AUTOMATIC PARALLELIZATION OF FORTRAN PROGRAMS IN THE PRESENCE OF PROCEDURE CALLS

Rémi TRIOLET [1]

Paul FEAUTRIER [2] *François IRIGOIN* [3]

ABSTRACT

Scientific programs must be transformed and adapted to supercomputers to be executed efficiently. Automatic parallelization has been a very active research field for the past few years and effective *restructuring compilers* have been developed.

Although CALL statements are not properly handled by current automatic tools, it has recently been shown that parallel execution of subroutines provides good speedups on multiprocessor machines.

In this paper, a method to parallelize programs with CALL statements is described. It is based on the computation of subroutine effects. The basic principles of the method are to keep the general structure of the program, to compute the effects of called procedures on those calling and to find out and use predicates on scalar variables to improve array handling.

First it is shown that some knowledge of the CALL statement context is necessary to compute accurately a property over a procedure. This leads us to define the notions of *static procedure occurrence* and *static occurrence tree.*

Then a new concept, called *region,* is introduced to define precisely the effect of a procedure execution. These regions allow us to describe, in a calling procedure, the parts of arrays which are read or written by the called procedure executions. The main lines of the effect computation algorithm are given. It is based on a bottom-up analysis of the static occurrence tree.

The computation and the use of predicates among scalar variables to improve array handling are also, as well as regions, new in restructuring compilers. Classical semantics analysis methods are adapted to our special needs and extended to the interprocedural case. It is also briefly explained how the predicates these methods provide can be used.

Finally, the introduction of our method in a restructuring compiler is reported.

[1] University of Illinois, Center for Supercomputer R & D.
[2] Université PARIS VI, MASI
[3] Ecole des Mines de Paris, Centre d'Automatique et Informatique.

Introduction

To be efficient, scientific programs must undergo various transformations before being run on modern supercomputers. Manually implementing these transformations is tedious and error prone work. Hence, much research work [All1, Bro, Cot, Kuc] aims at automating these transforms. In order to detect the parallelism inherent in the source program, modern restructuring compilers build a *dependence graph*. This graph highlights the conflicts between statements due to variable accesses [All2, Ban]. It is computed from the knowledge of read and written variable sets for each statement; in the case of procedure calls, no satisfactory method for the determination of these sets is known.

A pessimistic attitude is always possible: take for these sets the union of all arguments and global variables. In many cases, this induces a large loss in possible parallelism.

Another solution is to expand procedure calls in line [Hus]. This solution is very efficient in the case of small low-level procedures, but generates problems when applied to all calls. Firstly, the initial program structure is lost, with a corresponding loss in intelligibility†. Secondly, false data dependences are created, especially by local dynamic arrays‡. Thirdly, the program size is considerably increased. Lastly, in a language like FORTRAN, there are many details and anomalies which must be attended to, thus greatly increasing the complexity of the whole process [Tri2].

In this paper we propose to tackle directly the problem of computing the conflicts between the program's components. In the first part, we justify the global design of our method: to keep the procedural structure of the program, to treat each call separately, and to use predicates on the values of scalar variables to obtain more precise information on array accesses.

In the second part, we describe in details all the data we gather on the source program. We introduce the notion of a procedure *static occurrence*. We then show how the data we are looking for may be obtained as limit of an iterative process; this limit is guaranteed to be found in a finite number of steps.

In the third part, we show how to compute each type of data. In particular, we give a precise notation to describe the effect of a procedure call, and a method to compute effects that takes into account the call context. We adapt classical semantics analysis methods to the interprocedural case. This provides predicates between program variables that are used to sharpen the analysis of array accesses.

Finally, we show in the fourth part how to use these ideas in a restructuring compiler; in particular, we show in an example how to test dependences between statements due to region overlapping.

---

† This is significant for source to source restructuring compilers only.

‡ This phenomenon is obviated for scalar variables by the technique of scalar expansion; however, there is today no corresponding algorithm for arrays.

1. General Description

1.1. Principles

Suppose we are able to calculate exactly the effect of any procedure call on all program variables. Furthermore, suppose that the description of this effect is such that we are able to compute exactly the dependence graph of the program. We may then parallelize the source program as if the procedure calls were blocks of assignment statements. In favorable cases, the result of this process is a program where several procedures (or, in the case of a parallel loop, several instances of the same procedure) may be executed in parallel.

In [Don], Dongarra reported on the manual application of this technique to the well-known numerical libraries, LINPACK and BLAS. He obtained speedups between 7 and 9 on a Denelcor-HEP, where the maximum possible value is 10; this clearly shows the interest of automating this technique.

1.2. Advantages

The parallel version has the same structure as the original program, and is still readable by its author. This technique does not increase the size of the program modules. Scalars and arrays local to a procedure may be safely ignored when computing its effect.

The resulting program contains parallel procedure calls. This is very convenient for modern MIMD machines: Denelcor-HEP, Cray-XMP, Cray-2, NYU-Ultracomputer [Got], RP3 [Pfi], CEDAR [Gaj], etc... The size of parallel tasks tends to be large; the relative impact of the activation overhead is reduced. Furthermore, for many operating systems (Cray-2, Cray-XMP, HEP...) the multitasking facility is accessible only through *asynchronous CALLs*. In this case, our technique is directly useful.

1.3. Outlines of the Proposed Solution

1.3.1. Describing a Procedure Call Effect

We define a new concept, called *region,* to describe the sub-arrays often used in scientific programs: rows, columns, diagonals, sub-vectors, ... Statements using a procedure, whether as a CALL or through function evaluations, as well as ordinary statements are associated to a unique node in the control graph. We compute the read and modified regions† for each node.

This is a multi-stage process: accessed regions are computed in the notations of the called procedure and are translated in the calling program notations, according to associations between formal and actual parameters.

1.3.2. Predicates

Regions associated to a statement must include the union of accesses done by all iterations of this statement. When an array is accessed, unknown subscript values must be ignored leading to an inaccurate region. Therefore we compute and use predicates on integer scalar variables to constraint subscript values.

† In the following the word *accessed* will be used instead of *read and/or modified*

A study has shown that our needs are best served by predicates in the form of linear equalities and inequalities among integer scalar variables [Tri1]. Such predicates will be called *linear predicates.*

1.3.3. Aliasing

To understand a procedure, we need to know all associations between variables†. Aliasing is a special form of association, where at least one of the variables is a formal parameter [Hec]. It is possible to interpret the FORTRAN-77 standard as restricting aliasing to the read only case, which is harmless. However, as this standard is not enforced by usual compilers, we propose to take aliasing into account. Nevertheless, aliasing is a minor problem; the reader is referred to [Tri1] for the details of our technique.

2. An Iterative Algorithm To Compute Parallelization Data

2.1. Static Occurrence of a Procedure.

To compute the effect of a procedure, we need properties of its initial state. They are computed from predicates valid just before the CALL to the procedure. If this was done only once for all CALLs to a procedure, it would be necessary to use a predicate no stronger than the *or-ing* of all these predicates, with a corresponding loss in precision.

To obviate this problem, we introduce the notion of *static occurrence of a procedure.* To each procedure are associated as many static occurrences as there are distinct sequences of CALLs from the main program to the procedure. This may be modeled as a *static occurrence tree,* which is finite since recursion is forbidden in FORTRAN 77. Figure 1. shows a sample program, its call graph and its static occurrence tree. For instance, there are three occurrences of the procedure R: R1, R2, R3.

2.2. Description of Parallelization Data

Let $\Omega = \{\omega_1, \omega_2, ...\}$ be the set of static procedure occurrences in a program. If ω is an element of Ω, $P(\omega)$ is the procedure which ω is an occurrence of. The aim of our analysis is the construction of the following global functions:

$A(\omega) = fa_\omega$

where fa_ω is a function telling which pairs of variables are alias for ω.

$I(\omega) = fi_\omega$

where fi_ω is a function associating to a node of $P(\omega)$ a set of linear predicates which are true before every execution of the corresponding statement for ω.

$M(\omega) = fm_\omega$

where fm_ω is a function giving the sub-arrays accessed by procedure calls in ω.

In the sequel, fa_ω, fi_ω and fm_ω will be called the aliasing function, predicate function and manipulation function, respectively.

† This is also true for the dependence computation; however associations are often ignored in this case.

```
PROGRAM M        SUBROUTINE P     SUBROUTINE Q     SUBROUTINE R
...              ...              ...              ...
CALL P  (a)      CALL Q  (d)      CALL R  (e)      END
...              ...              ...
CALL R  (b)      END              END
...
CALL P  (c)
...
END
```

Figure 1-(a). Fortran Code

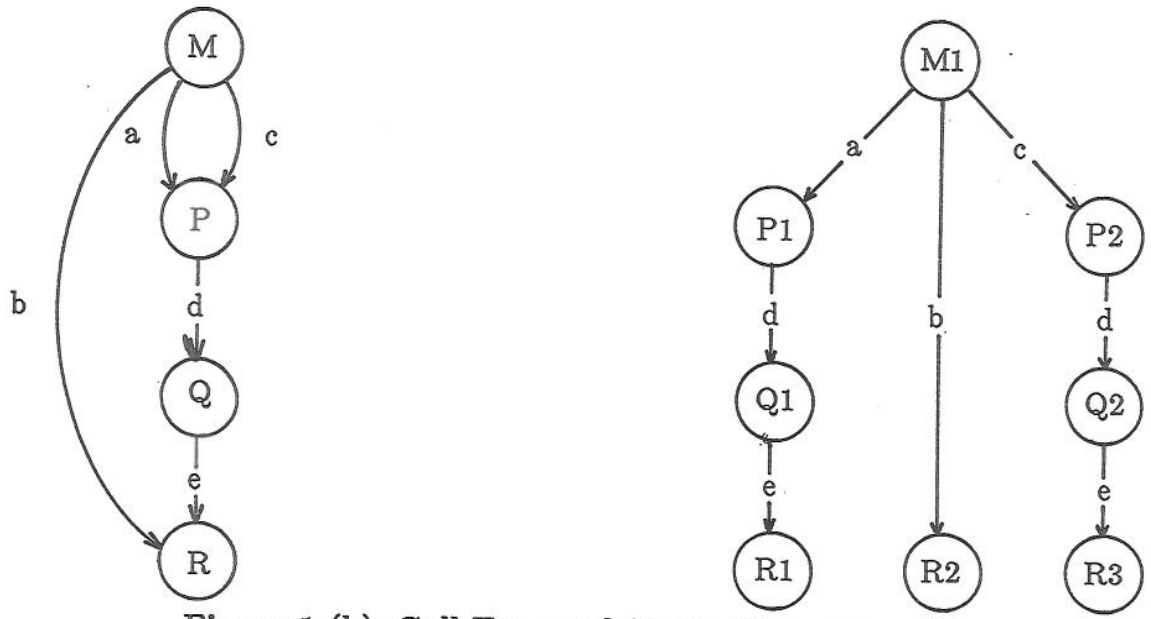

Figure 1-(b). Call Tree and Static Occurrence Tree

2.3. Computing Parallelization Data

A direct computation of A , I and M is not optimal: each of them may† use data supplied by the others. For instance, the computation of M benefits from the knowledge of predicates and aliasing, i.e. of I and A. The following iterative algorithm solves this problem:

```
i := 0;
while NOT-TERMINATED do
    compute A_i as a function of I_i ;
    compute M_i as a function of I_i and A_i ;
    compute I_i+1 as a function of M_i and A_i
    i := i+1;
done
```

† we use "may" because *pessimistic* decisions are always possible in the absence of further information.

I_o is the initial predicate function, which associates the empty set to every statement in every static occurrence.

This algorithm terminates provided functions A_i, I_i and M_i are computed with algorithms detailed in [Tri1] and sketched in § 3. Convergence is detected when two successive values of I are equal: NOT-TERMINATED is simply $I_i \neq I_{i-1}$.

3. Computing the Parallelization Functions

3.1. Regions

As previously discussed, we need a technique to describe sub-arrays usually manipulated in scientific programs like:

(a,b) array elements: $T(3,4)$, $U(I,J-3)$

(c) a regularly spaced subset: $V(I,1)$, $V(I,3)$, ..., $V(I,2*N+1)$

(d) a rectangular sub-matrix: $W(3{:}5,I+1{:}I+4)$

(e) a triangular sub-matrix of X

(f) a diagonal of a matrix Y

and so on.

The regions allow a precise dependence computation, providing we can decide whether two regions have common elements (see § 4).

A region is a pair $r=(T,\Sigma)$ where T is a variable name and Σ is a set of linear predicates on the values of T's subscripts, which appear in Σ as the special variables (ϕ_j), $j=1..7$. The (ϕ_j) act as bound variables which may be systematically renamed, for instance when computing the intersection of two regions.

Figure 2 lists the regions which correspond to the sub-arrays of examples (a)-(f) above. All these regions are exact descriptions of these sub-arrays. A scalar S is represented by the region $(S,\emptyset)$.

3.2. The Manipulation Function M

The algorithm is based on a bottom-up traversal of the static occurrence tree. Thus, the analysis of a procedure call c_p included in a node n_p of a static occurrence ω_p is attempted after the processing of the static occurrence ω_q associated to c_p; hence regions accessed by all ω_q procedure calls are known. This bottom-up analysis begins at the leaves of the static occurrence tree, in which there are no calls. The analysis of a call like c_p is split into six different phases, M1 to M6, whose overall organization is depicted in figure 3.

3.2.1. Dead Code Elimination (M1)

We attempt to detect dead code in ω_q by evaluating boolean expressions with the help of linear predicates provided by $I(\omega_q)$ (see § 3.5). An algorithm is proposed in [Tri1]; the result is the set $AN(\omega_q)$ of *Accessible Nodes*.

$(T, \{\phi_1 = 3,\ \phi_2 = 4\})$ (a)

$(U, \{\phi_1 = I,\ \phi_2 - J = -3\})$ (b)

$(V, \{\phi_1 = I,\ 1 \leq \phi_2 \leq 2N+1,\ \phi_2 - 2k = 1\})$ (c)

$(W, \{3 \leq \phi_1 \leq 5,\ I+1 \leq \phi_2 \leq I+4\})$ (d)

$(X, \{\phi_2 - \phi_1 \leq -1\})$ (e)

$(Y, \{\phi_1 = \phi_2\})$ (f)

Figure 2: A Few Regions

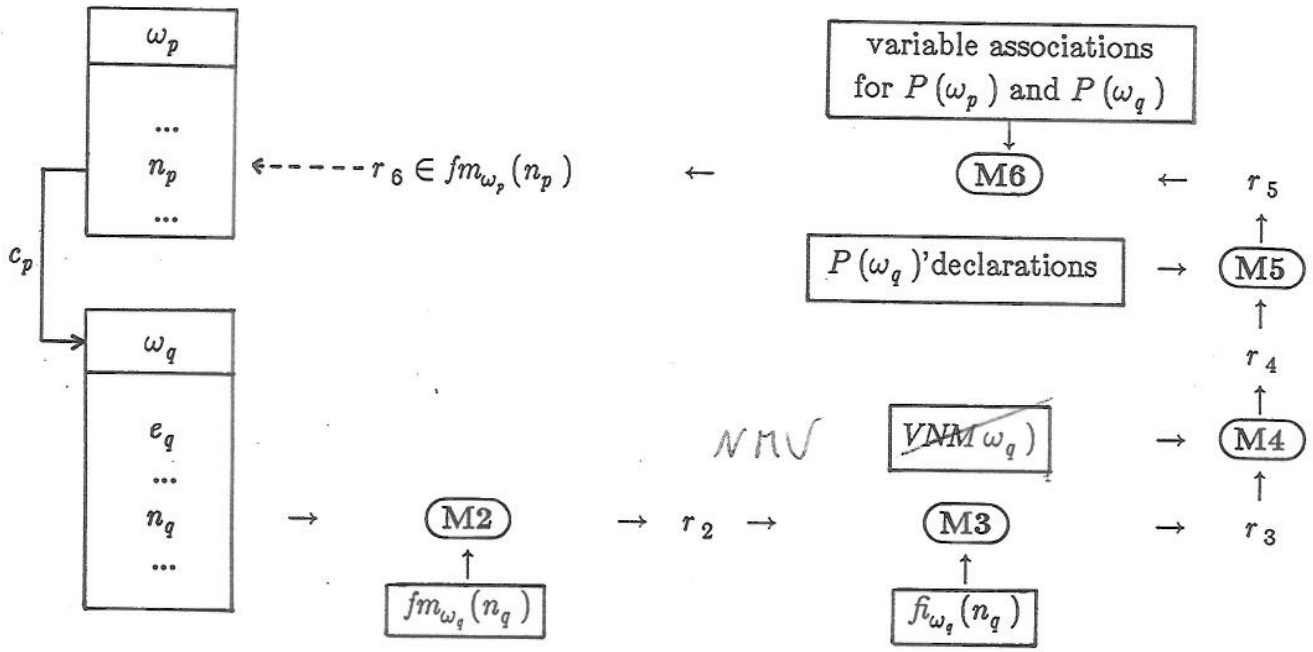

Figure 3: Computing the Manipulation Function

```
      DO 20 I = 1, N
          DO 10 J = I-K, I+K
              T(I,J) = ...
10        CONTINUE
20    CONTINUE
```

Figure 4. Two Nested DO-Loops

3.2.2. Region Determination (M2)

For a node in $AN(\omega_q)$, accessed regions are of two different origins:

- indirectly accessed regions come from procedure calls in the current node. They are given by the function $M(\omega_q) = fm_{\omega_q}$ which is available since the analysis is bottom-up.
- directly accessed regions are explicit in the statement associated to the node.

To translate a reference such as $T(x_1, x_2, \ldots, x_d)$ into a region, we construct the set X of subscript expressions x_i which are linear combinations of integer scalars. The corresponding region is (T,Σ) with:

$$\Sigma = \bigcup_{x_i \in X} \{\phi_i = x_i\}.$$

For instance:

$$T(I,J) \overset{M2}{\rightarrow} r_2 = (T,\{\phi_1 = I,\ \phi_2 = J\})$$

$$T(I+2*J+1,I*K) \overset{M2}{\rightarrow} r_2' = (T,\{\phi_1 - I - 2J = 1\})$$

The value of the second index is lost, since I*K is not linear.

3.2.3. Using the Execution Context (M3)

The description of a region may be sharpened by including in Σ all predicates which are valid at the current node. These predicates are given by $I(\omega_q)$. As an example, the assignment to T(I,J) inside the double DO-loop of figure 4 leads to the following region:

$$r_2 \overset{M3}{\rightarrow} r_3 = (T,\{\phi_1 = I, \phi_2 = J, 1 \leq I \leq N, I - K \leq J \leq I + K\}).$$

3.2.4. Region Widening (M4)

A region is a symbolic entity which may be mapped on real memory addresses when the values of all variables (i.e the memory state) are known.

Our aim is to express all regions found in M3 by reference to the memory state just before the execution of n_p. This is a two step process: going from the state at an arbitrary node n_q of $P(\omega_q)$ to the state at e_q, entry node of $P(\omega_q)$; and then to the state at n_p.

The second step is easy, as it depends only on the semantics of the CALL mechanism (see M6). The first one is much harder: we need relations between variable values at n_q and e_q. One solution is the technique of symbolic execution [Fea].

In the interest of simplicity, we propose to restrict our attention to the set $NMV(\omega_q)$ of variables which are not modified by nodes in $AN(\omega_q)$. The value of such a variable is equal at e_q and n_q. The construction of $NMV(\omega_q)$ is straightforward since modified regions at each node are known from M3.

To pass from a region (T,Σ) at n_q to the corresponding one at e_q is simply to eliminate from Σ all variables outside $NMV(\omega_q)$ while keeping all relations they induce on the variables of $NMV(\omega_q)$. This implies an information loss, hence the name widening.

If for instance $NMV(\omega_q) = \{N, K\}$ the region r_3 quoted in § 3.2.3. is widened to:

$$r_4 = (T, \{1 \leq \phi_1 \leq N, \phi_1 - K \leq \phi_2 \leq \phi_1 + K\})$$

which describes a band centered on the main diagonal of T.

3.2.5. Using Array Bounds (M5)

We assume that arrays are accessed within their bounds. Transformations based on this assumption may modify the semantics of incorrect programs. This is no problem since their semantics is not well defined: their results may vary from run to run or from machine to machine.

This assumption allows us to add to all regions from M4 another set of constraints between indices and corresponding array bounds. Suppose for instance that T is declared by:

```
DIMENSION T(N+1, -7:N*N)
```

We add to each region of T the set:

$$\{1 \leq \phi_1 \leq N+1, -7 \leq \phi_2\}$$

The upper bound constraint on ϕ_2 is lost because it is quadratic. This gives r_5:

$$r_4 \xrightarrow{M5} r_5 = (T, \{1 \leq \phi_1 \leq N, \phi_1 - K \leq \phi_2 \leq \phi_1 + K, 1 \leq \phi_1 \leq N+1, -7 \leq \phi_2\})$$

3.2.6. Region Translation (M6)

Regions from M5 are expressed in the notations of $P(\omega_q)$. Last phase M6 aims at translating them in the notations of the calling procedure $P(\omega_p)$. This depends on variable associations induced by the CALL and COMMON mechanisms.

Unusual cases of association, allowed by FORTRAN, are handled in a pessimistic way. For instance, if a global variable A of $P(\omega_p)$ is partially associated with a global variable T of $P(\omega_q)$, the translation of any region of T is $(A, \emptyset)$. A similar decision is made for partial associations between actual and formal parameters.

A much better job can be done if the association is *sensible:*

- both arrays have equal bounds;
- the formal array is associated with a sub-array of the actual parameter.

In this case the base name of the region is simply changed. Then, region predicates are translated: each $P(\omega_q)$ variable must be expressed in term of $P(\omega_p)$ variables. Associations implied by the CALL and COMMON mechanisms are studied to provide relations among integer scalars of both procedures. Then, each $P(\omega_q)$ variable is either replaced when a linear relation exists for it, or eliminated.

A few examples of translation are given in Figure 5. For (T1), M is replaced by $2I-J+7$ but L is eliminated since $I*J$ is not linear. This elimination produces $\phi_1 - 2I + J \leq 7$. In (T2), N, being associated with a vector element, must be eliminated; this produces $1 \leq \phi_1 \leq 50$. In example (T3) predicate $M < IND$ from $P(\omega_q)$, found by ω_q semantics analysis, is propagated upward to $P(\omega_p)$.

```
SUBROUTINE Q( L , M , N )
COMMON /G/ MAT(10,20) , VEC(50) , IND
  ...
END
```

```
PROGRAM P
COMMON /G/ MAT(10,20) , VEC(50) , IND
  ...
CALL Q( I*J , 2*I-J+7 , VEC(I) )
  ...
END
```

$$(MAT, \{\ \phi_1 = L\ ,\ \phi_2 = M\ ,\ L \leq M\ \}) \overset{M6}{\rightarrow} (MAT, \{\ \phi_2\text{–}2I+J = 7,\ \phi_1\text{–}2I+J \leq 7\ \}) \qquad \text{(T1)}$$

$$(VEC, \{\ \phi_1 = N,\ 1 \leq N \leq 50\ \}) \overset{M6}{\rightarrow} (VEC, \{\ 1 \leq \phi_1 \leq 50\ \}) \qquad \text{(T2)}$$

$$(VEC, \{\ \phi_1\text{–}IND = 1,\ M < IND\ \}) \overset{M6}{\rightarrow} (VEC, \{\ \phi_1\text{–}IND = 1,\ 2I\text{–}J\text{–}IND < -7\ \}) \qquad \text{(T3)}$$

Figure 5. Examples of Region Translation

3.3. Computing the Predicate Function I

Several methods have been proposed in the literature [Cou, Kar, Kil, Jou]. Killdall's method gives a technique to detect variables whose value does not depend on the execution path. A very general method for approximating fix-points, by Cousot, may be used to obtain various type of predicates: bounds on the value of variables, linear constraints, etc... Karr's method allows one to propagate linear equalities. Jouvelot uses non-standard semantics to collect predicates.

The local analysis of a procedure is done as follows. Initial predicates - attached to its entry node - are given and extra predicates are mainly provided by assignment and test nodes. The knowledge of modified variables allows one to delete predicates which are no longer valid after the execution of a node. This information is available since an approximation of M is known when computing I (see § 2.3). As a matter of fact, nodes which contain procedure calls only affect predicates by destroying some of them. Finally, predicates are propagated through the procedure by an iterative algorithm.

For the whole program, the algorithm to compute I is based on a top-down traversal of the static occurrence tree. One static occurrence of procedure ω_p is processed by the local analysis to compute predicates for all nodes in $P(\omega_p)$. Then, if ω_q is a static occurrence of a procedure called by a node n_p in ω_p, initial predicates for $P(\omega_q)$ are computed from those of n_p by a translation process similar to phase (M6) above.

This top-down analysis starts from the main program where DATA statements provide initial predicates.

Local analysis is a complex process, hence, a simple method based on DO-loop properties is proposed in [Tri1] to find predicates useful for program parallelization.

--

3.4. Computing the Aliasing Function A

The reader is referred to [Tri1] for a description of a technique for the evaluation of the aliasing function A . Predicates found by the above process are used to improve the results of this analysis.

3.5. Using Predicates

All algorithms described in this paper depend on techniques to manipulate linear predicates. Many such techniques are available in the literature and were summarized in [Tri1]. The most important ones are:

- methods to eliminate redundancy from linear inequality systems;
- methods to test the feasibility of linear inequality systems: Shostak [Sho] and Fourier-Motzkin [Duf];
- a method to discard a variable from linear inequality systems;
- a method to find the equations implied by a linear inequality system;
- a method to solve a system of linear equations on the integer line.

In a restructuring compiler, these techniques are of paramount importance and are used in many algorithms beside the analysis of procedure calls: symbolic evaluation of numerical and boolean expressions, dependence computation, etc...

4. Improvement of a Restructuring Compiler

The data provided by the three functions A , M and I allows an accurate dependence graph computation. Of course, dependences created by CALL statements are accurately tested thanks to M . This was our main goal. In addition, predicates provided by I may be used to suppress spurious dependences. For instance, knowing that $K<0$ allows the DO-loop in figure 6 to be vectorized. Finally, aliasing data may be used to validate transformations against hidden variable associations.

```
      DO 20 I = 1, N
         T(I+K) = T(I) + ...
20    CONTINUE
```

Figure 6. A Sequential DO-loop ?

Here is a dependence test example. Consider subroutine MM in figure 7. Suppose that the CALL statement (C) to SMXPY modifies only one column of A, described by the following region:

$$(A\ ,\{1\leq\phi_1\leq N_1,\phi_2=I\})$$

The output dependence test for iterations I and I' of statement (C) is equivalent to deciding the feasibility of the following linear system:

$$\Sigma \begin{cases} 1 \leq I \leq N3,\ 1 \leq I' \leq N3 & \textit{semantics of loop } (L) \\ 1 \leq \phi_1 \leq N1,\ \phi_2 = I & \textit{region accessed by iteration } I \\ 1 \leq \phi_1' \leq N1,\ \phi_2' = I' & \textit{region accessed by iteration } I' \\ \phi_1 = \phi_1',\ \phi_2 = \phi_2' & \textit{existence of a dependence} \\ I < I' & \textit{iteration } I \textit{ is executed before iteration } I' \end{cases}$$

It is easy to see that this system is unfeasible by considering I, I', ϕ_2 and ϕ_2'. This proves there is no output dependence. The reader can convince himself by testing a few similar systems that loop (L) is parallel.

```
      SUBROUTINE MM (A,B,C,N1,N2,N3)
      REAL A(N1,N3),B(N1,N2),C(N2,N3)
      DO 10 I = 1,N3                          (L)
         CALL SMXPY(N2,A(1,I),N1,C(1,I),B)    (C)
10    CONTINUE
      RETURN
      END
```

Figure 7. Subroutine MM

5. Conclusion

In this paper a method to extend restructuring techniques in the presence of procedure calls was presented. Computing interactions between several modules and predicates among variables allows a finer description of array accesses and an improved dependence test.

A simplified version of this method has been implemented in PARAFRASE [Kuc] at the Center for Supercomputing Research and Development of the University of Illinois by R. Triolet. Experiments were done on LINPACK. This package contains 235 DO-loops, 98 of them containing CALL statements to BLAS routines. 22 of the later were correctly found parallel by the modified version of PARAFRASE, whereas only 20 were hand parallelized by the authors...

As a final remark, let us note that the data we gather may have other uses beside parallelization/vectorization. For instance, detecting possible array bound violations, interprocedural type checking or suggesting optimizations is easy in the above framework.

References

All1. J.R. Allen and K. Kennedy, "PFC: a program to convert Fortran to a parallel form," in *Supercomputers, Design and Application*, ed. K. Hwang (1982). COMPSAC, Tutorial

All2. J. R. Allen, "Dependence Analysis for Subscripted Variables and its Application to Program Transformations," PhD Thesis, Dept. of Mathematical Science, Rice University (1983).

Ban. U. Banerjee, "Speedup of Ordinary Programs," Report No UIUCDCS-R-79-989, University of Illinois (1979). PhD Thesis

Bro. B. Q. Brode, "VAST: A Vectorization Tool for the Cyber-205," Internal Report, Pacific-Sierra Research Corp. (1982).

Cot. J. Cottet, C. Renvoise, and D. Sciamma, "Vesta: Vectorisation automatique et paramétrée de programmes," *Proc. of the 6th Int. Symp. on Programming* (1984).

Cou. P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," *Proc. of the 5th POPL Conf.* (1978).

Don. J. J. Dongarra and R. E. Hiromoto, "A Collection of Parallel Linear Equations Routine for the Denelcor HEP," *Parallel Computing* 1(2), pp.133-142, North-Holland (1984).

Duf. R. J. Duffin, "On Fourier's Analysis of Linear Inequality Systems," *Mathematical Programming Study 1*, North-Holland (1974).

Fea. P. Feautrier, "Projet VESTA: Outil de calcul symbolique," *Proc. of the 6th Int. Symp. on Programming* (Apr. 1984).

Gaj. D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "CEDAR: A Large Scale Multiprocessor," *Proc. of the 1983 Int. Conf. on Parallel Processing* (1983).

Got. A. Gottlieb, R. Grishman, C. Kruskal, K. MacAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer. Designing a MIMD, Shared-Memory Parallel Machine," *Proc. of the 9th Symp. on Computer Architecture* (1982).

Hec. M.S. Hecht, *Flow Analysis of Computer Programs*, North-Holland (1977).

Hus. C.A. Huson, "An In-Line Subroutine Expander for Parafrase," Report No UIUCDCS-R-82-1118, University of Illinois (1982). M.S. Thesis

Jou. P. Jouvelot, "Evaluation sémantique des conditions de Bernstein," Rapport Interne MASI #70, Université Paris VI (Feb. 1985).

Kar. M. Karr, "Affine Relationships Among Variables of a Program," *Acta Informatica*(6) (1976).

Kil. G. Killdal, "A Unified Approach to Global Program Optimization," *Proceedings of the 1st POPL Conference* (1973).

Kuc. D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proc. of the 4th Int. Conf. on Computer Software and Application* (Oct. 1980).

Pfi. G. F. Pfister and al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. of the 1985 Int. Conf. on Parallel Processing* (1985).

Sho. R. Shostak, "Deciding Linear Inequalities by Computing Loop Residues," *ACM Journal* 28(4), pp.769-779 (1981).

Tri1. R. Triolet, "Contribution à la parallélisation automatique de programmes," Thèse de Docteur-Ingénieur, Université Paris VI (Dec. 1984).

Tri2. R. Triolet, "Problèmes posés par l'expansion de procédure en Fortran 77," Rapport Technique, Centre d'Automatique et d'Informatique de l'Ecole des Mines de Paris (1985).

--