Characterization of program dependecies by integer programming technique

Yvon Jégou

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Characterization of program dependencies by integer programming techniques

Yvon Jégou

N° 2138

Novembre 1993

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

Characterization of program dependencies by integer programming techniques

Yvon Jégou

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet CALCPAR

Rapport de recherche n ° 2138 — Novembre 1993 — 18 pages

Abstract: Integer programming techniques can be used in the characterization of relations between the variables of programs. Such techniques are considered to be far too expensive to be used in an automatic optimizing tool. We show that the cost of such a technique can be balanced by the quality of the information that can be extracted. Our algorithm transforms sets of equations and constraints that are extracted, by the optimizing compiler, from programs and produces a characterization of the solution domain of the system. The result is a restricted domain of possible values for each variable as well as a set of new pertinent constraints. The system is run incrementally from the compiler. Its application domain is not limited to the computation of dependencies. In fact, it can be useful everywhere the decision of the compiler depends on the characterization of relations between variables of the program.

Key-words: dependencies, integer programming, constraints, optimizations

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Caractérisation des dépendances dans les programmes par des techniques de programmation sur les entiers

Résumé : Les techniques de programmation sur les entiers peuvent être utilisées dans la cractérisation des relations entre les variables entières des programmes, et plus particulièrement dans le calcul des dépendances. Mais on considère en général que ces techniques sont trop coûteuses pour être intégrées dans les outils automatiques. Nous montrons que ce coût peut être justifié par la qualité des relations qui peuvent être produites. Notre algorithme transforme des ensembles d'équations et de contraintes extraites par le compilateur et produit une caractérisation du domaine de solutions du système. Ce résultat contient une restriction des domaines de valeurs des variables ainsi qu'un ensemble de nouvelles contraintes plus pertinentes. Cet algorithme peut être appelé de manière incrémentale par le compilateur. Son domaine d'application n'est pas limité au seul calcul des dépendances. Il se révèle utile partout où un choix du compilateur dépend d'une caractérisation de relations entre des variables.

Mots-clé : dépendences, programmation entière, systèmes de contraintes, optimisation

1 Introduction

Many optimizations performed by modern compilers are based on a precise analysis of the memory accesses inside the programs. The optimizations we are interested in cover vectorization, parallelization, the exploitation of data reuse, data locality — temporal as well as spacial —, loop reordering, data distribution and loop distribution. A characterization of the relations between memory accesses is needed for all these transformations.

In the first generation of optimizing compilers, the dependence analysis techniques were oriented towards the detection of vector codes, and were based on the computation of simple relations. In modern systems, the loop control variables do not define the unique dimensions of the systems. Our generalized dependency test computes the relations between the variables of a system under a set of constraints. These variables can be loop control, program symbols, can represent subsets of the index space of an array, or can represent other values in the test. The application domain of the dependency characterization extends now to

- vectorization and parallelization
- data partitioning from an iteration space partitioning. Once an iteration space has been distributed on a parallel system, it becomes possible to deduce a partitioning of the data which improves locality in the transfers
- iteration space partitioning from a data distribution. In this case, the data are already distributed across the memories. As in the previous case, the locality of the accesses can be increased by an accurate partitioning of the iteration space
- improving data reuse. When the same element of an array is accessed many times in a loop nest, the compiler tries to resequence the loop nest in order to bring these accesses closer in time. Such a transformation optimizes the memory hierarchy utilisation.
- improving data locality. The optimizer computes the subset of the iteration space which contains accesses to memory locations that are close to each other.

Many authors have studied the dependency test problem, [Ban88, Wolfe89, BanWol87, Wallace88, Pugh91]. Usually, these tests were either approximated tests or data dependency oriented tests. The complexity of the algorithm was an important factor. More recently, the use of general integer programming techniques have been proposed. The application domain of the test is no longer limited to dependency detection, and extends, now, to the optimization of accesses to the memory hierarchy. Our solution belongs to this last category. Because the cost of any integer programming techniques depends strongly on the number of variables, the algorithms that have been proposed so far try to minimize this number of variables in the system. Our solution is not based on such a minimization, and in many cases, new artificial variables are introduced because their presence can accelerate the characterization of the solutions.

The next section shows how the system interacts with a compiler from the compiler's point of view. Then we describe how the equations and the constraints are represented in

section 3 and how these are transformed in section 4. Section 4 also reports on the extraction of information from the system during the transformations.

2 Principles of the generalized dependency characterizer

Characterizing the dependencies between two memory accesses corresponds to characterizing the solution of an integer linear system with constraints. The equations of the system express the equalities that must be asserted in order to access the same memory location. The constraints limit the domain of the computation. The constraints are extracted from the loop and the variable declarations. In some cases, they can also be deduced from assertions or conditional instructions.

For instance, consider the following program fragment.

```
m2 = m1-n
do i1 = 1, n
   do i2 = 1, n
      A(i1+m1) = A(i1+m1) + B(i2, i1)*A(i2+m2)
   end do
end do
```

As a first step in the optimization of these sequential loops, the accesses to the array `A` must be analyzed. The transformations that are possible on the loops depend first on the dependency on these accesses. We consider here the relations between `A(i1+m1)` and `A(i2+m2)`. The self dependency on `A(i1+m1)` must also be considered. The system of equations that must be characterized is

$$\begin{aligned} \texttt{i1} + \texttt{m1} &= \texttt{i2} + \texttt{m2} && \textit{(equality of addresses)} \\ \texttt{m2} &= \texttt{m1} - \texttt{n} && \textit{(definition of } \texttt{m2}\textit{)} \end{aligned}$$

and the constraints are

$$\begin{aligned} 1 \leq \texttt{i1} \leq \texttt{n} && \textit{(definition of } \texttt{i1}\textit{)} \\ 1 \leq \texttt{i2} \leq \texttt{n} && \textit{(definition of } \texttt{i2}\textit{)} \end{aligned}$$

When our dependency characterization algorithm is applied to this system of equations and constraints, it tries to compute the domain of possible solutions for each variable. In our system, all the variables have the same status.

The first step of the dependency characterization transforms the constraints which contain more than one variables into equations after the introduction of a spare variable. Thus, in our example, the constraint $\texttt{i1} \leq \texttt{n}$ is transformed into equation $\texttt{i1} - \texttt{n} = t1$ and the new simple constraint $t1 \leq 0$. After this transformation, the remaining constraints espress bounds on the variables. They are applied to the variable descriptions and are removed from the system. After this step, our system becomes

$$\begin{array}{rl|rcl} \mathtt{i1}+\mathtt{m1}-\mathtt{i2}-\mathtt{m2}=0 & & 1 & \leq & \mathtt{i1} \\ \mathtt{m2}-\mathtt{m1}-\mathtt{n}=0 & & 1 & \leq & \mathtt{i2} \\ \mathtt{i1}-\mathtt{n}-t1=0 & & t1 & \leq & 0 \\ \mathtt{i2}-\mathtt{n}-t2=0 & & t2 & \leq & 0 \end{array}$$

The transformations described in 4 are applied to the equations which become

$$\begin{array}{rcl|rcl} \mathtt{m1}+\mathtt{i1}-\mathtt{i2}-\mathtt{m2} & = & 0 & 1 & \leq & \mathtt{i1} \\ -t2+\mathtt{i1} & = & 0 & 1 & \leq & \mathtt{i2} \\ -\mathtt{n}-\mathtt{i1}+\mathtt{i2} & = & 0 & t1 & \leq & 0 \\ t1-2\mathtt{i1}+\mathtt{i2} & = & 0 & t2 & \leq & 0 \end{array}$$

The domain reduction step computes new bounds for each variable in the system. From equation $-t2+\mathtt{i1}=0$ and the bounds $t2 \leq 0, 1 \leq \mathtt{i1}$, new bounds are computed for $t2$ and $\mathtt{i1}$: $1 \leq t2 \leq 0$ and $1 \leq \mathtt{i1} \leq 0$. Obviously, the set of solutions is empty. This means that the reads on `A(i2+m2)` do not access the same memory locations as the writes on `A(i1+m1)`: they are independent. The compiler can conclude that the outer loop can be run in parallel.

Usually, the decision process of a compiler is not limited to the choice between a sequential interpretation of a loop and a parallel one. The optimization process is more complex. For instance, consider the following simple loop.

```
n=100
do i= 1, n
   a(i) = a(n-i)
end do
```

The corresponding system is

$$\begin{array}{l|l} \mathtt{i}_1=\mathtt{n}-\mathtt{i}_2 & 1 \leq \mathtt{i}_1 \leq \mathtt{n} \\ \mathtt{n}=100 & 1 \leq \mathtt{i}_2 \leq \mathtt{n} \end{array}$$

In this system, $\mathtt{i}_1$ and $\mathtt{i}_2$ represent two different occurences of the same variable `i`. This system has solutions but the domain of solutions of each variable is its domain of definition: the iterations of this loop cannot be run independently. This is the first step in the decision process of the compiler. Dependent does not mean sequential. Parallel interpretation may still be possible if some relation can be enforced between the iterations. Then, for the second step, we can add some constraint on $\mathtt{i}_1$ and $\mathtt{i}_2$. The sequential loop contains two kinds of dependencies on the accesses to `a` : RaW (Read after Write) on some locations and WaR (Write after Read) on some other locations. Let us consider the RaW dependencies. Some element of `a` are written at iteration $\mathtt{i}_1$ in the sequential loop before being read at iteration $\mathtt{i}_2$; This is expressed by the constraint $\mathtt{i}_1 < \mathtt{i}_2$. So this constraint is added to the system which is now

$$\begin{array}{l|l} \mathtt{i}_1=\mathtt{n}-\mathtt{i}_2 & 1 \leq \mathtt{i}_1 \leq \mathtt{n} \\ \mathtt{n}=100 & 1 \leq \mathtt{i}_2 \leq \mathtt{n} \\ & \mathtt{i}_1 < \mathtt{i}_2 \end{array}$$

The resulting domains of values for $\mathtt{i}_1$ and $\mathtt{i}_2$ are now $1 \leq \mathtt{i}_1 \leq 49$ and $51 \leq \mathtt{i}_2 \leq 100$. The RaW dependency is present only on a part of vector `a`. These results show on which part of the vector there is a dependency – the domain of `i1` –, inside which subset of the iteration space these variables are written – the domain of `i1` – and in which part they are read – the domain of `i2`. At this stage, a possible decision for the compiler may be to split the iteration space in two or three parts. This transformation must be validated for the other dependencies that exist in the same loop.

Suppose the RaW dependency was the only one in the loop. The iteration space can be cut into three subsets :

a write subset containing elements of the domain of $\mathtt{i}_1$ but no element from the domain of $\mathtt{i}_2$.

a read subset containing elements of the domain of $\mathtt{i}_2$ but no element from the domain of $\mathtt{i}_1$.

a read / write subset containing the intersection of the two domains.

The iterations which do not belong to the domain of $\mathtt{i}_1$ or to the domain of $\mathtt{i}_2$ can be associated with any subset.

We have got three loops. The write subset is executed first. It should be runnable in parallel (no self dependency). After the write subset, the read/write subset is executed and is sequential. The read subset is then executed and is parallel. In this example, the read / write subset is empty.

```
n=100
do i= 1, 50
   a(i) = a(n-i)
end do
do i= 51, 100
   a(i) = a(n-i)
end do
```

After this transformation, each loop is optimized independently, and both are found to be parallel. In each subset, the iterations need not be contiguous in order to generate such a transformation. However, contiguous subsets are easier to implement because they do not modify directly the order of memory accesses. The algorithm is simpler.

In this example, we have shown the guideline of our dependency analysis system. In order to be integrated inside the decision process, it must interact with the optimizing compiler. When a dependency is to be analyzed, the compiler sends a set of equations and a set of constraints to the solver and then asks for the domain of solutions for one or more variables. The compiler analyzes these results and can send new equations and new constraints to the system. This interaction may continue until the compiler decides that it has enough knowledge on the dependency to make the decision.

The key point in the interaction between the compiler and the dependency system is in the expression of solution domains. In our simple examples, these results were intervals

of integers. We will see that this is not the only possible characterization of the solution domain: new constraints involving two or more variables can also be produced. In fact, the dependency characterizer represents the value domain of each internal variable as an interval. Each external variable (a variable known from the compiler) corresponds to an internal variable. In the resolution process, artificial variables can also be introduced by the characterizer, which have no meaning outside. The relations between internal variable x'_i and external variable x_i is defined using two constants s_i and c_i by the equation $x_i = s_i x'_i + c_i$: the value domain of an external variable is sparse.

On request from the compiler, the system also produces lists of constraints on subsets of the variables. Consider the previous example when the value of `n` is not known at compile time.

$$\mathtt{i}_1 = \mathtt{n} - \mathtt{i}_2 \quad \left| \quad \begin{array}{r} 1 \leq \mathtt{i}_1 \leq \mathtt{n} \\ 1 \leq \mathtt{i}_2 \leq \mathtt{n} \\ \mathtt{i}_1 < \mathtt{i}_2 \end{array} \right.$$

The result domains are now $[1, 1073676288]$ for $\mathtt{i}_1$, $[2, 2147352577]$ for $\mathtt{i}_2$ and $[3, 2147352578]$ for `n`. The large values of the upper bounds have been computed from the largest representable integer for the variables of the program $(2^{31} - 1)$. These results cannot be exploited. The compiler needs the relations between the loop variables and the loop bounds in order to make a decision. Such an information can be obtained by a constraint request. The constraint request lists two or more variables and the characterization system computes constraints involving these variables. In our case, to a relation request on $\mathtt{i}_1$ and `n`, the system produces the constraints

$$\begin{array}{r} 2 \leq -\ \mathtt{i}_1 + \mathtt{n} \leq 2147352577 \\ -2147352576 \leq\ 2\ \mathtt{i}_1 - \mathtt{n}\ \leq -1 \end{array}$$

The first constraint comes from the initial constraints and does not contain much more information. The second relation is more powerful because it indicates a part of the iteration space. Each time a constraint of the form $\alpha\mathtt{i} \leq \mathtt{n} - C$ between a loop variable `i` and the upper bound `n` is obtained, the iteration domain of a loop can be cut at the point $\lfloor \frac{\mathtt{n}-C}{\alpha} \rfloor$. The resulting loop is now

```
do i= 1, n/2
   a(i) = a(n-i)
end do
do i= n/2+1, n
   a(i) = a(n-i)
end do
```

The presence of symbols in the equations is not an effective limitation to the resolution of the system.

3 System representation

3.1 Equations and constraints

The dependency algorithm manipulates equations and internal variable domains. An internal variable domain is defined by a lower bound and an upper bound. The interface between the system and the compiler associates the internal variables with the external ones by a multiplicative factor s and a displacement c. The first time an external variable is met, the value s is fixed to one, c to zero and the bounds are extracted from the equation or the constraint. Where the bounds cannot be computed they are fixed to be the largest and the lowest representable integers in the language. The inequations of two or more variables are transformed into equations by the introduction of spare variables. For instance, $expr_1 \leq expr_2$ is transformed into $expr_1 + S_1 = expr_2$. The initial domain of S_1 is $[0, u]$ where u is computed from $expr_1$ and $expr_2$.

The equations are stored as usual in an array of integers. Because equation or variable exchanges are frequent during the computation, the equations and variables are accessed through renumbering vectors.

Consider the fragment

```
do i=1,n
   do j=i,n
      a(i,j) = a(j, i) ...
   end do
end do
```

The equations and the constraints are

$$\begin{array}{r|l} & 1 \leq \mathtt{i}_1 \leq \mathtt{n} \\ \mathtt{i}_1 = \mathtt{j}_2 & \mathtt{i}_1 \leq \mathtt{j}_1 \leq \mathtt{n} \\ \mathtt{j}_1 = \mathtt{i}_2 & 1 \leq \mathtt{i}_2 \leq \mathtt{n} \\ & \mathtt{i}_2 \leq \mathtt{j}_2 \leq \mathtt{n} \end{array}$$

In the dependency system the problem is represented by the equations and the domains

$$\begin{array}{r|l} & \mathtt{i}_1 : [1, 2147352578] \\ \mathtt{i}_1 - \mathtt{j}_2 = 0 & \mathtt{i}_2 : [1, 2147352578] \\ \mathtt{j}_1 - \mathtt{i}_2 = 0 & \mathtt{j}_1 : [1, 2147352578] \\ \mathtt{j}_1 - \mathtt{i}_1 - S_1 = 0 & \mathtt{j}_2 : [1, 2147352578] \\ \mathtt{j}_2 - \mathtt{i}_2 - S_2 = 0 & S_1 : [0, 2147352577] \\ \mathtt{i}_1 + S_3 - \mathtt{n} = 0 & S_2 : [0, 2147352577] \\ \mathtt{i}_2 + S_4 - \mathtt{n} = 0 & S_3 : [0, 2147352577] \\ \mathtt{j}_1 + S_5 - \mathtt{n} = 0 & S_4 : [0, 2147352577] \\ \mathtt{j}_2 + S_6 - \mathtt{n} = 0 & S_5 : [0, 2147352577] \\ & S_6 : [0, 2147352577] \end{array}$$

3.2 The variables

The semantics of the variables in the characterization system are not related to the semantics of corresponding variables in the program. All the variables have the same status. For instance, a variable representing a loop control and a variable representing a loop bound have the same status. Moreover, artificial variables – variables without any meaning in the source program – can be introduced during the computation by the system or by the compiler. The artificial variables can represent information that is known from the compiler or that is to be computed by the system.

dependance direction and distance

A usual use of such an artificial variable is in the dependency distance computation. When $\mathtt{i}_1$ and $\mathtt{i}_2$ represent two instances of the same loop control variable `i`, equation $di = \mathtt{i}_1 - \mathtt{i}_2$ can be systematically introduced by the compiler. An analysis of the domain of di produces information on the dependency (direction, possible distances).

loop reordering

Because the loop control variable have no special status, classical implicit ordering such as lexicographic ordering of a nest of iterations has no interpretation in our system. However, equations and variables representing such information can be introduced. A variable can represent the date associated with some program event. Then relations between the dates of these events can be evaluated. Let us consider the classical loop

```
do i=1,n
   do j=1,n
      a(i,j)= a(i-1,j)+a(i,j-1)
   end do
end do
```

This code section is sequential. This can be expressed by dating the events. For instance, one can introduce the variable d and equation $d = \mathtt{i} \times M_n + \mathtt{j}$ which associates the date value d with iteration `(i,j)`. Here, M_n is a constant with $M_n \geq n$ — in general, the known upper bound of the variable can be used.

The array element $\mathtt{a}(\mathtt{i}_1, \mathtt{j}_1)$ is written during iteration $(\mathtt{i}_1, \mathtt{j}_1)$ at date $d_1 = \mathtt{i}_1 \times M_n + \mathtt{j}_1$. The same element is read during iteration $(\mathtt{i}_2, \mathtt{j}_2)$ and during iteration $(\mathtt{i}_3, \mathtt{j}_3)$ such that $(\mathtt{i}_2 - 1, \mathtt{j}_2) = (\mathtt{i}_3, \mathtt{j}_3 - 1) = (\mathtt{i}_1, \mathtt{j}_1)$. The dates $d_2 = \mathtt{i}_2 \times M_n + \mathtt{j}_2$ and $d_3 = \mathtt{i}_3 \times M_n + \mathtt{j}_3$. In order to analyse the relations between the writes and the reads to the same element can appear, we introduce the variables dd_2 and dd_3 defined by equations $dd_2 = d_2 - d_1$ and $dd_3 = d_3 - d_1$.

The whole system is

$$
\begin{array}{rcl|ccccc}
\mathtt{i}_1 & = & \mathtt{i}_2 - 1 & & & & & \\
\mathtt{j}_1 & = & \mathtt{j}_2 & 1 & \leq & \mathtt{i}_1 & \leq & \mathtt{n} \\
\mathtt{i}_1 & = & \mathtt{i}_3 & 1 & \leq & \mathtt{i}_2 & \leq & \mathtt{n} \\
\mathtt{j}_1 & = & \mathtt{j}_3 - 1 & 1 & \leq & \mathtt{i}_3 & \leq & \mathtt{n} \\
d_1 & = & \mathtt{i}_1 \times M_n + \mathtt{j}_1 & 1 & \leq & \mathtt{j}_1 & \leq & \mathtt{n} \\
d_2 & = & \mathtt{i}_2 \times M_n + \mathtt{j}_2 & 1 & \leq & \mathtt{j}_2 & \leq & \mathtt{n} \\
d_3 & = & \mathtt{i}_3 \times M_n + \mathtt{j}_3 & 1 & \leq & \mathtt{j}_3 & \leq & \mathtt{n} \\
dd_2 & = & d_2 - d_1 & & & \mathtt{n} & \leq & M_n \\
dd_3 & = & d_3 - d_1 & & & & &
\end{array}
$$

Once this system is reduced, the variable dd_2 and dd_3 are found to have only positive values. This result can be interpreted as following: according to the date definition d, the array elements are always written before being read. Because this date corresponds to the semantics of the `do` loops, these relations have to be respected in any valid loop reordering.

Before trying any program transformation, a loop reordering can be validated from the new dates associated to the events. Consider the new date $t = \mathtt{i} + \mathtt{j}$ associated to iteration $(\mathtt{i}, \mathtt{j})$. We introduce equations $t_1 = \mathtt{i}_1 + \mathtt{j}_1$, $t_2 = \mathtt{i}_2 + \mathtt{j}_2$, $t_3 = \mathtt{i}_3 + \mathtt{j}_3$ to represent the new dates of the writes and reads to the same element. Equations $dt_2 = t_2 - t_1$ and $dt_3 = t_3 - t_1$ express the relations between these dates. Once the new system is reduced, dt_2 and dt_3 are found to have only positive values. The orders of the write and reads accesses to the same element of the array is the same in the new date system as in the original one: the new dates define a possible reordering of the loop execution.

This conclusion leads to code transformations.

4 The system transformations

4.1 The standard system representation

The basic system transformation is the combination of two equations in order to eliminate a common variable from one of them. Our experiments show that the most interesting equations are those which depend on the minimum of variables. In an $n \times m$ system (n equations, m variables), $n \leq m$, the variables are split into two sets: a *diagonal set* of n variables and a *column set* of $m - n$ variables. By a sequence of linear combinations of the equations the system is transformed in such a way that each variable from the diagonal set appears in only one equation. Each time two synonyms are discovered — two variables x and y are synonymous if an equation of the form $\alpha x + \beta y = C$ can be produced — one of the variables is removed as well as the equation. The bounds of the removed variables are reported on the one that is kept. After this transformation, two external variables can be associated with the same internal one. The standard representation of the system is shown in figure 1.

Such a representation is obtained by a classical elimination process.

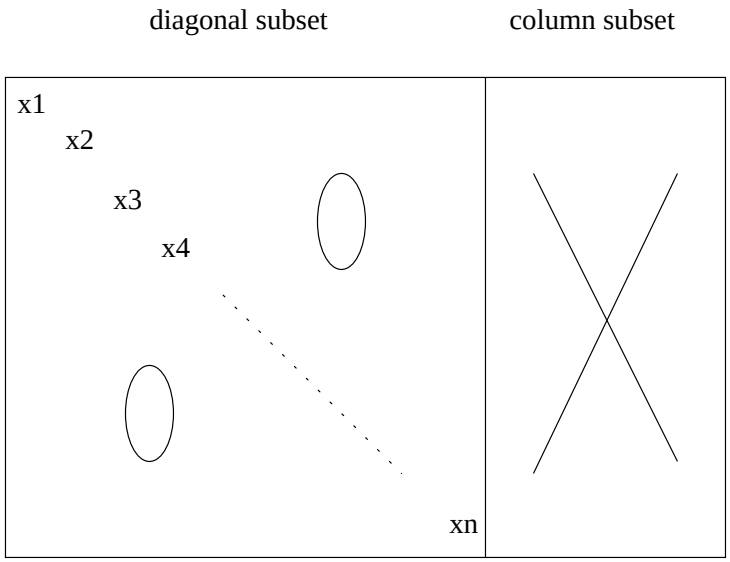

Figure 1: standard representation

4.2 GCD reduction

Each time a new equation is introduced in the system, or is created by linear combination, this equation is gcd-reduced : the gcd of all the factors is computed. All the variable factors and the constant factor are divided by the gcd. The gcd-reduction can fail if the constant factor of the equation cannot be divided by the gcd, in which case, the system has no solution : empty domain results are reported to the compiler. Consider the loop

```
do i= low, high
   a(2*i+n) = b(i)
   a(2*i+n+1) = c(i)
end do
```

The equation $2\mathtt{i}_1 + \mathtt{n} = 2\mathtt{i}_2 + \mathtt{n} + 1$ cannot be gcd-reduced. This system has no solution.

4.3 Domain stride

A variable x has a domain stride S if all its possible values can be written $x = S \times i + C$, with S and C constant. Each time a new equation is created, each variable of the equation is considered. The gcd of all the other non null factors of the equation is computed. This gcd becomes the domain stride of the variable. Consider equation $\alpha_1 x_1 + \alpha_2 x_2 + .. + \alpha_n x_n = C$

and variable x_i. The domain stride s_i of x_i is the gcd of $\alpha_j, j \neq i$. If s_i is different from one, the equation can be written $\alpha_i x_i + s_i \times (\Sigma_{j \neq i} \alpha_j x_j) = C$. Note that s_i and α_i are prime relatively after the gcd-reduction. s_i must divide $\alpha_i x_i - C$. The solutions for x_i can be written $x_i = s_i x'_i + r_i$. After this substitution the system is $\alpha_i s_i x'_i + \alpha_i r_i + s_i \times (\Sigma_{j \neq i} \frac{\alpha_j}{s_i} x_j) = C$ and after factorization under s_i, we get $\alpha_i r_i + s_i V = C$ with $V = \alpha_i x'_i + (\Sigma_{j \neq i} \alpha_j x_j)$.

From Bezout and with the Extended Euclidian algorithm, we can compute a value r for the unknown r_i. The variable x_i is replaced by $s_i x'_i + r$ in all the equations. After this substitution, the original equation is gcd-reduced and becomes $\alpha_i x'_i + (\Sigma_{j \neq i} \frac{\alpha_j}{s_i} x_j) = \frac{C - \alpha_i r}{s_i}$.

Internal variable x_i has been removed from the system, and the corresponding external variable is associated with x'_i. The solution domain of x'_i is computed from the known solution domain for x_i.

Consider the example

$$\begin{array}{rl} \text{equation:} & 11x_1 - 3x_2 + 6x_3 = 1 \\ \text{domains:} & x_1 : [1, ..], x_2 : [1, ..], x_3 : [1, ..] \\ \text{interfaces:} & s_i = 1, c_i = 0, 0 \leq i \leq 3 \end{array}$$

After transformation, the system becomes

$$\begin{array}{rl} \text{equation:} & 11x'_1 - x_2 + 2x_3 = -7 \\ \text{domains:} & x'_1 : [0, ..], x_2 : [1, ..], x_3 : [1, ..]. \\ \text{interfaces:} & s_1 = 3, s_2 = 1, s_3 = 1, c_1 = 2, c_2 = 0, c_3 = 0. \end{array}$$

4.4 Symbolic stride

A variable x possesses a symbolic stride S if it can be written $x = S \times x' + V$ where S is a constant and V is variable but invariant in the program fragment. The definition of a symbolic stride depends on the semantics of the variables of the code fragment : the symbols — invariants — must be distinguished from the true variables. The request for symbolic stride computation defines a subset of the external variables as the set of symbols. The computation of the symbolic stride is similar to the computation of the domain stride, but the factors of the symbols are ignored in the gcd computation.

The value V cannot be computed at compile time, but is invariant in the fragment. The knowledge of the symbolic strides has no direct effects on the system. In many cases, however, it can allow other factorizations in the system. When a symbolic stride is discovered on a variable x, two new variables x' and V are created, the equation $x = S \times x' + V$ is added with the constraint $0 \leq C < S$.

4.5 Subexpression factorization

Subexpression factorization consists of looking for two relatively non prime factors in an equation. If such a factor f is found on two variables x_i and x_j, the equation can be written $f(\frac{\alpha_i x_i}{f} + \frac{\alpha_j x_j}{f}) + \Sigma_{k \neq i, k \neq j} \alpha_k x_k = C$. An artificial variable v can be created; an equation $v = \frac{\alpha_i x_i}{f} + \frac{\alpha_j x_j}{f}$ is introduced and combined with the other equations. Subexpression factorization is interesting because it introduces triples (equations of three variables) into the system.

4.6 Domain reduction

The domain reduction phase tries to reduce the upper bound and to increase the lower bound of each variable. Three algorithms of different complexity are used, the minmax algorithm, the minmax refinement, and the exact minmax computation on triples.

4.6.1 Minmax algorithm

The minmax algorithm scans the variables of an equation and computes lower and upper bounds for each variable from the lower and upper bounds of the other variables in the equation. Consider equation $\alpha x = \Sigma_i \alpha_i x_i + c$. An upper and lower bound for x can be extracted as follows. $ub(X)$ and $lb(X)$ denote the known (or computed) upper and lower bounds of expression X.

$$\begin{array}{rcl}
ub(\alpha x) & \leq & ub(\Sigma_i \alpha_i x_i + c) \\
 & \leq & \Sigma_i ub(\alpha_i x_i) + c \\
lb(\alpha x) & \geq & lb(\Sigma_i \alpha_i x_i + c) \\
 & \geq & \Sigma_i lb(\alpha_i x_i) + c \\
ub(\alpha_i x_i) & = & \alpha_i ub(x_i) \text{ if } \alpha_i > 0 \\
 & = & \alpha_i lb(x_i) \text{ if } \alpha_i < 0 \\
lb(\alpha_i x_i) & = & \alpha_i ub(x_i) \text{ if } \alpha_i < 0 \\
 & = & \alpha_i lb(x_i) \text{ if } \alpha_i > 0 \\
ub(x) & \leq & \lfloor \frac{ub(\alpha x)}{\alpha} \rfloor \text{ if } \alpha > 0 \\
 & \leq & \lfloor \frac{lb(\alpha x)}{-\alpha} \rfloor \text{ if } \alpha < 0 \\
lb(x) & \leq & \lceil \frac{lb(\alpha x)}{\alpha} \rceil \text{ if } \alpha > 0 \\
 & \leq & \lceil \frac{ub(\alpha x)}{-\alpha} \rceil \text{ if } \alpha < 0
\end{array}$$

The new domain of variable x is the intersection of its previous domain and the interval $[lb(x), ub(x)]$.

An iteration of this algorithm on the system is relatively fast, but the quality of the results depend heavily on the current form of the system. One could reiterate until no bound is modified but the convergence rate can be very poor. After many experiments, we have found that the convergence rate is greatly improved if the system is transformed in such a way that the variable whose bounds we are computing and the "most sensitive" variables are in the column part. By "most sensitive", we mean variables whose bounds have been the most recently reduced, and also variables with narrow domains. Intuitively, such a representation produces a maximum number of equations that contain a maximum of "constrained" variables.

Let us consider the following equations and constraints from [Pugh91].

$$\begin{array}{rcl}
a & = & 11x + 13y, \qquad a \geq 3, \quad a \leq 21 \\
b & = & 7x - 9y \quad , \qquad b \geq -8, \quad b \leq 6
\end{array}$$

In a first step, b and a are the most recently constrained variables. From the combined equation $9a + 13b - 190x = 0$, the interval of x is reduced to $[0, 1]$. Now x becomes the most constrained variable, before b. The interval of y is reduced to $[0, 1]$ from equation $b = 7 * x - 9 * y$. After this step, the interval of values of x, y, a or b cannot be reduced any more.

In general, two iterations are run on the whole set of equations. In the first run, the most recently constrained variables are placed in the column part. For the second run, the most constrained variables are selected.

4.6.2 Minmax refinement

The bounds obtained from the minmax algorithm are not necessarily solutions for each equation. If we consider the previous example, setting $x = 0$, we get $a = 13y$ and $b = -9y$: the system has no solution for the known lower bound of x. The minmax refinement can be run after the minmax algorithm. Let us consider the equation $X = \Sigma_{i=1}^{n} \alpha_i x_i$ with the constraints $\alpha_i > 0, x_i \geq 0, \forall i$. In this equation, the lower bound of X is zero. This corresponds to the minmax algorithm. If the constraint $X > 0$ is added, then a lower bound of X is $min_i \alpha_i$.

In the previous example, from equation $a = 11x + 13y$ and the new constraints $x \geq 0, y \geq 0$ we deduce $a \geq 11$. In order to refine a bound the equation must be rewritten in such a way that the factors are strictly positive, and the variables positive or null. If we replace a by $up(a) - a'$, x by $up(x) - x'$ and y by $up(y) - y'$, we obtain the equation $a' - 21 + 11 + 13 = 11x' + 13y'$. Then from $a' + 3 \geq 11$ we obtain a new upper bound $a \leq 13$ for a.

The same refinement can be applied on equation $b = 7x - 9y$ and produces $-2 \leq b \leq 0$. When the minmax algorithm is rerun with these new bounds, this problem is found to have no solution. In fact, increasing the lower bound of a from 3 to 11 is sufficient to introduce a contradiction in the system.

When the ratio of the lowest factor and the other factors is important, the minmax refinement is not limited to the value of this lowest factor: all the multiples lower than the other factors can be considered.

4.6.3 Exact bounds on triples

In many cases, the minmax refinement gives good results, mainly when large factors appear in the equation. However, it does not guarantee that the computed bounds are solutions. We have developed an exact bounding algorithm on triples (equations with three unknowns). When applied to a variable of a triple, this algorithm first checks if the lower bound of this variable is a solution of the equation.

Let us consider equation $\alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 = c$. If we consider variable x_1, the equation is rewritten $\alpha_1 x_1 - c' = \alpha_2 x'_2 + \alpha_3 x'_3$ with $\alpha_i > 0$ and the constraints $x'_2 \geq 0, x'_3 \geq 0$. If minmax refinement has been run on the equation, the known constraint on x_1 is such that $\alpha_1 x_1 - c' \geq V$. If $V = 0$, $x_1 = V + c'$ is a solution and we have $x'_2 = x'_3 = 0$. If $V > 0$, let us

consider equation $V = \alpha_2 x'_2 + \alpha_3 x'_3$. From a solution of $1 = \alpha_2 x'_2 + \alpha_3 x'_3$, we can compute two solutions x_2^v and x_3^v which verify the equation. These solutions are not unique and we choose $0 \leq x_2^v < \alpha_3$. If the corresponding value x_3^v is positive, then we have got a solution for the equation, and no further refinement is needed.

If $x_3^v < 0$, V is not a solution. The values $V + \alpha_1, V + 2\alpha_1, ...$ are checked until a solution is found. Fast enumeration algorithms can be applied, depending on the ratio of α_2 and α_3.

4.6.4 The complexity of the minmax computations

The exact bound computation can be applied on triples only, and at least two factors of the equation must be different from unity. This case is not very frequent in real programs. Moreover, the already known bounds are frequently solutions. Only the first step of the algorithm is run in this case.

The basic minmax algorithm and the minmax refinement algorithm have the same complexity. These two algorithms can be mixed. In general, very few iterations are necessary to obtain convergence.

4.7 Constraints extraction

The result of a dependency computation is not limited to the value domain of the variables. After the resolution phase, the system contains new constraints on the artificial variables. These constraints cannot be exploited directly by the compiler because the exact semantics of the spare variables are not known outside the system. The constraints extraction phase allows the compiler to store these constraints in a more exploitable form. A constraint extraction request on two variables x_1 and x_2 produces a list of expressions on these two variables and the domain of possible value of this expression (the bounds and a stride). This request first transforms the system of equations in order to place the requested variables in the column part. The equations which contain these two variables are extracted. The result is a list of expressions of the form $\alpha x + \beta y \in domain$. The domain is an interval and a stride. The interpretation of these expressions is left to the compiler, and depends mainly on the semantics of the variables.

For instance, consider the loop

```
do i=1, n
   a(i)= a(n-i)
end do
```

Because the loop bound `n` is symbolic, we cannot extract as precise information as in the case where the value of `n` is known at compile time. A typical reaction of the compiler in this case is to ask for the relations between the loop variable and the loop limits.

After transformation, the system is

$$\begin{array}{rcl} \mathtt{i}_2 - \mathtt{n} + \mathtt{i}_1 & = & 0 \\ s_1 + \mathtt{n} - 2\mathtt{i}_1 & = & 0 \end{array}$$

and the domain of s_1 is $s \in [..., -1]$.

The answer of the system to a constraint request on $\mathtt{i}_1$ and $\mathtt{n}$ is computed from these two equations.

$$\begin{array}{rcccll} 2 & \leq & -\mathtt{i}_1 + n & \leq & 2147352577 & \text{stride } 1 \\ -2147352576 & \leq & 2 \times \mathtt{i}_1 - n & \leq & -1 & \text{stride } 1 \end{array}$$

The first one is already known to the compiler. The second one has the form $\alpha\mathtt{i} \leq \mathtt{n}$: it splits the iteration domain in two parts : a first part from the low bound to the point $\mathtt{n}/\alpha$ and a second part from $\mathtt{n}/\alpha + 1$ to $\mathtt{n}$. The loop can be written

```
do i=1, n/2
   a(i)= a(n-i)
end do
do i=n/2+1, n
   a(i)= a(n-i)
end do
```

Now, each of these two loops can be run in parallel.

4.8 Some complexity consideration

Integer programming techniques are usually considered to be too expensive to be used in compilers. Our system falls into this category, and its cost is an important factor for its introduction in an automatic tool. We have measured its response time on many problems. We have found that its introduction is realistic. The most complex steps are rarely needed. For instance, before running the stride computations, the number of factors equal to unity is counted in an equation. If two or more unit factors exist, the stride computation is not run. If only one exists, the stride is computed only for this variable.

In the case of the minmax computation, we limited the number of iterations to four or five. When the algorithm does not converge within these limits, it can be reported to the compiler which might take some action.

5 Future work

The system has been run extensively, and should be introduced in an experimental compiler in the near future. However, we are still studying some parts of the algorithm. For instance, the major cost of the minmax computation lies in the system transformations that are required. These transformations can be avoided if all the equations that have been produced are kept in a table and the minmax computations are run on this list. The storage cost of such a representation must be compared to the economy in the transformations. The experiments show that triples are very frequent in the system. An intermediate decision could be to store only the triples. In this case, a more specific and economic representation could be adopted. Because the system is incremental, there is some redundancy in the transformations. It

seems that the use of such a table can avoid redundant computations: when a new equation is created, the transformation is stopped if the equation is already in the table. We have measured that the most interesting equations are triples.

The system is built on a bignum package because all the intermediate results in the computation cannot be represented in machine arithmetic. However, when the representation of program variables is limited to n bits, all the computations of our system can be run with $3n$ bits without overflow. In fact, a limitation to $2n$ bits should rarely produce overflows and can be accepted. The next generation of the system will be run with 64 bit arithmetic.

6 Conclusion

Integer programming techniques are usually considered to be far too expensive to be introduced in an automatic restructuring system. We have reported the results of our experiments and we have shown that such a technique can be efficiently used to extract information from programs. The information that can be computed is not limited to the extraction of distance or direction vectors. It extends to the characterization of the value domain of program variables where some assertion is verified. Such techniques can also be used to compute new relations between the variables. Many transformations such as loop interchanging, loop skewing, loop distribution, loop splitting as well as data partitioning can be decided from the result of integer programming technique applications.

References

[Pugh91] W. Pugh The Omega Test : a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing'91*, November 1991.

[Ban88] U. Banerjee, *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA. 1988.

[LiTh88] A. Lichnewsky, F. Thomasset, Introducing symbolic problem solving techniques in the dependence testing phase of a vectorizer. In *Proceedings of 1988 International Conference on Supercomputing*, Saint-Malo, France, July 1988.

[Wolfe89] M.J. Wolfe *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[BanWol87] M.J. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137-178, April 1987.

[HagPol91] M. Haghighat, C. Polychronopoulos, Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. *Advances in Languages and Compilers for Parallel Processing*, MIT Press, 1991.

[Wallace88] D.R. Wallace. Dependence of multidimentional array references. In *Proceedings of 1988 International Conference on Supercomputing*, Saint-Malo, France, July 1988.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399