CENTRE DE RECHERCHE EN INFORMATIQUE

Interprocedural Analyses for Programming Environments

François Irigoin

Août 1992

Workshop on Environments and Tools For Parallel Scientific Computing, CNRS-SNF, Saint-Hilaire du Touvier, France, 7-8 Septembre 1992

Document EMP–CRI A-227

Interprocedural analyses for programming environments

F. Irigoin[a] [1]

[a]Centre de Recherche en Informatique, Ecole des Mines de Paris, 77300 Fontainebleau, France

Abstract

To investigate interprocedural parallelization, powerful interprocedural analyses were designed and implemented in PIPS, an experimental source-to-source parallelizer. Preconditions on integer scalar variables and accurate procedure effects called *regions* are computed to support the parallelization process. Although automatic interprocedural parallelization turned out to be much harder than expected a few years ago, interprocedural analyses can be exploited in many other ways in automatic tools and parallel programming environments. Motivating examples, selected from real applications, are shown and algorithms used to compute preconditions, array regions and array-kill information are presented.

1. INTRODUCTION

Multiprocessors are becoming more and more common place but they are still mainly used either as research tools or as sets of processors in a multiprogramming environment. This relative lack of success is due to many reasons. Lack of standard in the parallel programming language area, lack of support for debugging parallel programs, lack of motivation for many users who are interested in wallclock time and who do not use a dedicated multiprocessor for production work, to name but a few.

However these reasons would disappear if automatic parallelization were as successful as automatic vectorization because users could write their programs in their usual sequential languages and debug them in their usual environment. They could still design their programs with parallelism in mind but they would not have to express it nor to worry about its usefulness. Free improvements in wallclok time would then be welcome. This has been known for about 10 years and a set of research projects have been launched since then to reach this goal [1] [5] [7] [13].

In spite of this massive effort, no convincing success has yet been reported on any real application. Reasons for this setback have been investigated and are now known [3]. There is large and medium grain parallelism in applications but it is extremely hard to find and even harder to exploit. Vectorisation only requires analysis of small program sections. Parallelization requires analysis of large sections and any local failure leads to a global failure. Vectorization is supported and enhanced by a rich set of program transformation.

[1]This research was partially funded by DRET contract 87-017-01-018

Parallelization at the medium and large grain level is supported by no interprocedural transformation.

However some progress has been made during the last few years and more powerful techniques have been tested. Interprocedural transformations have been tried [11]. Array privatization algorithms have been published [8]. And interprocedural analysis techniques have been developed and tried on large programs [12].

PIPS, *Parallléliseur Interprocédural de Programmes Scientifiques*, is a research project that was started in 1988 at Ecole des Mines de Paris to investigate interprocedural issues. It is now mature enough to analyze realistic applications of a few thousands lines and its interprocedural analyses have produced interesting results for key problems, namely interprocedural parallelization, array privatization and loop selection for effective parallelism.

Two large examples are shown in section 2 to explain why automatic or manual parallelization requires extensive interprocedural analysis and how PIPS proceeds. The general structure of our prototype is briefly presented in section 3 to provide the basic elements necessary to understand the algorithms described in the next two sections. The computation of the interprocedural preconditions and regions, which were shown necessary for interprocedural parallelization in section 2, are detailed in section 4 and 5 respectively. Regions are computed with an improved version of the algorithm presented in [16] because MUST/MAY information is added to support array privatization. Results of these two kinds of analysis are obviously used for dependence computation and program parallelization but other applications are briefly presented in section 6.

2. EXAMPLES

An unusual amount of space is used here to show examples. Firstly, because we are dealing with interprocedural analysis, it is necessary to give a few procedures instead of a simple loop. Secondly, automatic interprocedural parallelization, if not analysis, is too often considered a lost cause. So it is important to use parts of real applications to support results in this area and to show the difficulty of interprocedural parallelization.

The first example, `ZPROAX`, shows that information must be propagated down the call tree to be able to perform array kill analysis and that memory effects must then be propagated upwards to restructure and parallelize a loop. The second one, `EXTRMAIN`, also shows that information must be propagated downwards, even to perform intraprocedural parallelization. It also shows how region computation relies on interprocedural information and how regions are later used to perform interprocedural parallelization.

2.1. Array kill information

Subroutine `ZPROAX` performs an intricate block matrix multiplication (see figure 1). This multiplication is decomposed into a matrix-vector multiplication (`MXV`) and an accumulation step (`MXVADD`), which uses a temporary workspace, `/FASTMEM/`, to preserve an intermediate value. This piece of code may not be optimal but it was extracted as such from an application.

The user willing to parallelize loop 1 in `ZPROAX` may use local syntactic information to

```
      SUBROUTINE ZPROAX(A,B,NOE,NC,X,Y)
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION A(NOE,NOE,3),B(NOE,NOE),X(NOE,NC),Y(NOE,NC)

      LYN=NOE*NC
      CALL VCLR(LYN,Y)
      NCI=NC-1
      DO 1 NNS=1,NCI
         CALL MXVADD(A(1,1,1),NOE,X(1,NNS),NOE,Y(1,NNS))
         CALL MXVADD(A(1,1,2),NOE,X(1,NNS+1),NOE,Y(1,NNS))
         CALL MXVADD(A(1,1,3),NOE,X(1,NNS),NOE,Y(1,NNS+1))
1     CONTINUE
      CALL MXVADD(B,NOE,X(1,NC),NOE,Y(1,NC))
      RETURN
      END

      SUBROUTINE VCLR(LX,X)
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION X(*)
      DO 2 I=1,LX
2        X(I)=0.
      RETURN
      END

      SUBROUTINE MXVADD(A,M,X,N,Y)
      IMPLICIT REAL*8(A-H,O-Z)
      COMMON/FASTMEM/Z1(640),Z2(640),D(640)
      DIMENSION A(M,N),X(N),Y(M)
      DO 3 I=1,M
3        Z1(I)=Y(I)
      CALL MXV(A,M,X,N,Y)
      DO 4 I=1,N
4        Y(I)=Y(I)+Z1(I)
      RETURN
      END

      SUBROUTINE MXV(A,M,X,N,Y)
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION A(M,N),X(N),Y(M)

      DO 5 I=1,M
5        Y(I)=0.
      DO 6 J=1,N,3
      DO 6 I=1,M
6        Y(I)=Y(I)+A(I,J)*X(J)+A(I,J+1)*X(J+1)+A(I,J+2)*X(J+2)
      RETURN
      END
```

Figure 1. Complex array-kill information for Z1 in MXVADD

avoid loop-carried conflicts on Y. He may decide to align the third statement of loop 1 so as to access Y(1,NNS) in each three statements and to peel off the first iteration. But he also has to pay attention to Z1 in /FASTMEM/ which does not appear in EXTR code. Summary data flow information (SDFI) shows him that this array may be read and written by each call to MXVADD. The following memory effects are found by PIPS for the first call in body of loop 1:

```
C               < MAY BE READ    >: A(*,*,*) X((/I,I=1,NOE,1/),NNS)
C                                   Y((/I,I=1,NOE,1/),NNS) MXVADD:Z1(*)
C               < MAY BE WRITTEN>: Y((/I,I=1,NOE,1/),NNS) MXVADD:Z1(*)
C               <MUST BE READ    >: NOE
        CALL MXVADD(A(1,1,1), NOE, X(1,NNS), NOE, Y(1,NNS))
```

Note that effects on X and Y are more precise than traditional SDFI and that they sometime are sufficient to parallelize loops with calls.

Intraprocedural region analysis shows that Z1(1:M) must be written in loop 1 before Z1(1:N) is read in loop 2. In other words, array Z1 is *killed* by procedure MXVADD and it may be stack-allocated to avoid output dependences between iterations of loop 1 in ZPROAX if it can be shown that M is greater than or equal to N. This piece of information is carried by the interprocedural precondition of MXVADD computed by PIPS:

```
C                          P() {M==N}
      SUBROUTINE MXVADD(A,M,X,N,Y)
```

Formal parameters M and N are always associated to the same actual argument NOE and this relationship is preserved although their numerical value is unknown. It is now possible to conclude that array Z1 is a temporary and that it does not have to be considered when restructuring loop 1 in ZPROAX.

Subroutine VCLR expects a vector as argument and a matrix is passed. Its *linearized* form is exploited and LYN contains the total number of elements. But its value is not an affine function of other integer variables and PIPS is not able to keep track of it:

```
C               < MAY BE WRITTEN>: Y(*,*)
C               <MUST BE READ    >: LYN
      CALL VCLR(LYN, Y)
```

2.2. Automatic parallelization

Program EXTRMAIN was adapted from an application (see figure 2). Modifications are shown in lower case in the main routine: arguments read from disk were not checked in the original version. It also was necessary to shorten the call graph for lack of space and interest. There should be an intermediate module between EXTRMAIN and EXTR to correctly set up arguments NI and NC.

The interesting loop here is loop 300 in EXTR which was slightly shorten to fit on one page. But all parallelization difficulties were preserved and it is not obvious to see if this loop can be run in parallel.

Let us consider the potential conflict between references T(J,1,NC+3) and T(JH,1,NC+3). The relationship between J and JH introduces two more variables, J1 and J2. The upper

```
      PROGRAM EXTRMAIN DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L

      READ(NXYZ) I1,I2,J1,JA,K1,K2
      if(j1.ge.1.and.k1.ge.1) then
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         call extr(ni,nc)
       endif
       END

      SUBROUTINE EXTR(NI,NC)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L

      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         JH=J1+J2-J
         T(JH,1,NC+3)=T(J,1,NC+3)
  300 CONTINUE
      END

      REAL FUNCTION D(J,K,JP,KP)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L

      D=SQRT((T(J,K,L )-T(JP,KP,L ))**2
     1      +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2      +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

Figure 2. Intricate use of array T

loop bound for index J still adds another one, JA. If these equalities and inequalities are used, it is necessary to assume a potential conflict.

However, this conflict actually never occurs because of constraints imposed on these variables during the initialization phase of program EXTRMAIN. PIPS propagates them downwards as preconditions in EXTR:

```
C P(JH, J, K, L) {J<=JA, J1<=J, 2<=J1, 2<=K1, L==NI, 1+J2==2JA, K==K1}
         T(J,1,NC+3) = S2*S3/((S1-S2)*(S1-S3))

C P(JH, J, K, L) {2<=K1, 2<=J1, J1<=J, J<=JA, J+JH==J1+J2, L==NI,
C                 1+J2==2JA, K==K1}
         T(JH,1,NC+3) = T(J,1,NC+3)
```

The letter P stands for Precondition because the predicate is true before the statement is executed. The list of arguments contains the variables which were modified between the procedure entry point and the current statement. The predicates are self-explanatory.

Since symbols in the predicates represent values, distance values are introduced for modified variables which may have a different value in each precondition. The dependence system is then built with both of them and it is easily found not feasible [13].

Loop 300 cannot yet be declared parallel because function D, which is used to initialize scalar S1, S2 and S3, also uses array T as SDFI information shows:

```
C                <MUST BE READ    >: K J L T(*,*,*)
C                <MUST BE WRITTEN>: S1
         S1 = D(J, K, J, K+1)
```

This information is not accurate enough to parallelize loop 300. If function D really uses every element in array T, there are conflicts between the call sites and the new definitions of T. Region analysis is able to refine the read effect:

```
C  <T(PHI1,PHI2,PHI3)-R-MAY-{L<=PHI3, PHI3<=L+2}>
         S1 = D(J, K, J, K+1)
```

Region information must first be decrypted! The comment line added by PIPS means that elements (PHI1,PHI2,PHI3) of array T *MAY* be *R*ead if the predicate on the PHI variables is true. It is obvious here that PHI1 and PHI2 are not constrained and that loop 300 must apparently be kept sequential because no relation between L and NC is known.

It is still possible to refine the region information by requiring an interprocedural analysis of scalar integer variables. Regions in the called procedures are then computed in a context set up by their callers. The interprocedural precondition for function D is:

```
C      P() {2<=J, 2<=K, 1+K==KP, J==JP}
       D = SQRT((T(J,K,L)-T(JP,KP,L))**2+(T(J,K,L+1)-T(JP,KP,L+1))**2+
      &(T(J,K,L+2)-T(JP,KP,L+2))**2)
```

Parameters J and K are bounded and the three calls to D create a strong relationship between parameter pairs J/JP and K/KP.

The elements of T which are parametrically read by D are now precisely characterized and bounds for PHI1 and PHI2 appear:

```
C  <T(PHI1,PHI2,PHI3)-R-MUST-{K<=PHI2, 2<=J, 2<=K, L<=PHI3, PHI2<=1+K,
                              PHI3<=2+L, 1+K==KP, J==JP, PHI1==J}>
      D = SQRT((T(J,K,L)-T(JP,KP,L))**2+(T(J,K,L+1)-T(JP,KP,L+1))**2+
     &(T(J,K,L+2)-T(JP,KP,L+2))**2)
```

This parametric region is translated at each call site in EXTR and PHI2 has a different value each time:

```
C  <T(PHI1,PHI2,PHI3)-R-MAY-{PHI3<=2+L, PHI2<=K+1, L<=PHI3, 2<=K,
                             2<=J, K<=PHI2, PHI1==J}>
         S1 = D(J, K, J, K+1)
C  <T(PHI1,PHI2,PHI3)-R-MAY-{PHI3<=2+L, PHI2<=K+2, L<=PHI3, 1<=K,
                             2<=J, 1+K<=PHI2, PHI1==J}>
         S2 = D(J, K+1, J, K+2)+S1
C  <T(PHI1,PHI2,PHI3)-R-MAY-{PHI3<=2+L, PHI2<=K+3, L<=PHI3, 0<=K,
                             2<=J, 2+K<=PHI2, PHI1==J}>
         S3 = D(J, K+2, J, K+3)+S2
```

They can now be used as basis for a dependence system if another region is associated to assignment statements like this one:

```
C  <T(PHI1,PHI2,PHI3)-W-MUST-{J<=JA, J1<=J, 2<=K1, 2<=J1, L==NI, J2+1==2JA,
                              K==K1, PHI3==NC+3, PHI2==1, PHI1==J}>
         T(J,1,NC+3) = S2*S3/((S1-S2)*(S1-S3))
```

It is now easy to prove independence because PHI2 equals 1 in the assignment and must be greater than 2 in function D (see K and its past relation with K1).

With the appropriate analysis options and after a privatization step, PIPS manages to find that loop 300 is parallel:

```
      DOALL J = J1, JA
         PRIVATE S3,S2,S1
         S1 = D(J, K, J, K+1)
         S2 = D(J, K+1, J, K+2)+S1
         S3 = D(J, K+2, J, K+3)+S2
         T(J,1,NC+3) = S2*S3/((S1-S2)*(S1-S3))
      ENDDO
      DOALL J = J1, JA
         PRIVATE JH
         JH = J1+J2-J
         T(JH,1,NC+3) = T(J,1,NC+3)
      ENDDO
```

3. PIPS OVERVIEW

PIPS is described in [13] and this overview is limited to elements necessary to understand analyses. PIPS, like every interprocedural tool, is designed on top of a data base to avoid global recompilations whenever possible.

To keep execution time at a reasonable level, each analysis is performed only once on each procedure and produces a summary result. These summaries are later used at call sites to avoid re-analyzing the same procedure twice for the same kind of information.

The call tree is assumed acyclic since recursive calls are prohibited by the Fortran standard and are not used in most real applications. Top-down and bottom-up analyses are controlled by a *make*-like mechanism, called *pipsmake*, which makes sure that necessary arguments like summaries or procedure abstract syntax trees are available before a new analysis phase is started.

Also, programs are assumed mostly structured. Restructuring techniques have been known since the late sixties and badly structured programs cannot be parallelized. Instead of using a control dependence graph or a general control flow graph, we have designed a hierarchical control flow graph. If the input program is structured, its abstract syntax tree (AST) is directly used to express its control flow. Algorithms are recursively defined over the underlying recursive data structure made of elementary statements, basic blocks, tests and loops.

Branches are kept as local as possible. If the branch and its target belong to the same block, this block is replaced by a control flow graph (called *unstructured*) but other elements in the AST, at a higher or lower level, are preserved as syntactic elements. Thus control effects are masked as much as possible and nodes in these local control flow graph contain abstract syntax subtrees. These subtrees may contain other unstructured. Thus algorithms can still be mainly recursively defined and quite accurately handle non-structured programs.

4. PRECONDITIONS

The preconditions used in PIPS are predicates over scalar integer variable values. These predicates hold just before a statement is executed. This explains their name of *precondition*. They do not have to be exact, or even optimal in some sense, but must be safe: preconditions must hold for any possible execution of the program.

Preconditions are computed in two phases. Procedure effects on integer variables are abstracted by *transformers* and propagated from bottom to top in the call tree. Preconditions are then broadcasted from the root procedure down to the call graph leaf procedures and transformers are used to interpret call statements.

4.1. Lattice

PIPS predicates express affine relationships between some sets of input data and some sets of output data. Each relationship is approximated by a polyhedron, which means that values are constrained by affine equalities and inequalities. The underlying lattice, build over polyhedra, is similar to Halbwachs' lattice [10] but used in a different way.

Halbwachs designed his analysis to discover relations among scalar variables holding true at some point of a program. A possible application of such an analysis is static array bound checking. PIPS ultimate goal is to parallelize programs. This is based on the dependence graph computation and requires some knowledge about *two* statements which may depend on each other.

Every predicate in PIPS is based on initial input values, e.g. `I#INIT`, and final output values, e.g. `I#NEW`. Sometimes intermediate values are also used to combine predicates. To make predicates more compact and easier to read, trivial equations like `I#NEW=I#INIT` are neither kept in memory nor printed because they are very numerous. Instead the set of variables for which this equation does not hold is shown as arguments of the predicate. For instance, `T(I) { I == N }` means that the value produced for `I` is no longer its input value. `I`'s output value is `N`'s output value and also `N`'s input value since `N`'s value was not changed. Such a predicate models the effect of the assignment statement: `I := N`.

Two kinds of predicates are used in PIPS. The first kind is called *transformer* and represents the effect of a statement, elementary or compound. They are denoted by a `T`. The other kind is called *precondition* and they are denoted by the letter `P`. They express the effect of a module on its store between its entry point and the decorated statement. For instance, `P(I) { I == N }` could be encountered after a test like `IF(I.EQ.N) THEN`.

Sets of input, output and intermediate values are computed for each module. Exact EQUIVALENCEs between variables are dealt with by sharing the same values and by introducing extra-equations. Badly equivalenced variables are not retained for transformer and precondition analysis. Variables in COMMON are handled like any other variable.

4.2. Transformers

As explained in section 3, transformer computation is a sequence of recursive calls to specialized algorithms, each dealing with one of the AST elements.

Elementary statements

Assignments of scalar integer variables are scanned. If the right hand side is affine over admissible variables, the assignment is entered as an equation in the predicate. If the same variable appears in both left-hand and right-hand side, its left-hand side occurence is renamed as new and its right-hand side as initial. Thus statement `I := I + 1` becomes `T(I) { I#NEW = I#INIT + 1 }`.

Basic blocks

A transformer is assumed to have been associated to each statement in the basic block. They now have to be combined two by two. New values in the transformer mapped to the first statement are renamed as intermediate values. Initial values in the second transformer are also renamed as intermediate values. The two transformers are then merged and intermediate values are eliminated by projection. This produces a new input/output relationship.

Tests

When the test condition is polyhedric, it is added to the transformer associated to the true body. When its negation is polyhedric, it is added to the false alternative transformer. Then the convex hull of the two predicates is used as predicate for the test transformer. Because of interactions between true and false transformers and the test conditions, additional information is retained although the convex hull of a condition and its negation is the always true predicate. This effect is visible in subroutine `ABSVAL` (see figure 3).

Some non-convex conditions like `I.NE.10` can also be dealt with accurately by taking into account three potential branches in the test: `I.LT.10`, `I.EQ.10` and `I.GT.10`. The convex hull of all branches is returned.

A faster algorithm is also available. It uses as unique input all definitions in the test and its two alternatives, to build a projection transformer. All modified variables are put in the argument field and no equation nor inequality is put in the predicate field. For instance, transformer `T(I,J,K,N) {}` will destroy any information you might have had previously gathered about any of the four variables, `I`, `J`, `K` and `N`.

Loops

Some kind of fix-point is needed to handle loops. To let the user trade speed against accuracy, modified variables can simply be eliminated by a projection transformer as for tests. It is also possible to compute a fix-point over equalities, à la Karr [14]. Once the loop body transformer is known, its equations are used to build an affine transformation matrix A and invariants, like induction variables, are given by computing a matrix of linear form T such that $T(A - I) = 0$ where I is the identity matrix. Halbwachs [10] iterative algorithm can also be applied if inequalities are sought.

Unstructured

A simple projection fix-point can be used for unstructureds as for loops. Else they can be accurately analyzed with Halbwachs algorithm which is designed to process any control flow graph.

4.3. Preconditions

Preconditions are propagated from the module entry point down to the AST leaves. Transformers, which are computed during a preliminary phase, are applied to preconditions to obtain postconditions, which usually are the precondition of the next statement.

Initial precondition

An initial precondition is derived from `DATA` and/or `PARAMETER` statements. If no information at all is available, an empty predicate is used to represent the set of all possibles values: `P() {}` . An initial precondition may also be computed interprocedurally when calling procedure are analyzed.

Elementary statements

The postcondition is obtained by applying the transformer to the precondition. Variables in the precondition and initial values in the transformer are renamed as intermediate values. The two predicates are put together and intermediate values are projected.

Basic blocks

The postcondition can be obtained in two different ways. Either the transformer associated to the block is applied or the last postcondition obtained from the statements in the block is used. The second solution was chosen in PIPS because it is more accurate,

albeit slower, when the block contains tests because more information is available when convex hulls are performed.

Tests

The precondition is narrowed into a true and a false postconditions with the test condition and its negation, when they are polyhedric. The two new postconditions are propagated along the true and the false branches and the convex hull of the two results is returned. Some non-convex conditions can be decomposed into a set of convex conditions as explained in the previous section. As for basic blocks, a faster but less accurate solution is to apply the test's transformer.

Loops

The loop body precondition is obtained by a fix-point operator. As for transformers, three levels of accuracy are possible. The fastest operator is the projection along all variables that are mutated in the loop body and iterator. The most accurate algorithm is Halbwachs because it is able to generate inequalities. If equalities are accurate enough, Karr's algorithm can be used.

Unstructured

As for transformers, unstructureds can be accurately dealt with using Halbwachs algorithm. As unstructureds are assumed rare, PIPS uses a simple projection. Variables modified in any node are projected. Every node in the control flow graph receives the projection as precondition. The postcondition of the unstructured computed by PIPS is not the convex hull of all its nodes' postcondition. The transformer associated to the unstructured is applied to its precondition.

4.4. Interprocedural Analysis

Procedures are processed in reverse post-order when transformers are synthesized and then in post-order when preconditions and postconditions are propagated. A transformer derived for a procedure body is stored as the summary transformer for that procedure and used each time the procedure is called, after a translation using bindings between actual and formal parameters . Preconditions propagated down to a call site are also translated and the summary precondition of the called procedure is replaced by its convex hull with the new precondition.

Summary transformer

The transformer computed for a procedure body may contain information about local variables. This information is meaningless for any caller since local variables are desallocated on return. The procedure transformer is projected along these variables and information about formal parameters, global variables and static variables is preserved.

Transformer translation

When a call to a procedure is encountered, its summary transformer is retrieved from the database. It is necessarily up-to-date and available because of pipsmake and because

the reverse post-order is enforced. The values in the summary transformer are named after variables in the callee. Equations are built to link formal parameters and actual arguments when they are affine. They are added to the summary transformer and formal parameters are projected. Global variables are simply renamed.

Precondition translation

Preconditions are computed during the top-down phase. Each time a call to a procedure is encountered, its precondition is available. The process is similar to transformer translation. Equations are added and local variables are eliminated. Global variables are also renamed and a precondition for the callee is obtained.

Summary precondition

If no summary precondition has been computed yet, the precondition for the current call site is used. If a summary precondition is already available, it is replaced by the convex hull of itself and the call site precondition. The order does not matter because the convex hull operator is commutative. This lets PIPS keep information about constant parameters, about ranges and about more complex, but useful, relationships. For instance, function `D` is called three times in `EXTR` and the relationships between formal parameters `J/JP` and `K/KP` are preserved:

```
C P() {2<=J, 2<=K, J==JP, K+1==KP}
```

4.5. Related work

In 1986 we pointed out in [16] that interprocedural parallelization could not be successfully performed without a precise symbolic semantics analysis which is used both directly in dependence tests and indirectly when regions are computed. Halbwachs analysis seemed then to be the most promising, but experience showed that it had to be adapted to procedure calls, to dependence testing needs and to large number of variables.

The interprocedural adaptation of Cousot and Halbwachs work is a version of the algorithm described in [4] for interprocedural constant propagation and more precise lattices. Transformers and preconditions are just particular instantiation of *return functions* and *jump functions.*

The algorithms described in the previous sections were implemented in PIPS years later and we are no aware of similar systems, although the use of symbolic constraints was suggested by others [9].

Many frameworks have been advocated to analyze programs and to parallelize programs. Symbolic evaluation [6] and abstract interpretation are often suggested, but, beyond correctness, it is important to know what the underlying lattice is and which operators are used. PIPS performs an abstract interpretation based on polyhedron theory which produces more general results than constant propagation or affine relationships. However, convexity conditions do not let transformers and precondition convey as much information as a conditional symbolic evaluation framework and this might be a serious deficiency according to [15]. This is also why transformers and preconditions remain small objects in memory.

```
      PROGRAM CALLABSVAL

      I = 1
      CALL ABSVAL(I, IABS)
C P(IABS, I) {I<=IABS, 0<=I+IABS, I==1}
      I = -1
C P(IABS, I) {0<=1+IABS, 1<=IABS, 1+I==0}
      CALL ABSVAL(I, IABS)
C P(IABS, I) {0<=I+IABS, I<=IABS, I+1==0}
      END

C T(NABS) {0<=N+NABS, N<=NABS}
      SUBROUTINE ABSVAL(N, NABS)
      IF(N.LT.0) THEN
         NABS = -N
      ELSE
         NABS = N
      ENDIF
C P(NABS) {0<=NABS+N, NABS<=1, N<=NABS}
      END
```

Figure 3. Polyhedric approximation

For instance, program CALLABSVAL in figure 3 is shown with preconditions in the main and at the return point of procedure ABSVAL. The procedure is summarized by its transformer as returning NABS which is always larger than both N and -N. The two postconditions shown after the calls to ABSVAL in the main were derived from this transformer and they turn out to be less precise than the ABSVAL's return precondition which indirectly implies that NABS equals 0 or 1.

It is possible to take advantage of the accurate precondition at the call site on one hand, and of the accurate analysis in the callee on the other hand, by using the intersection of the transformed precondition in the caller and the returned precondition in the callee.

5. REGIONS

A region, as defined in [16], is a set of array elements defined by affine equalities and inequalities. For instance, the effect on array A of the piece of code:

```
IF(I.LE.N) THEN
   A(2*I+1) = 0
ENDIF
```

is the set of elements $\{\phi_1 | \exists i \ \exists n \ \ \phi_1 = 2 * i + 1 \wedge i \leq n\}$ where ϕ_1 stands for a subscript value and i and n for values of variables I and N.

In order to support array kill and dependence analyses, two pieces of information are added. It is necessary to know if the array elements are defined (WRITE) or used (READ).

This is called the action. It is also necessary to know if every element in the region is certainly accessed (MUST) or simply potentially accessed (MAY). This is called the approximation. Of course, the array name is also retained. So regions are quadruplets and printed like:

```
C <T(PHI1,PHI2,PHI3)-R-MAY-{PHI3<=60, L<=PHI3, 1<=PHI1, 1<=PHI2,
                            1<=PHI3, C PHI2<=21, PHI1<= 52, PHI3<=L+2}>
```

This means that elements of array T whose subscripts meet the conditions may be read, but no others.

Regions are built bottom-up, intra- and inter-procedurally, from elementary references and statements in leaf procedures of the call graph, to the largest compound statement in the call graph root. It is assumed that affine preconditions are available for every statement beforehand. These affine preconditions may have been computed intra- or inter-procedurally, depending on the accuracy need.

5.1. Intraprocedural Analysis

Intraprocedural region analysis has to deal with elementary statements like assignments, basic blocks, tests and loops. The algorithm described here is very conservative as far as approximation is concerned because it was found sufficient to provide useful array-kill information for real applications.

Elementary statements

Each statement is scanned and each array reference is converted into a region whose predicate contains the statement precondition and equations linking the ϕ variables and the subscript expressions when these are affine. The approximation is set to MUST. The action depends on the reference position in the statement: only the left-hand side is defined.

If two regions reference the same array and if they are both using it or both defining it, they are merged and the new region predicate is the convex hull of their two predicates. The approximation becomes MAY because elements in the convex hull may not belong to any of the two initial regions. Consider assignment `X = T(1) + T(3)`. Element `T(2)` is not used although the statement region is `<T(PHI1)-R-MAY-{1 <= PHI1, PHI1 <= 3}`.

The dependence graph could be used to trade accuracy against space. Regions should only be merged if there is a dependence arc between the corresponding references. This has not been implemented because no real program has yet shown a need for it.

Basic blocks

Each statement has already been analyzed and associated with a set of regions. These regions may depend on variable values which are not constant over the basic blocks. Such variables are easy to find and the statement regions that reference them are projected along their values. Approximations for regions that are projected necessarily become MAY, even if the convex projection is exact. Regions that are combinable, because they do reference the same array and perform the same action, are merged and their approximation field is set to MAY.

Tests

Each branch has already been analyzed. Their associated regions are processed as in basic blocks: variables modified in the test body are eliminated and regions are merged. But every region approximation is set to MAY. Again, more complex strategies can be designed to preserve MUST information if there is a real need for it.

Loops

The set of regions associated with the loop body could almost be used as regions for the loop, but for the loop index which is obviously modified at each iteration. However, if the loop increment is 1, and if the upper and lower bounds are both affine and if the projection along the loop index is exact (see [2] for sufficient conditions), then a MUST region remains a MUST region after projection. This is the basis for array kill information. If a region does not use the index in its predicate, it also preserves its MUST approximation.

Nested loop 6 in figure 1 is an interesting example. A MUST region is produced for array Y because I's increment is 1 and because J is not used in Y's predicate. However regions for A and X are MAY region because a convex hull had to be applied at statement 6 level and also because J's increment is 3. It would be pretty difficult to show that A and X are entirely read and it would also be useless because use-def chain algorithms mainly need MUST information for definitions.

Unstructured

The small control flow graphs appearing in PIPS hierarchical control flow graph are handled in a very simple way. Sets of regions are associated to every node. Every variable modified anywhere in the unstructured has to be eliminated in these regions. All approximations are decreased to MAY.

5.2. Interprocedural region analysis

To preserve a reasonable analysis speed, each analysis is only performed once on each subroutine. The set of regions associated with a procedure body is filtered to eliminate local information and effects and to produce a region summary. This region summary is then used at each call site after a translation from the callee's name space into the caller's name space.

Summary regions

As explained, *summary regions* are derived for each subroutine by taking regions associated to the procedure body, which is just a large compound statement, by masking local effects, and by projecting local variables used in the predicate if ever some remain.

When no interprocedural information about a formal parameter is available, summary regions usually are not interesting. For instance the regions of T that is found read by function D is very large:

```
C <T(PHI1,PHI2,PHI3)-R-MAY-{PHI3<=2+L, L<=PHI3}>
```

because no relation was available between formal parameter pairs K and KP, and J and JP. PHI1 and PHI2 are not constrained, but by the array declaration.

However, interprocedural summary preconditions do change this as was shown in section 2. The previous region becomes:

```
C <T(PHI1,PHI2,PHI3)-R-MUST-{K<=PHI2, 2<=K, 2<=J, L<=PHI3, PHI2<=1+K,
C                            PHI3<=2+L, J==JP, 1+K==KP, PHI1==J}>
```

and lets PIPS parallelize loop 300 in EXTR.

Region translation

They are translated each time they are needed at a call site. Formal parameters are substituted with real arguments, when it is possible, in the predicate and in the reference. The usual affine conditions must hold for variables used in the predicate. Array arguments and formal parameters must conform. Regions are accurately translated if they both have the same dimension or if the formal array fits in a sub-array of the argument. For instance, a formal vector can be recognized as a matrix column. However reshapings like matrix linearization, as performed in the call to VCLR in ZPROAX, are not handled: affine conditions are not sufficient to prove than VCLR touches only elements in matrix Y.

Global arrays and global variables are just renamed from the callee's name space into the caller's name space. Commons are assumed to have the same layout in each procedure.

6. OTHER APPLICATIONS

MUST regions can be used to compute use-def chains and to privatize arrays. A write-must region kills a read region if the read region is included. Inclusion can be decided by converting the read region predicate into its equivalent generating system and by testing the generators against the write-must region constraints. In and out sets are also valuable in a parallel programming environment to show the users pieces of code which are algorithmically parallel although output dependences linked to uses of COMMON may prevent automatic parallelization. This would be very useful to guide a user willing to parallelize ZPROAX (see figure 1). He would know that Z1 has no algorithmic interest.

Preconditions are used to perform a partial evaluation of statements. This may improve code optimisability, eliminate dead code and also increase expression linearity. Preconditions can then be evaluated a second time and provide additional information.

Also user predicates can be added to summary preconditions and propagated over the whole program. This could be invaluable at the main level to precise the range and relationship of key parameters. It is very important to know how many iterations will be executed when one loop out of two parallel ones has to be chosen for effective parallel execution [3].

Finally, symbolic relationships between variables are used to estimate program time complexity interprocedurally [17]. For instance, PIPS computes the following polynomial for procedure ZPROAX (see figure 1):

```
C 32*NC.NOE^2 + 104*NC.NOE - 21*NOE^2 + 14*NC - 69*NOE + 2*U_RANGE - 1/3
```

References to intermediate variables like NCI, M or N have all been substituted interprocedurally thanks to preconditions. The special variable U_RANGE, unknown range, shows

that the estimation process failed somewhere. The failure can easily be tracked down because a complexity polynomial is attached to each statement. The call to `VCLR` cannot be handled because variable `LYN` is not linearly related to key parameters `NC` and `NOE`.

Basic numerical coefficients are tabulated for different machines and compilers. They can also be set to symbolically count the number of floating point operations or any other specific operation. These coefficients are combined according to the AST.

7. CONCLUSION

Complex information must be gathered to restructure programs beyond procedure boundaries. Two real life examples were used in section 2 to illustrate the power of interprocedural symbolic analysis and interprocedural region analysis in dealing with subtle interactions between variables. The key pieces of information used in the parallelization process were all automatically generated by PIPS. The algorithms we use are based on polyhedron theory and they mainly rely on projections and convex hull computations. They were described in great details so as to let the reader understand the basic steps performed by PIPS when analyzing a program.

We have received some very preliminary user feedback on PIPS. They mainly fell overwhelmed by the amount of information provided by the system. They would like to only receive relevant information, but the editing performed by hand in section 2 is not easy to program! PIPS was designed as a compiler and not as a parallel programming environment. Preconditions, regions, SDFI and complexity formulae are simply dumped in windows on the screen and this is far from fulfilling actual needs. For instance, users dream about receiving a short simple answer to the *simple* question: what is the most time-consuming loop in the program and why is it sequential? This is still far away and would require a much better understanding of the parallelization process. However the interprocedural analyses presented here provide a better basis to answer their questions than was ever before available.

PIPS is still being improved and future work will aim at increasing its robustness and its efficiency as well as measuring its qualities and pitfalls on real applications.

8. ACKNOWLEDGEMENTS

The author thanks Alexis Platonoff for his implementation of the region computation and Pierre Jouvelot for his careful proofreading of this paper.

9. REFERENCES

1 F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, An overview of the PTRAN analysis system for multiprocessing, *Journal of Parallel and Distributed Computing* , Vol. 5, No 5, (1988).

2 C. Ancourt, F. Irigoin, Scanning Polyhedra with DO loops, in the *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'91)*, Williamsburg, (1991).

3 W. Blume, R. Eigenmann, Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs, to appear in *IEEE Transactions of Parallel and Distributed Systems*, (1992).

4 C. D. Callahan, K. D. Cooper, K. Kennedy and L. Torczon, Interprocedural Constant Propagation, in the *Proceedings of the ACM Symposium on Compiler Construction*, (1986).

5 D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon, Parascope: A Parallel Programming Environment, in the *Inter. J. of Supercomputer Applications* (1988).

6 B. Dehbonei, P. Jouvelot, Semantical Interprocedural Analysis by Partial Symbolic Evaluation, in the *2nd ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation Techniques*, (1992).

7 E. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, D. Padua, Cedar Fortran and its restructuring compiler, in Languages and Compilers for Parallel Computing II, MIT Press, (1990).

8 P. Feautrier, Array Expansion, in the *Proceedings of the ACM International Conference on Supercomputing*, St-Malo, (1988).

9 M. Haghighat, C. Polychronopoulos, Symbolic Dependence Analysis for High Performance Parallelizing Compilers, in the *Proceedings of 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, MIT Press (1990).

10 N. Halbwachs and P. Cousot, Automatic Discovery of Linear Restraints Among Variables of a Program, in the *Conference Record of the Tenth ACM Annual Symposium on Principles of Programming Languages*, (1978).

11 M. Hall, K. Kennedy, K. McKinley, Interprocedural transformations for parallel code generation, Supercomputing'91, Albuquerque (1991).

12 P. Havlak, K. Kennedy, An Implementation of Interprocedural Bounded Regular Section Analysis, in *IEEE Transactions on Parallel and Distributed Systems*, v. 2, n. 3 (1991).

13 F. Irigoin, P. Jouvelot, R. Triolet, *Semantical Interprocedural Parallelization: An Overview of the PIPS Project*, ACM International Conference on Supercomputing (ICS'91), Cologne, 1991

14 M. Karr, Affine Relationships among Variables of a Program, *Acta Informatica*, (1976).

15 D. Maydan, Accurate Analysis of Array References, PhD dissertation, Stanford, (1992).

16 R. Triolet, F. Irigoin and P. Feautrier, Direct Parallelization of Call Statements, in the *Proceedings of the ACM Symposium on Compiler Construction*, (1986).

17 L. Zhou, Complexity Estimation in the PIPS Parallel Programming Environment, CONPAR'92, (1992).