Fuzzy Array Dataflow Analysis

Jean-François Collard
LIP
ENS Lyon
46 Allée d'Italie
F-69364 Lyon Cedex 07
Jean-Francois.Collard@lip.ens-lyon.fr

Denis Barthou Paul Feautrier
PRiSM Laboratory
Université de Versailles
45 Avenue des Etats-Unis
F-78035 Versailles Cedex
{Denis.Barthou, Paul.Feautrier}@prism.uvsq.fr

Abstract

Exact array dataflow analysis can be achieved in the general case if the only control structures are `do`-loops and structural `if`s, and if loop counter bounds and array subscripts are affine expressions of englobing loop counters and possibly some integer constants. In this paper, we begin the study of dataflow analysis of dynamic control programs, where arbitrary `if`s and `while`s are allowed. In the general case, this dataflow analysis can only be fuzzy.

1 Introduction

Gathering information on data values is a classical task in advanced compilers, known as *Dataflow Analysis* [1]. However, this technique only deals with scalar data, and sees an array as an indivisible object. On the other hand, vectorization and parallelization methods are mainly based on the parallelism hidden by independent references to distinct parts of arrays. Various dependence tests have been proposed [2]. However, these tests are not exact, and, even when they are, cannot distinguish between true dependences, which describe a real information flow, and spurious dependences, in which the value purported to be transmitted is destroyed before being used. To obviate this difficulty, methods have been designed to compute, for every array cell read in a right-hand-side expression (the "sink"), the very operation which produced it (the "source"). These methods are called *Array Dataflow Analyses* (ADA) [6, 10], or *Value-Based Dependence Analyses* [11]. These ADAs, however, make quite stringent hypotheses on the input programs. The only accepted control structures are the `do` loop and the sequence; loop bounds and array subscripts must be affine functions of surrounding loop counters and possibly of symbolic constants a.k.a. *structure parameters*. Programs following this model have been called "static control programs" in [6]. The same paper showed that an exact ADA can be mechanically performed on static control programs.

This paper deals with handling general `if`s and `while` loops. With such unpredictable control structures, no exact information can be hoped for in the general case. The aim of this paper is to show that even partial information can be automatically gathered thanks to *Fuzzy Array Dataflow Analysis* (FADA). Moreover, should this FADA be applied to a static control program, it is easy to show that the precise, classical ADA is a special (non-fuzzy) case of FADA.

PPOPP '95 Santa Clara, CA USA
© 1995 ACM 0-89791-701-6/95/0007...$3.50

1.1 Paper Overview

Section 2 gives a motivating example. Section 3 then briefly reviews the exact array dataflow analysis for static control programs proposed in [6]. Section 4 defines our program model. Section 5 details the algorithm.

1.2 Notations

The k-th entry of vector $\vec{x}$ is denoted by $\vec{x}[k]$. The subvector built from components k to l is written as: $\vec{x}[k..l]$. If $k > l$, then this vector is by convention the unique vector of dimension 0. Furthermore, $\ll$ denotes the strict lexicographical order on such vectors. In this paper, "max" always denotes the maximum operator according to the $\ll$ order. An instance of statement S is denoted by $\langle S, \vec{x} \rangle$, where $\vec{x}$, the iteration vector of S, is the vector built from the counters of loops surrounding S.

2 A Motivating Example

The following two sibling codes exemplify dynamic control programs.

```
      program M
      do i = 1 , n
S0        a(i) = ...
          if .. then
             do j = i , n+2
S1              a(j) = a(j-2)
             enddo
          endif
      enddo
```

```
      program N
      do i = 1 , n
S0 :      a(i) = ...
          while ... do
             do j = i , n+2
S1 :            a(j) = a(j-2)
             enddo
          enddo
      enddo
```

Let us consider Program `M` first, and suppose that $n = 4$. Let us study the case of the *instance* of statement S_1 when $i = 3$ and $j = 4$, ie $\langle S_1, 3, 4\rangle$. Note that we don't even know at compile-time if this instance actually executes. If it does, however, then the problem is to know where and when the right-hand-side value `a(2)` was produced. This source may be an instance of S_1, but not if $i > 3$, since this instance would execute *after* $\langle S_1, 3, 4\rangle$. Since the source must write into `a(2)`, the value of j is fixed to 2. This source cannot be an instance of S_1 for $i = 3$ either, since j is greater than or equal to i. Thus, *possible sources* are instances $\langle S_1, 1, 2\rangle$ and $\langle S_1, 2, 2\rangle$. Another potential source is $\langle S_0, 2\rangle$. Note moreover that $\langle S_0, 2\rangle$ overwrites the value that $\langle S_1, 1, 2\rangle$ may have written. Thus, the set of potential sources is $\{\langle S_0, 2\rangle, \langle S_1, 2, 2\rangle\}$.

Actually, the iteration points of S_1 fall into three groups (see Fig. 1 (b)):

- A member (i, j) of the first group is such that $j \geq i + 2$. It has one and only one possible source from S_1 (namely, $\langle S_1, i, j - 2\rangle$) since if point (i, j) executes then $(i, j - 2)$ did execute too.

- On the contrary, a member of the second group has an unpredictable source. However, all the members of this group have at least one source, since all the array cells they read (`a(1)` through `a(n-1)`) are written into by S_0. Dotted edges symbolize this.

- Finally, members of the third group do not have sources in the given program.

The analysis for Program `N` is obviously similar, except that the iteration domain of S_1 is 3-dimensional. Program `N` will serve as a running example in the sequel.

3 Review of an Exact Array Dataflow Analysis

The aim of this section is to summarize an array dataflow analysis of static control programs [6]. The reader is referred to [6] for details.

The *depth* of a construct is the number of surrounding loops. The counter of a loop at depth p is the $(p+1)$-th component of the iteration vector.

The sequential execution order is written $\prec$ and is defined by:

$$\langle S, \vec{x}\rangle \prec \langle R, \vec{y}\rangle \equiv \vec{x}[1..N_{SR}] \ll \vec{y}[1..N_{SR}] \vee (\vec{x}[1..N_{SR}] = \vec{y}[1..N_{SR}] \wedge T_{SR}), \quad (1)$$

where N_{SR} is the number of loops surrounding both S and R, and T_{SR} is a boolean which is true iff S precedes R in the program text. Notice that this sequential order can be split with respect to depths:

$$\langle S, \vec{x}\rangle \prec \langle R, \vec{y}\rangle \equiv \bigvee_{p=0}^{N_{SR}} \langle S, \vec{x}\rangle \prec_p \langle R, \vec{y}\rangle \quad (2)$$

where for $p = 0..N_{RS} - 1$:

$$\langle S, \vec{x}\rangle \prec_p \langle R, \vec{y}\rangle \Leftrightarrow (\vec{x}[1..p] = \vec{y}[1..p]) \wedge (\vec{x}[p+1] < \vec{y}[p+1]) \quad (3)$$

and:

$$\langle S, \vec{x}\rangle \prec_{N_{SR}} \langle R, \vec{y}\rangle \Leftrightarrow \vec{x}[1..N_{SR}] = \vec{y}[1..N_{SR}] \wedge T_{SR} \quad (4)$$

For a given loop at depth p, $\vec{x}[p+1]$ has a minimum and a maximum which are given by the loop bounds. In the case of exact ADA, these bounds are affine functions of outer loop counters and structure parameters:

$$l_p(\vec{x}[1..p]) \leq \vec{x}[p+1] \leq u_p(\vec{x}[1..p]). \quad (5)$$

The iteration domain of a statement S is denoted by $\mathbf{D}(S)$ and is given by the conjunction of all inequalities (5) for the surrounding loops.

Let us consider two statements S and R. Suppose that S writes into an array `a` and that R reads that same array:

```
S :  a(f(x)) = ...
                ...
R :  ...     = a(g(y))
```

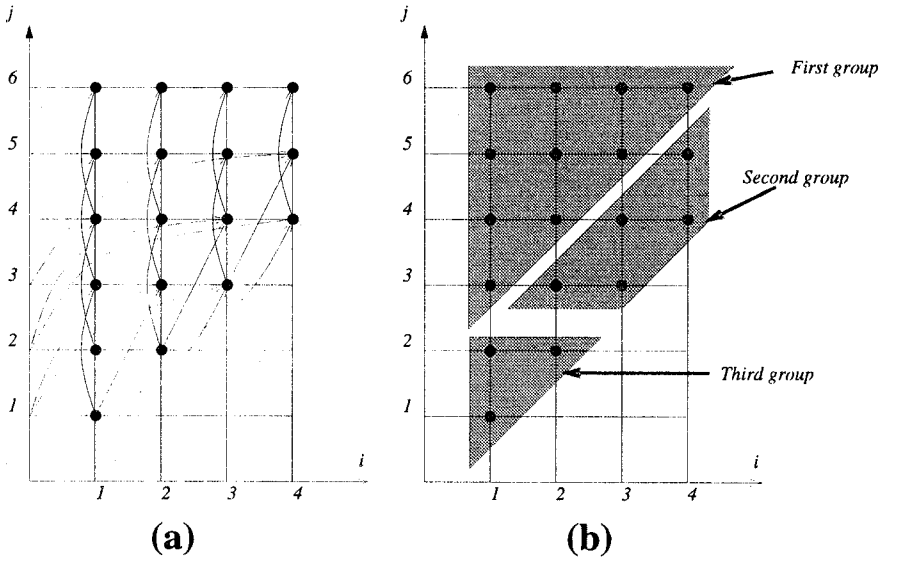

Figure 1: Dataflow graph of Program M.

The aim of array dataflow analysis is to find the source of the value $\mathtt{a}(g(\vec{y}))$ read in R for a given $\vec{y}$. This source is denoted by $\sigma(\langle R, \vec{y}\rangle)$. To be a candidate source, an operation $\langle S, \vec{x}\rangle$ has to satisfy the following constraints:

Existence predicate: $\langle S, \vec{x}\rangle$ is a valid operation:

$$\vec{x} \in \mathbf{D}(S). \tag{6}$$

Conflicting accesses: $\langle S, \vec{x}\rangle$ and $\langle R, \vec{y}\rangle$ access the same array cell:

$$f(\vec{x}) = g(\vec{y}). \tag{7}$$

f and g are possibly multi-dimensional affine functions w.r.t. $\vec{x}$ and $\vec{y}$, respectively.

Sequencing condition: $\langle S, \vec{x}\rangle$ is executed before $\langle R, \vec{y}\rangle$:

$$\langle S, \vec{x}\rangle \prec \langle R, \vec{y}\rangle. \tag{8}$$

Environment: The set of candidates is to be computed under the hypothesis that $\langle R, \vec{y}\rangle$ is a valid operation, i.e. $\vec{y} \in \mathbf{D}(R)$.

The set of candidate sources is thus:

$$\mathbf{Q}_{SR}(\vec{y}) = \{\vec{x} \mid \vec{x} \in \mathbf{D}(S),\ f(\vec{x}) = g(\vec{y}),\ \langle S, \vec{x}\rangle \prec \langle R, \vec{y}\rangle\}.$$

Thanks to (2), this set can be split for each possible depth p:

$$\mathbf{Q}^p_{SR}(\vec{y}) = \{\vec{x} \mid \vec{x} \in \mathbf{D}(S),\ f(\vec{x}) = g(\vec{y}),\ \langle S, \vec{x}\rangle \prec_p \langle R, \vec{y}\rangle\}. \tag{9}$$

Since each predicate $\prec_p$ is affine, $\mathbf{Q}^p_{SR}(\vec{y})$ is a polyhedron. The *direct dependence* from S to R at depth p is the maximal element according to the $\ll$ order:

$$K^p_{SR}(\vec{y}) = \max\ \mathbf{Q}^p_{SR}(\vec{y}). \tag{10}$$

The maximal value is computed for each depth by integer linear programming [5]. The corresponding *operation* is denoted by $\mathcal{S}^p_{SR}(\vec{y}) = \langle S, K^p_{SR}(\vec{y})\rangle$.

In general, however, there are several statements $S_1, \ldots, S_m$ writing into the same array, and as many sets of candidates. One has to compute the maximum of the direct dependences:

$$\sigma(\langle R, \vec{y}\rangle) = \max\{\mathcal{S}^p_{S_k R}(\vec{y}) \mid 1 \le k \le m\,, 0 \le p \le N_{S_k R}\}. \tag{11}$$

This maximum is computed with the help of a set of simplification rules, which are given in Section 5.2. The result of the analysis is a *quasi-affine selection tree* or *quast*, i.e. a many-level conditional in which:

- Predicates are tests for the positiveness of affine forms in the loop counters and structure parameters.
- Leaves are either operation names whose iteration vector components are again affine, or $\perp$. The special name $\perp$ indicates that the array cell under study is not modified in some piece of code. A coherent way of thinking about $\perp$ is to consider it as the name of an operation which is executed once before all other operations of the program, i.e.:

$$\forall S, \vec{x} : \perp \prec \langle S, \vec{x}\rangle. \tag{12}$$

4 Definition of Fuzzy Array Dataflow Analysis

4.1 Program model

In this paper, our aim is to extend the scope of array dataflow analysis to programs respecting the following

constraints: 1) The only data structures are integers, reals, and arrays thereof. 2) The only control structures are the sequence, the **do** loop, the **while** loop, and the **if..then..else** construct. **gotos** and procedure calls are forbidden. 3) Basic statements are assignments to scalars or array elements. 4) No pointer, **EQUIVALENCE** or aliasing is allowed. 5) Array subscripts must be affine functions of **do** loops counters and structure parameters. 6) Array subscripts must stay within array bounds. This condition is usually imposed by parallelizing compilers. The rationale is that a program which does not conform to this restriction is incorrect and can be incorrectly handled by the compiler. Besides, the property can be statically checked when array subscripts do not depend on **while** counters.

Similarly to **do** loops, an iteration of a **while** loop is denoted by giving its ordinal number w in the iteration sequence Most of the definitions given in [6] for static control programs translate directly to the present model. In particular, the meaning of the execution order (1) is that, when the set of operations of a given execution is known, *then* their execution order is given by (1).

4.2 Hidden variables

To be definite, we will postulate that there exists a set of "hidden variables", collectively denoted by $\vec{\xi}$, which completely determine the set of operations of the associated execution. If the value of these variables were known, and if we could completely analyze the behavior of the program, then we could in principle predict the number of iteration of each **while** loop and the outcome of each test. We will suppose here that a knowledge of the hidden variables allows us to compute:

- the iteration count of each **while** loop. Let W be such a loop and let $\vec{x}$ be the iteration vector of the loops (of any kind) surrounding W. We will assume the existence of a function $\varphi_W(\vec{x},\vec{\xi})$ giving the iteration count of instance $\vec{x}$ of W when the value of the hidden variables is $\vec{\xi}$. In the case of a **while** loop at depth d, the corresponding inequalities are:

$$1 \leq \vec{x}[d+1] \leq \varphi_W(\vec{x}[1..d],\vec{\xi}). \qquad (13)$$

  When the **while** loop under discussion is clear from the context, the W subscript may be omitted.

  In the following, we suppose that our source program is totally correct, i.e. that all **while** loops terminate in a finite number of iterations. This is equivalent to saying that φ always has finite values.

- the outcome of each test. The value of the predicate of conditional C at iteration $\vec{x}$ will be written $\tau_C(\vec{x},\vec{\xi})$. Here again the subscript C may be omitted if no ambiguity results.

The iteration domain of statement S will be written $\mathbf{D}(S,\vec{\xi})$. $\mathbf{D}(S,\vec{\xi})$ is given by the conjunction of all applicable inequalities (5) or (13) and of the τ predicate associated to governing conditionals. For instance, the iteration domain of S_1 in Program **M** is $\{i,j \mid 1 \leq i \leq n, i \leq j \leq n+2, \tau(i,\vec{\xi})\}$. The iteration domain of S_1 in Program **N** is $\{i,w,j \mid 1 \leq i \leq n, 1 \leq w \leq \varphi(i,\vec{\xi}), \iota \leq j \leq n+2\}$.

In the case of static control programs, the only hidden variables are the structure parameters, and the iteration count of all **do** loops may only depend linearly on the structure parameters and outer loop counters. In that case, the function φ can be written explicitly and handled exactly by the dependence analyzer. Similarly, the function τ can be handled exactly if it is an affine function of loop counters and structure parameters.

5 A FADA algorithm

The purpose of this paper is to analyze programs where some existence predicate (6) effectively depends on hidden variables. Our objective is to show that it is still possible to compute approximate sources in this case. The method will be to compute, for each read reference in operation $\langle R,\vec{y}\rangle$, its source $\varsigma(\langle R,\vec{y}\rangle,\vec{\xi})$ as a function of the hidden variables. Since the values of the hidden variables are unknown by definition, the best we can do is to take as an approximation to the real source:

$$\sigma(\langle R,\vec{y}\rangle) = \bigcup_{\vec{\xi}} \varsigma(\langle R,\vec{y}\rangle,\vec{\xi}). \qquad (14)$$

In so doing, we have to take care not to use approximations too early. This is possible by introducing additional parameters, and by proving that varying these parameters is equivalent to modifying the hidden variables. These parameters are introduced, when necessary, in direct dependences (Section 5.1). The combination of direct dependences is described in Section 5.2. The result is expressed as a function of additional parameters whose elimination is dealt with in Section 5.3.

5.1 Direct dependence computation

This section is devoted to the evaluation of (10). It so happens that in some cases, an exact solution may be found even in the presence of **while** loops or tests. These cases are investigated in the next section. We then proceed to the general case.

5.1.1 Exact solutions

The case of while loops. Let us consider the case of a candidate source which is governed by a **while** loop W at depth d. An exact computation of $K^p_{SR}(\vec{y})$ can

be made if and only if the φ's in the existence condition of the source candidate can be eliminated. Suppose that R is also inside W. Among other inequalities, the environment includes:

$$1 \leq \vec{y}[d+1] \leq \varphi(\vec{y}[1..d], \vec{\xi}). \quad (15)$$

Then:

Property 1 *The constraint $\vec{x}[d+1] \leq \varphi(\vec{x}[1..d], \vec{\xi})$ can be eliminated from the expression of $\mathbf{Q}^p_{SR}(\vec{y})$ if $d \leq p$.*

Proof $\mathbf{Q}^p_{SR}(\vec{y})$ is defined in (9). The sequencing predicate can be written as (3) or (4). Then, two cases may occur:

- If $d < p$, then either (3) or (4) implies $\vec{x}[1..d+1] = \vec{y}[1..d+1]$. Thus:

$$(15) \Rightarrow \vec{x}[d+1] \leq \varphi(\vec{x}[1..d], \vec{\xi}).$$

- If $d = p$, let us observe first that $d < N_{SR}$. Thus, $p < N_{SR}$, which means that the sequencing predicate $\prec_p$ at depth p is in the form (3) again. So (3) $\Rightarrow \vec{y}[1..p] = \vec{x}[1..p] \wedge \vec{x}[p+1] < \vec{y}[p+1]$. Thus, (15) $\Rightarrow \vec{x}[p+1] < \varphi(\vec{x}[1..p], \vec{\xi})$.

□

If all the φ's can be removed from S's existence predicate, then $\mathbf{Q}^p_{SR}(\vec{y})$ is a convex polyhedron and the corresponding direct dependence can be exactly computed.

In Program N, an operation $\langle S_1, \imath', w', \jmath' \rangle$ is a possible source of `a(j-2)` in operation $\langle S_1, \imath, w, \jmath \rangle$ if:

$$1 \leq \imath' \leq n, 1 \leq w' \leq \varphi(\imath', \vec{\xi}), \imath' \leq \jmath' \leq n+2$$

and,

$$\imath' < \imath \quad (16)$$
$$\vee(\imath' = \imath \wedge w' < w) \quad (17)$$
$$\vee(\imath' = \imath \wedge w' = w \wedge \jmath' < \jmath) \quad (18)$$
$$\vee(\imath' = \imath \wedge w' = w \wedge \jmath' = \jmath \wedge T_{S_1 S_1}) \quad (19)$$

The environment is

$$E = \{1 \leq i \leq n, 1 \leq w \leq \varphi(\imath, \vec{\xi}), \imath \leq \jmath \leq n+2\}. \quad (20)$$

Since $N_{S_1 S_1} = 3$, direct dependences may occur at depths 0, 1, 2 and 3:

$p = 3$: The sequencing condition is (19). $T_{S_1 S_1}$ is false, hence $\mathbf{Q}^3_{S_1 S_1}(\imath, w, \jmath) = \emptyset$.

$p = 2$: The sequencing condition is (18). Since $\imath' = \imath$ and $w' = w$, the inequalities $1 \leq w' \leq \varphi(\imath', \vec{\xi})$ are implied by the environment and can be discarded in the definition of this set. Then, $\mathbf{Q}^2_{S_1 S_1}(i, w, j) = \{(\imath', w', \jmath') | i \leq \jmath' \leq n+2, \imath' = \imath, w' = w, \jmath' < \jmath, \jmath' = \jmath - 2\}$. Its maximum can thus be computed:

$$K^{-2}_{S_1 S_1}(\imath, w, \jmath) = \left| \begin{array}{l} \textbf{if } \jmath \geq \imath + 2 \\ \textbf{then } (\imath, w, \jmath - 2) \\ \textbf{else } \perp \end{array} \right. .$$

Equivalently,

$$\mathcal{S}^2_{S_1 S_1}(\imath, w, \jmath) = \left| \begin{array}{l} \textbf{if } \jmath \geq \imath + 2 \\ \textbf{then } \langle S_1, \imath, w, j - 2 \rangle \\ \textbf{else } \perp \end{array} \right. \quad (21)$$

$p = 1$: (17) implies that:

$$\mathbf{Q}^1_{S_1 S_1}(\imath, w, \jmath) = \{(\imath', w', \jmath') | 1 \leq \imath' \leq n, 1 \leq w' \leq \varphi(\imath', \vec{\xi}), \imath' \leq \jmath' \leq n+2, \imath' = \imath, w' < w, \jmath' = \jmath - 2\}$$

Again, $1 \leq w' \leq \varphi(\imath', \vec{\xi})$ is implied by (20) and can be discarded in this set expression. The corresponding possible source is:

$$\mathcal{S}^1_{S_1 S_1}(i, w, j) = \left| \begin{array}{l} \textbf{if } w \geq 2 \wedge \jmath \geq \imath + 2 \\ \textbf{then } \langle S_1, \imath, w - 1, \jmath - 2 \rangle \\ \textbf{else } \perp \end{array} \right. \quad (22)$$

$p = 0$: This case cannot be handled exactly (see 5.1.2).

Let us now consider the direct dependences from S_0 to S_1. The reader may check that the two direct dependences at depth 0 and 1 are:

$$\mathcal{S}^0_{S_0 S_1}(i, w, \jmath) = \left| \begin{array}{l} \textbf{if } \jmath \geq 3 \wedge \imath \geq j - 1 \\ \textbf{then } \langle S_0, \jmath - 2 \rangle \\ \textbf{else } \perp \end{array} \right. \quad (23)$$

and

$$\mathcal{S}^1_{S_0 S_1}(i, w, \jmath) = \textbf{if } \jmath = i + 2 \textbf{ then } \langle S_0, \imath \rangle \textbf{ else } \perp. \quad (24)$$

Note that if the only `while` in the source program is the outermost loop, then $d \leq p$ always holds, and Property 1 proves that the dataflow analysis is exact. This result justifies a conjecture in [3].

The case of conditionals. A similar result holds for conditionals. Let C be a conditional at depth c enclosing two statements S and R. S and R are thus governed by the same predicate τ, meaning that the environment includes $\tau(\vec{y}[1..c], \vec{\xi})$ while the definition of the candidate set $\mathbf{Q}^p_{SR}(\vec{y})$ includes $\tau(\vec{x}[1..c], \vec{\xi})$. Hence, if $p \geq c$, then $\vec{y}[1..c] = \vec{x}[1..c]$ and the former predicate implies the latter, which can be eliminated.

Similarly, if S and R are in opposite branches of C, then the definition of the candidate set includes $\neg\tau(\vec{x}[1..c], \vec{\xi})$. If $p \geq c$, this predicate is always false and the candidate set is empty.

In favorable cases, all φ and τ functions can be eliminated and the computation of an exact source is possible. For static control programs, the hypotheses of this section are trivially verified: this is the stage at which fuzzy ADA and exact ADA meet. The following section deals with cases where such simplifications are not possible

5.1.2 Expressing an imprecise solution

Section 2 gave an intuitive flavor of what the final result of FADA should be: sets of possible sources. Our aim, however is to postpone this use of sets so as to keep exact information as far as possible.

Let c be the depth of the innermost condition governing S, and let d be the depth of the innermost **while** loop enclosing S. Let us suppose that at least one of c and d is larger than the current depth p (if not, as we have seen, an exact computation is possible). A central property is that in computing approximations, we have only to consider the innermost test or **while** loop. To see this, suppose for instance that the source in question is governed by a test with predicate $\tau(\vec{x}[1..c])$ containing a **while** loop whose upper bound is $\varphi(\vec{x}[1..d])$, with $d \geq c$. Let us define a new function:

$$\psi(\vec{x}[1..d]) = \textbf{if } \tau(\vec{x}[1..c]) \textbf{ then } \varphi(\vec{x}[1..d]) \textbf{ else } 0.$$

The iteration domain associated to ψ is exactly the same as the one associated to φ and τ. Since ψ is no more and no less arbitrary than φ and τ, we may proceed as if the unique governing construction was the **while** loop. We can then proceed in a case by case manner.

The case of while loops. Suppose first that $c \leq d$, which means that the innermost **while** is inside the innermost test. The problem here is to express the set $\mathbf{Q}^p_{SR}(\vec{y}, \vec{\xi})$ which now depends on the hidden variables.

$$\begin{aligned}\mathbf{Q}^p_{SR}(\vec{y}, \vec{\xi}) \;=\; & \{\vec{x} \mid A\vec{x} \geq \vec{b},\ 1 \leq \vec{x}[d+1] \leq \varphi(\vec{x}[1..d], \vec{\xi}),\\ & f(\vec{x}) = g(\vec{y}), \langle S, \vec{x}\rangle \prec_p \langle R, \vec{y}\rangle\}\end{aligned}$$

where $A\vec{x} \geq \vec{b}$ subsumes the linear part of the existence predicate of S. Let $\mathcal{T}(\vec{\xi})$ be the subset of the iteration domain of the **while** where the loop is executed at least once:

$$\mathcal{T}(\vec{\xi}) = \{\vec{a} \mid \varphi(\vec{a}, \vec{\xi}) > 0\}.$$

The set of candidate sources at depth p is:

$$\mathbf{Q}^p_{SR}(\vec{y}, \vec{\xi}) = \bigcup_{\substack{\vec{a} \in \mathcal{T}(\vec{\xi}),\\ b = \varphi(\vec{a}, \vec{\xi})}} \{\, \vec{x} \mid A\vec{x} \geq \vec{b}, \vec{x}[1..d] = \vec{a},\ 1 \leq \vec{x}[d+1] \leq b,\ f(\vec{x}) = g(\vec{y}), \langle S, \vec{x}\rangle \prec_p \langle R, \vec{y}\rangle\}. \tag{25}$$

The lexicographical maximum of the above union belongs to one of the sets of the union. Hence, there exist $\vec{\alpha}$ in $\mathcal{T}(\vec{\xi})$, β such that the direct dependence $K^p_{SR}(\vec{y}, \vec{\xi})$ from S to R at depth p is the lexicographical maximum of the polyhedron:

$$\begin{aligned}\{\vec{x} \mid & A\vec{x} \geq \vec{b}, \vec{x}[1..d] = \vec{\alpha}, 1 \leq \vec{x}[d+1] \leq \beta,\\ & f(\vec{x}) = g(\vec{y}), \langle S, \vec{x}\rangle \prec_p \langle R, \vec{y}\rangle\}.\end{aligned}$$

We may consider that the values of $\vec{\alpha}$ and β are a "summary" of the values of the hidden variables as far as the current **while** loop is concerned. The set of candidate sources may be written $\mathbf{Q}^p_{SR}(\vec{y}, \vec{\alpha}, \beta)$ instead of $\mathbf{Q}^p_{SR}(\vec{y}, \vec{\xi})$. Its maximum $K^p_{SR}(\vec{y}, \vec{\alpha}, \beta)$ can easily be computed by a software like PIP [5] as a function of $\vec{y}$, of the original structure parameters, and of the additional parameters $\vec{\alpha}$ and β. Furthermore, it is easy to see that there exists a φ function, namely:

$$\begin{aligned}\varphi(\vec{a}, \vec{\xi}) \;&=\; \beta \text{ if } \vec{a} = \vec{\alpha},\\ &=\; 0 \text{ otherwise}\end{aligned}$$

such that $K^p_{SR}(\vec{y}, \vec{\xi}) = K^p_{SR}(\vec{y}, \vec{\alpha}, \beta)$. This shows that our parametric representation gives all possible sources and nothing but sources. The "fuzziness" of the solution is reflected in the fact that the values of the additional parameters $\vec{\alpha}$ and β are unknown.

> In Program **N**, the direct dependence from S_1 to itself at depth 0 has the following parametric candidate set:
>
> $$\begin{aligned}&\mathbf{Q}^0_{S_1 S_1}(i, w, j, \alpha, \beta) =\\ &\quad \{(i', w', j') \mid 1 \leq i' \leq n, i' = \alpha,\\ &\quad 1 \leq w' \leq \beta, i' \leq j' \leq n+2,\\ &\quad j' = j-2, i' < i\}.\end{aligned}$$
>
> Here is the solution:
>
> $$\mathcal{S}^0_{S_1 S_1}(i, w, j, \alpha, \beta) = \left| \begin{array}{l} \textbf{if } 1 \leq \alpha < i \wedge \beta \geq 1 \wedge j \geq \alpha + 2\\ \textbf{then } \langle S_1, \alpha, \beta, j-2\rangle\\ \textbf{else } \bot \end{array} \right. \tag{26}$$

The case of conditionnals: the if..then construct. Suppose next that $c > d$. Let S be a statement which writes into a and which is in the branch of a conditionnal governed by some predicate τ. We assume that $p < c$ (otherwise, the results of Section 5.1.1 apply). With the same notations as above, the set of candidates is given by:

$$\begin{aligned}\mathbf{Q}^p_{SR}(\vec{y}, \vec{\xi}) \;=\; & \{\vec{x} \mid A\vec{x} \geq \vec{b},\ \tau(\vec{x}[1..c], \vec{\xi}),\\ & f(\vec{x}) = g(\vec{y}), \langle S, \vec{x}\rangle \prec_p \langle R, \vec{y}\rangle\}.\end{aligned} \tag{27}$$

Let $\mathcal{T}(\vec{\xi})$ be the subset of the iteration domain of the conditional where its predicate evaluates to true. Then τ is defined by:

$$\tau(\vec{x}[1..c], \vec{\xi}) = \bigvee_{\vec{a} \in \mathcal{T}(\vec{\xi})} (\vec{a} = \vec{x}[1..c]).$$

The set of candidate sources at depth p can be written as.

$$\mathbf{Q}^p_{SR}(\vec{y},\vec{\xi}) = \bigcup_{\vec{a}\in\mathcal{T}(\vec{\xi})} \{\vec{x}|A\vec{x}\geq\vec{b}, \vec{x}[1..c]=\vec{a}, f(\vec{x})=g(\vec{y}), \langle S,\vec{x}\rangle \prec_p \langle R,\vec{y}\rangle\}.$$

The lexicographical maximum of the above union belongs to one of the sets of the union. Hence there exists $\vec{\alpha}$ in $\mathcal{T}(\vec{\xi})$ such that the direct dependence $K^p_{SR}(\vec{y},\vec{\xi})$ from S to R at depth p is the lexicographical maximum of

$$\{\vec{x}|A\vec{x}\geq\vec{b}, \vec{x}[1..c]=\vec{\alpha}, f(\vec{x})=g(\vec{y}), \langle S,\vec{x}\rangle \prec_p \langle R,\vec{y}\rangle\}. \quad (28)$$

Here again, it is easy to see that this procedure gives all sources and nothing but sources. When τ evaluates to false for all values of its iteration domain, we may choose for $\vec{\alpha}$ a value which does not satisfy $A\vec{\alpha}\geq\vec{b}$. This can always be done since 0 is not in the domain of any outer `while` loop and the other outer loops have bounded domains

The case of conditionnals: the `if..then..else` construct. The situation is more complicated in this case. If no variable is modified in both arms of the conditional, we can handle each arm independently along the lines of the preceding paragraph. Suppose now that there exists an array `a` which is modified in both arms of the conditional and which is read later:

```
      do x⃗ = . .
           if (P) then
S1:              a(f1(x⃗)) = ...
           else
S2:              a(f2(x⃗)) = ...
           endif
R:         ... = x(g(x⃗))
      enddo
```

With notations similar to (27), we have two sets of candidate sources:

$$\mathbf{Q}^p_{S_1R}(\vec{y},\vec{\xi}) = \{\vec{x}|\, A_1\vec{x}\geq\vec{b}_1,\ \tau(\vec{x}[1..c],\vec{\xi}), f_1(\vec{x})=g(\vec{y}), \langle S_1,\vec{x}\rangle \prec_p \langle R,\vec{y}\rangle\}.$$

$$\mathbf{Q}^p_{S_2R}(\vec{y},\vec{\xi}) = \{\vec{x}|\, A_2\vec{x}\geq\vec{b}_2,\ \overline{\tau}(\vec{x}[1\ \ c],\vec{\xi}), f_2(\vec{x})=g(\vec{y}), \langle S_2,\vec{x}\rangle \prec_p \langle R,\vec{y}\rangle\}.$$

where

$$\overline{\tau} = \neg\tau. \quad (29)$$

To handle each candidate set independently, we introduce two extra parameters $\vec{\alpha}_1$ and $\vec{\alpha}_2$ such that:

$$\tau(\vec{x}[1..c]) \equiv (\vec{x}[1..c]=\vec{\alpha}_1),$$
$$\overline{\tau}(\vec{x}[1..c]) \equiv (\vec{x}[1..c]=\vec{\alpha}_2).$$

The solution then proceeds as in the `if..then..` case. In so doing, we have lost property (29), with the consequence that spurious sources will be introduced. The results can be slightly improved in some cases by observing that since $\tau(\vec{\alpha}_1)$ and $\overline{\tau}(\vec{\alpha}_2)$ are both true, if $\overline{\tau}=\neg\tau$ then $\vec{\alpha}_1\neq\vec{\alpha}_2$. This information may be added to the context and used later when combining direct dependences.

A better result can be obtained in the frequent case where the candidate set $\mathbf{Q}^p_{S_1R}(\vec{y},\vec{\xi})$ is such that the first c components of $\vec{x}$ are uniquely determined: let ϕ_1 be the function such that:

$$A_1\vec{x}\geq\vec{b}_1,\ f_1(\vec{x})=g(\vec{y}), \qquad \langle S_1,\vec{x}\rangle \prec_p \langle R,\vec{y}\rangle \Rightarrow \vec{x}[1..c]=\phi_1(\vec{y}),$$

with a similar definition for ϕ_2. Such functions can be efficiently obtained, if they exist, by extracting the implied equalities from the above system of constraint. It is easy to see that the lexicographical maximum of $\mathbf{Q}^p_{S_1R}(\vec{y},\vec{\xi})$ is:

$$\textbf{if } \tau(\phi_1(\vec{y}),\vec{\xi}) \textbf{ then } \max\mathbf{Q}_1(\vec{y}) \textbf{ else } \bot,$$

where

$$\mathbf{Q}_1(\vec{y}) = \{\vec{x}|\, A_1\vec{x}\geq\vec{b}_1, f_1(\vec{x})=g(\vec{y}), \langle S_1,\vec{x}\rangle \prec_p \langle R,\vec{y}\rangle\},$$

with a similar formula for the second arm of the conditional. In these formulas, we may select arbitrarily the values of $\tau(\phi_1(\vec{y}),\vec{\xi})$ and $\neg\tau(\phi_2(\vec{y}),\vec{\xi})$, unless $\phi_1(\vec{y})=\phi_2(\vec{y})$. Any formula which depends on one parameter and which selects in turn the right solutions can be used as a parametric representation of the solution. One such formula is:

```
if α ≠ φ1(y⃗)
then max Q1(y⃗)
     if α = φ2(y⃗)
else then max Q2(y⃗)
     else ⊥
```

Observe that if $\phi_1(\vec{y})\neq\phi_2(\vec{y})$ then the $\bot$ leaf can be selected by taking $\alpha=\phi_1(\vec{y})\neq\phi_2(\vec{y})$ while if $\phi_1(\vec{y})=\phi_2(\vec{y})$, there is no value of α which selects this leaf.

5.2 Combining direct dependences

In the previous section, we studied direct dependences, i.e. the case where only one statement may be the source of the array cell read by R. In the general case, of course, many statements are potential sources. Let $S_1,...,S_m$ be the m statements writing into a given array cell. We suppose that, for each S_k, $k=1,..,m$, the possible solutions $\mathcal{S}^p_{S_kR}$ have already been computed. We now consider the problem of finding the unique source, which is

the maximal element among these solutions according to the sequential order.

$$\sigma(\langle R, \vec{y}\rangle) = \max\{\mathcal{S}^{\nu}_{S_k R}(\vec{y}) \mid 1 \le k \le m, 0 \le p \le N_{S_k R}\}.$$

Let n be the number of candidate sources $\mathcal{S}^{p}_{S_k R}(\vec{y})$. For expository reasons, we assign an index number $q, 1 \le q \le n$ to each $\mathcal{S}^{\nu}_{S_k R}(\vec{y})$, and rename the latter into $\mathcal{S}_q$. Then, the basic algorithm computes the following recurrence.

$$1 \le q \le n \qquad \mathcal{R}_q = \max(\mathcal{R}_{q-1}, \mathcal{S}_q).$$

with

$$\mathcal{R}_0 = \bot$$

Each recurrence step has to compute the maximum of two quasts. This is done with the help of the following rules[1]

Rule 1 $\max(\bot, v) = v$. *(This is simply a restatement of (12))*

Rule 2 *If* $u =$ **if** C **then** u_1 **else** u_2, *then:*

$$\max(u, v) = \textbf{if } C \textbf{ then } \max(u_1, v) \textbf{ else } \max(u_2, v)$$

Rule 3 *If* $u = \langle S, \vec{x}\rangle$ *and* $v = \langle R, \vec{y}\rangle$ *are elementary sources, then* $\max(u, v) =$ **if** $u \prec v$ **then** v **else** u

Rule 4 *Let* **if** p **then** u **else** v *be a subtree of a quast, and let* $\mathcal{C}$ *be its context (i.e. the set of predicates which are encountered on the unique path from the root to the subtree). Then if* $\mathcal{C} \wedge p$ *is not feasible, replace the subtree by* v. *Similarly if* $\mathcal{C} \wedge \neg p$ *is not feasible, replace the subtree by* u.

Rule 5 **if** C **then** u **else** $u = u$.

We can now combine the intermediate results for the running example.

$$\mathcal{R}_1 = (24) = \textbf{if } \jmath = \imath + 2 \textbf{ then } \langle S_0, \jmath - 2\rangle \textbf{ else } \bot$$

$$\mathcal{R}_2 = \max(\mathcal{R}_1, (23))$$

$$\Downarrow \quad \textit{Rule 2}$$

$$\mathcal{R}_2 = \begin{array}{|l} \textbf{if } \jmath = \imath + 2 \\ \quad \textbf{then} \begin{array}{|l} \textbf{if } \jmath \ge 3 \wedge \jmath \le \imath + 1 \\ \quad \textbf{then } \max(\langle S_0, \jmath - 2\rangle, \langle S_0, \jmath - 2\rangle) \\ \quad \textbf{else } \max(\langle S_0, \jmath - 2\rangle, \bot) \end{array} \\ \quad \textbf{else} \begin{array}{|l} \textbf{if } \jmath \ge 3 \wedge \jmath \le \imath + 1 \\ \quad \textbf{then } \max(\bot, \langle S_0, \jmath - 2\rangle) \\ \quad \textbf{else } \max(\bot, \bot) \end{array} \end{array}$$

$$\Downarrow \quad \textit{Rules 4 and 1}$$

$$\mathcal{R}_2 = \begin{array}{|l} \textbf{if } \jmath = \imath + 2 \\ \textbf{then } \langle S_0, \jmath - 2\rangle \\ \textbf{else} \begin{array}{|l} \textbf{if } \jmath \ge 3 \wedge \jmath \le \imath + 1 \\ \quad \textbf{then } \langle S_0, \jmath - 2\rangle \\ \quad \textbf{else } \bot \end{array} \end{array}$$

[1] Rules 1 and 2 have symmetrical counterparts which are not stated here.

For expository reasons, we simplify $\mathcal{R}_2$ into:

$$\mathcal{R}_2 = \textbf{if } \jmath \le \imath + 2 \wedge \jmath \ge 3 \textbf{ then } \langle S_0, \jmath - 2\rangle \textbf{ else } \bot$$

Then,

$$\mathcal{R}_3 = \max(\mathcal{R}_2, \mathcal{S}^2_{S_1 S_1}(\imath, w, \jmath))$$

$$= \max(\mathcal{R}_2, (21))$$

$$\Downarrow \quad \textit{Rules 2, then 3 and 1}$$

$$\mathcal{R}_3 = \begin{array}{|l} \textbf{if } \jmath \le \imath + 2 \wedge \jmath \ge 3 \\ \quad \textbf{then} \begin{array}{|l} \textbf{if } \jmath \ge \imath + 2 \\ \quad \textbf{then } \langle S_1, \imath, w, \jmath - 2\rangle \\ \quad \textbf{else } \langle S_0, \jmath - 2\rangle \end{array} \\ \quad \textbf{else} \begin{array}{|l} \textbf{if } \jmath \ge \imath + 2 \\ \quad \textbf{then } \langle S_1, \imath, w, \jmath - 2\rangle \\ \quad \textbf{else } \bot \end{array} \end{array}$$

$$\mathcal{R}_4 = \max(\mathcal{R}_3, \mathcal{S}^1_{S_1 S_1}(\imath, w, \jmath))$$

$$= \max(\mathcal{R}_3, (22))$$

$$\Downarrow \quad \textit{Rules 1, 2, 3, 4 then 5}$$

$$\mathcal{R}_4 = \begin{array}{|l} \textbf{if } \jmath \le \imath + 2 \wedge \jmath \ge 3 \\ \quad \textbf{then} \begin{array}{|l} \textbf{if } \jmath \ge \imath + 2 \\ \quad \textbf{then } \langle S_1, \imath, w, \jmath - 2\rangle \\ \quad \textbf{else } \langle S_0, \jmath - 2\rangle \end{array} \\ \quad \textbf{else} \begin{array}{|l} \textbf{if } \jmath \ge \imath + 2 \\ \quad \textbf{then } \langle S_1, \imath, w, \jmath - 2\rangle \\ \quad \textbf{else } \bot \end{array} \end{array}$$

Note that $\mathcal{R}_3 = \mathcal{R}_4$. This is no surprise since the source candidates from previous iterations $w', w' < w$, of the `while` loop are masked by operations of the current iteration w. This fact can be detected before embarking on the final calculation [6, 9], thus reducing the complexity of the method. Then, $\mathcal{R}_5 = \max(\mathcal{R}_4, \mathcal{S}^0_{S_1 S_1}(\imath, w, \jmath, \alpha, \beta)) = \max(\mathcal{R}_4, (26))$. Applying Rules 1, 2, 3, 4 and 5 yields:

$$\mathcal{R}_5 = \begin{array}{|l} \textbf{if } \jmath \le \imath + 2 \wedge \jmath \ge 3 \\ \quad \textbf{then} \begin{array}{|l} \textbf{if } \jmath \ge \imath + 2 \\ \quad \textbf{then } \langle S_1, i, w, \jmath - 2\rangle \\ \quad \textbf{else} \begin{array}{|l} \textbf{if } 1 \le \alpha < \imath \wedge \beta \ge 1 \wedge \jmath \ge \alpha + 2 \\ \quad \textbf{then} \begin{array}{|l} \textbf{if } \alpha = \jmath - 2 \\ \quad \textbf{then } \langle S_1, \jmath - 2, \beta, \jmath - 2\rangle \\ \quad \textbf{else } \langle S_0, \jmath - 2\rangle \end{array} \\ \quad \textbf{else } \langle S_0, \jmath - 2\rangle \end{array} \end{array} \\ \quad \textbf{else} \begin{array}{|l} \textbf{if } \jmath \ge i + 2 \\ \quad \textbf{then } \langle S_1, i, w, \jmath - 2\rangle \\ \quad \textbf{else } \bot \end{array} \end{array} \qquad (30)$$

5.3 Removing additional parameters

The result of this analysis may be considered as the final solution of the problem, since it gives a parametric representation of the possible sources in term of additional parameters. It may, however, be more interesting to "eliminate" the additional parameters in order to distinguish clearly the cases in which the source is precisely known from those in which there are several possible solutions.

Consider a leaf in which an additional parameter appears. This leaf represents the set of sources obtained by giving all possible values to these parameters. The set of possible values is obtained by "anding" all predicates in the unique path from the root of the quast to the leaf in question.

Rule 6 *Let $A(\vec{\alpha})$ be a leaf governed by l predicates $P_1, \ldots, P_l$ in the unique path from the root to the leaf. Then $A(\vec{\alpha})$ is transformed into $\{A(\vec{\alpha}) | \bigwedge_{i=1}^{l} P_i\}$.*

We first apply this rule in a systematic fashion. Then, any leaf in which new parameters occur is transformed into a *set* in which the new parameters are bounded by the predicates governing the leaf. Leaves which does not depends on parameters become singletons.

Now consider a quast **if** $C(\vec{\alpha})$ **then** A **else** B. Thanks to Rule 6, A and B are sets. Moreover, since the exact value of $\vec{\alpha}$ is unknown, we cannot predict the outcome of the test. The best we can do is to take as an approximation the union $A \cup B$

Rule 7 *A quast* **if** $C(\vec{\alpha})$ **then** A **else** B *is transformed into* $A \cup B$.

Proceeding on (30), we apply Rule 6 transforms $\mathcal{R}_5$ into:

```
if j ≤ i + 2 ∧ j ≥ 3
    if j ≥ i + 2
    then {⟨S1, i, w, j − 2⟩}
    else if 1 ≤ α < i ∧ β ≥ 1 ∧ j ≥ α + 2
         then if α = j − 2
              then {⟨S1, j − 2, β, j − 2⟩ | β ≥ 1}
              else {⟨S0, j − 2⟩}
         else {⟨S0, j − 2⟩}
then
else if j ≥ i + 2
     then {⟨S1, i, w, j − 2⟩}
     else {⊥}
```

(We used the fact that $\{\langle S_0, j-2\rangle | 1 \leq \alpha \wedge \beta \geq 1 \wedge j \geq \alpha + 2\} = \{\langle S_0, j-2\rangle\}$.) Then, applying Rule 7 eventually yields $\mathcal{R}_5$ equals to:

```
if j ≤ i + 2 ∧ j ≥ 3
    if j ≥ i + 2
    then {⟨S1, i, w, j − 2⟩}
    else {⟨S1, j − 2, β, j − 2⟩ | β ≥ 1} ∪ {⟨S0, j − 2⟩}
then
else if j ≥ i + 2
     then {⟨S1, i, w, j − 2⟩}
     else {⊥}
```

The reader may check that this result is exactly the result intuitively found in Section 2.

Observe that if we do not simplify our parametric quasts, then leaves which are governed by inconsistent predicates give empty sets by Rule 6, and then disappear by Rule 7. This observation shows that our quast simplification rules and our parameter elimination rules are consistent.

6 Conclusions

This paper gives a method to build a conservative approximation of the flow of values in dynamic control programs. Such programs include control-flow constructs such as `whiles` and `if..then..else` constructs, making both control and data flow unpredictable at compile-time. In this paper, we have shown that we can extend the notion of a unique source to that of a source *set*, and have designed a set of algorithms which give, in many cases, surprisingly precise results. The fuzzy array dataflow analysis has been implemented in Lisp within the PAF project at PRiSM Laboratory.

Our method is generic in so far as it gives a framework for fuzzy analysis that may be adapted to most exact analysis algorithms. More importantly, the net effect of our handling of `while` loops and tests is to add *equations* to the definition of the candidate set, thus improving the probability of success of fast analysis schemes like [10, 8] We have observed in fact that the time complexity of FADA is of the same order of magnitude as ADA. Some researchers already proposed techniques to handle flow-sensitive array data-flow analysis: in [4], Duesterwald, Gupta and Soffa describe a fixed point computation to discover *may*-reaching definitions. Even though their method does not handle multi-dimensional arrays and gives only maximal distances, a fuzzy array dataflow analysis along their lines may be an interesting alternative to this paper.

Moreover, the difficulty of foreseeing the flow of data does not only lie in the control flow (i.e. in solving (6) and (8)), but also in dynamic (resp. intricate) array access patterns, e.g. indirect subscripting (resp. non-linear mappings), or aliasing. The difficulty then lies in solving (7). Maslov [9] and Pugh and Wonnacott [11] tackle non-affine array subscripts, and propose a mechanism to derive *approximate dependencies* or *upper and lower bounds on dependences*, respectively. In our case, we could have easily handled an intractable conflict equation (7) by the same parametric scheme. However, since there is no concept of depth for subscripts, this would always have lead to maximum fuzziness, an uninteresting result. The solution seems to be to apply FADA to the variables occurring in intractable subscripts, so as to derive, if possible, an equivalent depth.

Applications of FADA to automatic parallelization include scheduling (along the lines of [7]), array privatization and register allocation [4]. As a concluding remark, note that a $\perp$ in a source set points to a possible programming error. Beyond automatic parallelization, a fuzzy array dataflow analysis may therefore be a general tool for translators, compilers and program checkers, as array dataflow analysis was.

Acknowledgments: The authors acknowledge the support of CNRS Program PRS, PRC/MRE contract

ParaDigme, DRET contracts 91/1180 and 93/2545A and the French-German research program Procope.

References

[1] A. V Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass, 1986.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[3] J.-F Collard Space-time transformation of while-loops using speculative execution. In *Proc. of the 1994 Scalable High Performance Computing Conf.*, pages 429- 436, Knoxville, Tenn., May 1994. IEEE

[4] E. Duesterwald, R. Gupta, and M.-L. Soffa A practical data flow framework for array reference analysis and its use in optimization. In *ACM SIGPLAN'93 Conf. on Prog. Lang. Design and Implementation*, pages 68–77, June 1993.

[5] P Feautrier Parametric integer programming. *RAIRO Recherche Opérationnelle* 22:243–268. September 1988.

[6] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53. February 1991.

[7] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[8] C. Heckler and L Thiele. Computing linear data dependencies in nested loop programs. *Parallel Processing Letters*, 1994. To appear.

[9] V Maslov. Lazy array data-flow dependence analysis. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. POPL*, pages 311–325, January 1994.

[10] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages*, pages 2–15, January 1993

[11] W. Pugh and D. Wonnacott. An exact method for analysis of value-based data dependences. Technical Report CS-TR-3196, U. of Maryland, December 1993.