] + 2S_3[7,8]$
$\mathtt{Out}.B(i,i)$	$= S_2[i]$	$\mid 1\leq i\leq N$	$\mathtt{Out}.B(3,1)$	$= S_2[5]$

Figure 4: Examples of F-relations and their instances

$\frac{\mathtt{Mb(Mb+1)}}{2}$ instances $S_1[i,j] \mid 1\leq j\leq i\leq \mathtt{Mb}$.

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:

$$S[\mathbf{w}] = F(S_{i_1}[\mathbf{r}_1],...,S_{i_m}[\mathbf{r}_m];\ \mathtt{In}.A_{j_1}(\mathbf{s}_1),...,\mathtt{In}.A_{j_n}(\mathbf{s}_n);\ \mathbf{r})\ \mid\ p(\mathbf{r}_1,...,\mathbf{r}_m;\ \mathbf{s}_1,...,\mathbf{s}_n;\ \mathbf{r};\ \mathbf{w})$$

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 $\mathbf{r}_1,...,\mathbf{r}_m;\ \mathbf{s}_1,...,\mathbf{s}_n;\ \mathbf{r};\ \mathbf{w}$: if $p(\mathbf{r}_1,...,\mathbf{r}_m;\ \mathbf{s}_1,...,\mathbf{s}_n;\ \mathbf{r};\ \mathbf{w})$ is True then statement instance $S[\mathbf{w}]$ is a result of application of function F to the values of statement instances $S_{i_1}[\mathbf{r}_1],...,S_{i_m}[\mathbf{r}_m]$, values of initial memory cells $\mathtt{In}.A_{j_1}(\mathbf{s}_1),...,\mathtt{In}.A_{j_n}(\mathbf{s}_n)$ and value of tuple $\mathbf{r}$.

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:

- $\mathbf{r}$: constant. Consume value of a constant.
- $\mathtt{In}.A_{j_l}(\mathbf{s}_l)$: memory read. Consume value of memory cell that it had before execution of the affine program fragment represented by F-relation.
- $S_{i_l}[\mathbf{r}_l]$: statement instance. Consume value computed by another statement instance.

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

- $S[\mathbf{w}] = x$: Value of x becomes a value of statement instance $S[\mathbf{w}]$ (to be consumed by another statement instance).
- $\mathtt{Out}.A_{k_l}(\mathbf{s}_l) = x$: 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:

1. There should be no contradictory definitions for every statement instance. That is, if some statement instance $S[\mathbf{v}]$ is defined twice: $S[\mathbf{v}] = F(ArgList_1)$ and $S[\mathbf{v}] = G(ArgList_2)$, then $F = G$ and $ArgList_1 = ArgList_2$.

2. There should be no cycles. That is, for every statement instance $S[\mathbf{v}]$ there should not exist chain of simple F-relation instances that defines $S[\mathbf{v}]$ 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 $S[\mathbf{v}]$ that appears as an argument of some simple F-relation from R there should exist a definition of $S[\mathbf{v}]$ in R.

Operations over F-relations.

$\mathsf{Domain}(R)$: Domain of F-relation R.

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

$$\mathsf{Domain}(R) \;=\; \{S_{i_1}[\mathbf{r}_1] \,\cup\, \cdots \,\cup\, S_{i_m}[\mathbf{r}_m] \mid \pi_{\mathbf{r}_1,\ldots,\mathbf{r}_m}(p(\mathbf{r}_1,\ldots,\mathbf{r}_m; \mathbf{s}_1,\ldots,\mathbf{s}_n;\ \mathbf{r};\ \mathbf{w}))\}$$

$\mathsf{Domain}(R.S_{i_l}[\mathbf{r}_l])$: Domain of F-relation R with respect to its argument $S_{i_l}[\mathbf{r}_l]$.

This is a set of statement instances consumed by reference $S_{i_l}[\mathbf{r}_l]$ of F-relation R:

$$\mathsf{Domain}(R.S_{i_l}[\mathbf{r}_l]) \;=\; \{S_{i_l}[\mathbf{r}_l] \mid \pi_{\mathbf{r}_l}(p(\mathbf{r}_1,\ldots,\mathbf{r}_m; \mathbf{s}_1,\ldots,\mathbf{s}_n;\ \mathbf{r};\ \mathbf{w}))\}$$

$\mathsf{Range}(R)$: Range of F-relation R.

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

$$\mathsf{Range}(R) \;=\; \{S[\mathbf{w}] \mid \pi_{\mathbf{w}}(p(\mathbf{r}_1,\ldots,\mathbf{r}_m; \mathbf{s}_1,\ldots,\mathbf{s}_n;\ \mathbf{r};\ \mathbf{w}))\}$$

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 $S[\mathbf{v}] \in \mathsf{Range}(R)$, we create a vertex $S[\mathbf{v}]$ in VFG.
- If statement instance $S_1[\mathbf{v}_1]$ is an argument of F-relation instance that computes $S_2[\mathbf{v}_2]$, (that is, if $S_1[\mathbf{v}_1]$ directly passes value to $S_2[\mathbf{v}_2]$) we create a directed arc $(S_1[\mathbf{v}_1],\ S_2[\mathbf{v}_2])$ in VFG.

Value Flow Graph is an *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 it 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 $p(\mathbf{r}_1,\ldots,\mathbf{r}_m;\ \mathbf{s}_1,\ldots,\mathbf{s}_n;\ \mathbf{r};\ \mathbf{w})$ is a conjunction of affine equalities and inequalities over integers. Our algorithms require the following operations to be performed on problems like p: $p_1 \wedge p_2$, $p_1 \vee p_2$, $\neg p$, $\exists\ v\ s.t.\ p$, $\forall\ v : p$. 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]:

$$\pi_{\mathbf{v}}(P(\mathbf{v},\mathbf{w})) = \exists\ \mathbf{w}\ s.t.\ P(\mathbf{v},\mathbf{w})$$

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

   $$S:\ \ a = F(b_1, b_2, ..., b_m; \mathbf{v})$$

   where $\mathbf{v}$ is a tuple of loop variables surrounding S and $b_1, ..., b_m$ are array or scalar references, create simple F-relation

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

   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]:

   $$S_x[\mathbf{w}] \rightarrow S_{i_l}[\mathbf{v}] \mid p(\mathbf{w},\mathbf{v})$$

   Convert each dependence relation to equivalent F-relation:

   $$S_{i_l}[\mathbf{v}] = S_x[\mathbf{w}] \mid p(\mathbf{w},\mathbf{v})$$

3. Propagate all assignments for statement instances $S_{i_l}[\mathbf{v}]$ to the statement where they are used, that is, to the statement S. Since each $S_{i_l}[\mathbf{v}]$ 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 [1]. 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.

```
S1: X(0) = 0
    DO i = 1, 2*N
S2:   SAVE(i) = X(2*N-i+1)
S3:   X(i) = X(i-1) + SAVE(i)
    END DO
```

$$\left[\begin{array}{ll} R_1 : S_1[] = 0 & \\ R_2 : S_2[i] = \mathtt{In}.X(2N{-}i{+}1) & \mid 1 \le i \le N \\ R_3 : S_2[i] = S_3[2N{-}i{+}1] & \mid N{+}1 \le i \le 2N \\ R_4 : S_3[1] = S_1[1] + S_2[1] & \\ R_5 : S_3[i] = S_3[i{-}1] + S_2[i] & \mid 2 \le i \le 2N \end{array}\right. \qquad (11)$$

$$(R_1, R_4), (R_2, R_4), (R_2, R_5), (R_3, R_5), (R_4, R_3), (R_4, R_5), (R_5, R_3), (R_5, R_5) \qquad (12)$$

$$\left[\begin{array}{ll} R_3 : S_2[i] = S_3[2N{-}i{+}1] & \mid N{+}1 \le i \le 2N \\ R'_4 : S_3[1] = \mathtt{In}.X(2N) & \\ R'_5 : S_3[i] = S_3[i{-}1] + \mathtt{In}.X(2N{-}i{+}1) & \mid 2 \le i \le N \\ R''_5 : S_3[i] = S_3[i{-}1] + S_2[i] & \mid N{+}1 \le i \le 2N \end{array}\right. \qquad (13)$$

$$\left[\begin{array}{ll} R'_4 : S_3[1] = \mathtt{In}.X(2N) & \\ R'_5 : S_3[i] = S_3[i{-}1] + \mathtt{In}.X(2N{-}i{+}1) & \mid 2 \le i \le N \\ R'''_5 : S_3[i] = S_3[i{-}1] + S_3[2N{-}i{+}1] & \mid N{+}1 \le i \le 2N \end{array}\right. \qquad (14)$$

Figure 5: Program, its VFG and characteristic graph

Propagate($R_1, R_2.S_1[\mathbf{v}_2]$) Begin
  Given:
    Simple F-relation R_1: $S_1[\mathbf{v}_1] = F(Args_1(\mathbf{u}_1)) \mid p(\mathbf{v}_1, \mathbf{u}_1)$
    Simple F-relation R_2: $S_2[\mathbf{w}] = G(S_1[\mathbf{v}_2], Args_2(\mathbf{u}_2)) \mid q(\mathbf{w}, \mathbf{v}_2, \mathbf{u}_2)$
  Perform propagation only if Transfer($R_1, R_2.S_1[\mathbf{v}_2]$) is not empty:
    Transfer($R_1, R_2.S_1[\mathbf{v}_2]$) = Range($R_1$) $\cap$ Domain($R_2.S_1[\mathbf{v}_2]$)
      $= \{S_1[\mathbf{v}_1] \mid \pi_{\mathbf{v}_1}(p(\mathbf{v}_1, \mathbf{u}_1) \wedge q(\mathbf{w}, \mathbf{v}_2, \mathbf{u}_2) \wedge \mathbf{v}_1 = \mathbf{v}_2)\}$
  Replace R_2 with $R'_2 \cup R''_2$:
    $R'_2 : S_2[\mathbf{w}] = G(F(Args_1(\mathbf{u}_1)), Args_2(\mathbf{u}_2)) \mid \pi_{\mathbf{w},\mathbf{v}_1,\mathbf{u}_2}(p(\mathbf{v}_1, \mathbf{u}_1) \wedge q(\mathbf{w}, \mathbf{v}_2, \mathbf{u}_2) \wedge \mathbf{v}_1 = \mathbf{v}_2)$
    $R''_2 : S_2[\mathbf{w}] = G(S_1[\mathbf{v}_2], Args_2(\mathbf{u}_2)) \mid q(\mathbf{w}, \mathbf{v}_2, \mathbf{u}_2) \wedge \neg\pi_{\mathbf{w},\mathbf{v}_2}(p(\mathbf{v}_1, \mathbf{u}_1) \wedge q(\mathbf{w}, \mathbf{v}_2, \mathbf{u}_2) \wedge \mathbf{v}_1 = \mathbf{v}_2)$
  In characteristic graph:
    Remove arcs coming to and from R_2
    Compute transfer sets for R'_2 and R''_2 and add corresponding arcs to CG
    If (vertex R_1 has no incoming arcs) Remove vertex R_1 from the CG
End

Figure 6: Algorithm to perform single propagation

- For each simple F-relation R_i we create one vertex in CG. Therefore CG has n vertices.
- For each pair consisting of simple F-relation R_i and argument $S[\mathbf{v}]$ of simple F-relation R_j we compute *transfer set*: Transfer($R_i, R_j.S[\mathbf{v}]$) = Range(R_i) $\cap$ Domain($R_j.S[\mathbf{v}]$). If this set is not empty we add to the CG arc (R_i, R_j) loaded with its transfer set Transfer($R_i, R_j.S[\mathbf{v}]$).

Characteristic graph is a finite graph with statically known number of vertices, but unlike VFG it can have cycles. These cycles reflect the recurrent nature of computations that take place in scientific programs. For example, F-relation (11) representing a program from Figure 5 consists of 5 simple F-relations. Therefore, CG of this program has 5 vertices. Characteristic graph arcs are listed in (12). There are 2 cycles in the CG: (R_5, R_5) and (R_3, R_5, R_3).

4.2 Single act of propagation

Function Propagate presented in Figure 6 propagates statement instances computed by simple F-relation R_1 to argument $S_1[\mathbf{v}_2]$ of simple F-relation R_2. The function checks that transfer set from

```
SafePropagation(F-relation R) Begin
    DoWhile (exists simple F-relations R1, R2 ∈ R and argument S[v] of R2
            such that Transfer(R1, R2.S[v]) = Domain(R2.S[v]))
        Propagate(R1, R2.S[v])
    EndWhile
End
```

Figure 7: Safe propagation algorithm that avoids splintering

R_1 to $R_2.S_1[\mathbf{v}_2]$ is not empty. Otherwise there is nothing to propagate.

Basically this function replaces instances of $S_1[\mathbf{v}_1]$ with expressions from R_1 that compute their values. Since R_1 does not necessarily compute values for all instances of $S_1[\mathbf{v}_2]$ 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)):

$$\begin{array}{ll} R_2 : S_2[i_1] = \mathtt{In}.X(i_2) & \mid 1 \le i_1 \le N \ \wedge \ i_2 = 2N - i_1 + 1 \\ R_5 : S_3[j_1] = S_3[j_3] + S_2[j_2] & \mid 2 \le j_1 = j_2 \le 2N \ \wedge \ j_3 = j_2 - 1 \end{array}$$

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

$$\begin{array}{ll} R_2 : S_2[i_1] = \mathtt{In}.X(i_2) & \mid 1 \le i_1 \le N \ \wedge \ i_2 = 2N - i_1 + 1 \\ R_5' : S_3[j_1] = \mathtt{In}.X(i_2) + S_3[j_3] & \mid 2 \le j_1 \le 2N \ \wedge \ i_2 = 2N - i_1 + 1 \ \wedge \ j_3 = j_2 - 1 \\ R_5'' : S_3[j_1] = S_2[j_2] + S_3[j_3] & \mid N+1 \le j_1 = j_2 \le 2N \ \wedge \ j_3 = j_2 - 1 \end{array}$$

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:

$$\begin{array}{l} R_1 : S[1] = x_0 \\ R_2 : S[i] = F(S[i-1]) \mid 2 \le i \le N \end{array} \Rightarrow \begin{array}{l} R_1 : S[1] = x_0 \\ R_2' : S[2] = F(x_0) \\ R_3 : S[i] = F(S[i-1]) \mid 3 \le i \le N \end{array} \Rightarrow \begin{array}{l} R_1 : S[1] = x_0 \\ R_2' : S[2] = F(x_0) \\ R_3' : S[3] = F(F(x_0)) \\ R_4 : S[i] = F(S[i-1]) \mid 4 \le i \le N \end{array} \Rightarrow \cdots \quad (15)$$

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(R1, R2.S[v]) = Domain(R2.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_5) and (R''_5, R''_5), each of unit length. The result of these propagations is F-relation (13).

On next iteration we perform safe propagation of R_3 to $R''_5.S_2[i]$. We also try to propagate R'_4 to $R'_5.S_3[i\text{–}1]$ 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:

$$DepProb(\mathbf{v}) \;=\; F(a_1(\mathbf{v}), ..., a_q(\mathbf{v}); \mathbf{v}) \geq 0 \;\;\wedge\;\; L(\mathbf{v})$$

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:

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

and propagate F-relations for $S_{a_1}[\mathbf{v}], ..., S_{a_q}[\mathbf{v}]$ 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

$$R'_D: \; S_D[\mathbf{v}] = F'(\mathbf{v}, \mathbf{w}) \mid L'(\mathbf{v}, \mathbf{w})$$

Relation SourceFunction($R.A$) Begin
Input: $R.A$ is a read reference surrounded by n loops with variables $\mathbf{r} = (r_1, ..., r_n)$.
(* Compute dependence relation that represents source function for $R.A$ *)
1: Static stack *AlreadyInAnalysis*
2: If (*AlreadyInAnalysis* contains $R.A$) Return (`NonAffine`)
3: Push $R.A$ to stack *AlreadyInAnalysis*
Relation $DepRel := \{\emptyset\}$
Dnf $NotCovered(\mathbf{r}) :=$ IsExecuted($R[\mathbf{r}]$)
Statement $W := R$
While ($NotCovered$ is feasible) do
  $W :=$ statement preceding statement W
  Statement W is surrounded by m loops with variables $\mathbf{w} = (w_1, ..., w_m)$
  If (W is assignment statement that writes to array of $R.A$) then
    Build dependence problem $DepProb(\mathbf{w}, \mathbf{r})$ for dependence from W to $R.A$
10: $(a_1, ..., a_q) :=$ list of non-affine references in $DepProb$)
11: For ($i := 1$ to q) $R_i :=$ SourceFunction(a_i)
12: If (all of R_i are `Affine`) then
13: Create F-relation R_D that represents values computed by $DepProb(\mathbf{w}, \mathbf{r})$
14: Propagate to R_D values carried by relations $R_1, ..., R_q$
15: If Domain(R_D) $= \{\emptyset\}$ then convert $DepProb$ to affine form
16: Else
17: Source function for $R.A$ is non-affine. Compute its affine approximation.
18: EndIf
    Relation $Cmax :=$ RelMax1$_{\ll}(W[\mathbf{w}] \rightarrow R.A[\mathbf{r}] \mid DepProb(\mathbf{w}, \mathbf{r}))$
    $DepRel := DepRel \cup Cmax$
    $NotCovered := NotCovered \wedge \neg$Range($Cmax$)
    ... AND SO ON, SEE [MAS94] ...
  EndIf
EndDo
20: Remove $R.A$ from top of stack *AlreadyInAnalysis*
Return ($DepRel$)

Figure 10: Lazy dependence analysis combined with global value propagation

where F' is an affine function and $\mathbf{w}$ is a tuple of variables added by propagation, then for each R'_D we generate affine constraint

$$F'(\mathbf{v}, \mathbf{w}) \geq 0 \ \wedge \ L'(\mathbf{v}, \mathbf{w})$$

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.`IP(I1)` in Figure 2 we build the following dependence problem:

$$p(l_w, l_r) \ = \ (S_3.\mathtt{I1}[l_w] = S_3.\mathtt{I2}[l_r] \ \wedge \ 1 \leq l_w < l_r \leq \mathtt{N1})$$

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:

$$S_D[l_w, l_r] \ = \ S_3.\mathtt{I1}[l_w] - S_3.\mathtt{I2}[l_r] \mid 1 \leq l_w \leq l_r \leq \mathtt{N1}$$

```
    DO i = 1, N
S1:   A(A(i)) = x
    END DO
```

$$1 \le i_w < i_r \le N \;\wedge\; S_1.\mathtt{A}(\mathtt{i})[i_w] = i_r \tag{16}$$

Figure 11: Propagation cycle example

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

$$\left[\begin{array}{ll} S_D[l_w, l_r] = -1 & | \, 1 \le l_w \le l_r \le \mathtt{N1} \;\wedge\; l_w = 2\alpha{+}1 \;\wedge\; l_r = 2\beta{+}1 \\ S_D[l_w, l_r] = 0 & | \, 1 \le l_w \le l_r \le \mathtt{N1} \;\wedge\; l_w = 2\alpha \;\wedge\; l_r = 2\beta{+}1 \\ S_D[l_w, l_r] = 0 & | \, 1 \le l_w \le l_r \le \mathtt{N1} \;\wedge\; l_w = 2\alpha{+}1 \;\wedge\; l_r = 2\beta{+}1 \\ S_D[l_w, l_r] = 1 & | \, 1 \le l_w \le l_r \le \mathtt{N1} \;\wedge\; l_w = 2\alpha \;\wedge\; l_r = 2\beta \end{array}\right.$$

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:

$$p(l_w, l_r) = \begin{array}{l} (1 \le l_w \le l_r \le \mathtt{N1} \;\wedge\; l_w = 2\alpha \;\wedge\; l_r = 2\beta{+}1) \vee \\ (1 \le l_w \le l_r \le \mathtt{N1} \;\wedge\; l_w = 2\alpha{+}1 \;\wedge\; l_r = 2\beta{+}1) \end{array}$$

Computing lexicographical maximum $\max_{\ll}(l_w \mid p(l_w, l_r))$ 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 [WCES94].

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 [Fea91, PW93, Mas94].

$$\left[\begin{array}{ll} R_1 : S_1[i] = S_0[i] & \mid 1 \le i \le N \\ R_2 : S_1[i] = F(S_1[i{-}N]) & \mid N{+}1 \le i \le 2N \\ R_3 : S_2[i] = G(S_1[i]) & \mid N{+}1 \le i \le 2N \end{array}\right] \qquad (17)$$

$$(R_1, R_2), (R_2, R_3)$$

$$\left[\begin{array}{l} Q_1 : S_1[i] = \left\{\begin{array}{ll} S_0[i] & \mid 1 \le i \le N \\ F(S_1[i{-}N]) & \mid N{+}1 \le i \le 2N \end{array}\right. \\ Q_3 : S_2[i] = G(S_1[i]) \mid N{+}1 \le i \le 2N \end{array}\right] \qquad (18)$$

$$(Q_1, Q_1), (Q_1, Q_3)$$

Figure 12: Characteristic graph vs system graph

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 many 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 $\mathsf{Transfer}(R_1, R_2)$ 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

[B+89] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.

[BE94] William Blume and Rudolf Eigenmann. Symbolic analysis techniques needed for effective parallelization of the Perfect benchmarks. Technical Report 1332, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1994.

[Blu92] William Joseph Blume. Success and limitations in automatic parallelization of the Perfect benchmarks^{TM} programs. Master's thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1992.

[BMO90] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence Web. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 257–271, June 1990. Published as SIGPLAN Notices Vol. 25, No. 6.

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

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), February 1991.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[HP92] M. Haghighat and C. Polychronopoulos. Symbolic program analysis and optimization for parallel compilers. Technical Report 1237, CSRD, Univ. of Illinois, August 1992. Presented at the 5th Annual Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, August 3-5, 1992.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *ACM '93 Conf. on Programming Language Design and Implementation*, pages 78–89, June 1993.

[Mas94] Vadim Maslov. Lazy array data-flow dependence analysis. In *ACM '94 Conf. on Principles of Programming Languages*, January 1994.

[Pug91] William Pugh. Uniform techniques for loop optimization. In *1991 International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[RF93] Xavier Redon and Paul Feautrier. Detection of recurrences in sequential programs with loops. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Proceedings of the 5th International Parallel Architectures and Languages Europe*, pages 132–145, June 1993.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[Voe92a] Valentin V. Voevodin. *Mathematical Foundations of Parallel Computing.* World Scientific Publishers, 1992. World Scientific Series in Computer Science, vol. 33.

[Voe92b] Vladimir V. Voevodin. Theory and practice of parallelism detection in sequential programs. *Programming and Computer Software (Programmirovaniye)*, 18(3), May 1992.

[WCES94] Daniel Weise, Roger Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *ACM '94 Conf. on Principles of Programming Languages*, January 1994.

[Wol92] Michael Wolfe. Beyond induction variables. In *SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, California, June 1992.

[Yer73] A. P. Yershov. Current state of program schemes theory. *Problems of Cybernetics (in Russian)*, 27:87–110, 1973.