Automatic Loop Parallelization: An Abstract Interpretation Approach

Laura Ricci
Department of Computer Science, University of Pisa
Corso Italia 40 - Pisa
email: ricci@di.unipi.it

Abstract

This paper presents an abstract interpretation to support the automatic parallelization of iterative constructs. The analysis approximates the range of values of any array subscript by integrating an interval analysis with one returning a set of simple linear restraints among program variables. To reduce the complexity of the analysis, a general methodology has been adopted that defines two different abstract domains, i.e. the interval domain and the bisector domain. The domains are then combined through the reduced product operator. A static tool implementing the analysis has been automatically generated through PAG. *The results of the analysis of some kernel loops of the Perfect Club Benchmark are shown.*

1. Introduction

The parallelization of sequential loops has been deeply investigated in the last decade. The basic analysis to implement this transformation is the *dependencies analysis*, which detects if several iterations of a loop refer to the same memory location and at least one modifies it. The problem is particularly challenging when array locations are referred within a loop body. An exact solution to this problem can be defined by solving a system of linear *Diophantine equations*. The proposals based upon this approach [10] are generally considered too computationally expensive to be practical. Hence, compilers generally exploit an *approximate method* which computes a safe approximation of the solution. One of the most widely used, even in commercial compiler, is the *Banerjee test* [10]. At first, this method computes, for each array reference R of a loop nest, an approximation of the *range of values* accessed by R, then it intersects different ranges to check if they overlap. If no overlap is detected, no dependency exists in the loop nest. A range is computed through a set of inequalities, the *Banerjiee inequalities*, returning the bounds of linear functions. The basic method has been extended to *triangular* or *trapezoidal loops* as well. In this case, the relations among loop indexes are exploited to restrict the range of values of the array subscript. The Banerjee test requires that both loop bounds and array subscripts are defined as *linear functions* of loop index variables. This implies that a subscript includes neither other variables but indexes nor any nonlinear expression. However nonlinear expressions or symbolic variables often appear in scientific programs, for instance in the *Perfect Club Benchmark* [11] .

Several proposals [12, 9] aim to generalize the Banerjee test. The *range test* [12] computes and compares *symbolic ranges* including both non linear terms and variables whose value is unknown at compile time. The test requires a symbolic analysis collecting equalities and/or inequalities between program variables. This information is exploited to detect *loop carried dependencies* for any loop L in a loop nest.

Each of these proposals is based upon a complex symbolic calculus implemented by an *ad hoc algorithm*.

This paper shows that a proper methodology can reduce the complexity of these analyses. We exploit the abstract interpretation framework [5, 6] to define a symbolic analysis for loop parallelization. A first advantage of this approach is that it reduces the complexity of the analysis through a *modular definition*, supported by a set of proper abstract operators. Furthermore, no *ad hoc algorithm* to implement the analysis has to be defined, because an analyzer can be automatically deduced from the specification of the analysis.

According to [5, 6], an abstract analysis can be defined through a set of simple abstract domains. Then the domains are *systematically combined*. In our case, two distinct domains are defined that describe, respectively, the numerical constraints of the variables and their symbolic relations. This is a noticeable difference with respect to existing analysis that introduce a single domain describing both properties. In our analysis, numerical constraints are described through the abstract domain of the intervals introduced in

Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)
0-7695-1730-7/02 $17.00 © 2002 IEEE

[5]. The analysis defines a new domain, the *bisectors domain* [2], describing a set of simple linear relations between variables. The interval and the bisector domain are systematically combined through the *reduced product operator* [6], which computes the Cartesian product of the two domains, then "reduces" each resulting element. The reduction exploits the information returned by either domain to improve that returned by the other one. In this way, when analyzing a statement, the analysis exploits the information returned by analysis of the previous statements on both domains. Furthermore, each abstract operator exploits the intervals to compute the relations and the other way round. This improves the accuracy with respect to an analysis that considers two domains, but does not integrate them.

The analysis has been implemented through PAG [1], an automatic analyzer generator. The abstract domains, the abstract operators and the reduced product are defined through a proper specification language.

The paper is organized as follows. Section 2 shows the definition of the abstract domains and of their reduced product. Section 3 introduces the abstract operators. In Section 4 the analysis is applied to some kernel loops of the Perfect Club Benchmarks. Section 5 describes the implementation in PAG. Section 6 presents related works. Finally Section 7 presents some conclusions.

2. The Abstract Domains

An abstract interpretation approximates program properties by evaluating a program on a simpler and computer representable domain of descriptions of "concrete" program states. This abstract domain exhibits a structure, i.e. an ordering, which is somehow present in the richer structure associated to the program execution. The relation between the concrete and the abstract domain is defined through a pair of functions, the *abstraction function* α and the *concretization function* γ that defines a Galois Connection [5]. The abstract interpreter, defined through a set of abstract operators corresponding to the instructions of the language, "executes" the program on the abstract domains. In this paper we discuss the abstract domain and the abstract operators defined for the dependency analysis. [7] shows The Galois Connection and the correctness of the analysis.

The Interval domain *Int* shown in Figure 1 has been introduced in [5] to describe the interval of values that each variable may assume at each point of a program. The domain can be easily extended to set of variables. The abstract calculus on this domain is defined according to the interval arithmetic. For instance, some basic operators are

$$\begin{array}{lcl} [a,b]+[c,d] & = & [a+c, b+d] \\ [a,b]-[c,d] & = & [a-d, b-c] \\ [a,b]*[c,d] & = & [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \end{array}$$

Since the abstract domain is infinite, a *widening* $\bigtriangledown$ and *narrowing* $\bigtriangleup$ are defined. In the following, we will exploit the definitions in [5]. The analysis defined on this domain will be referred as the interval analysis.

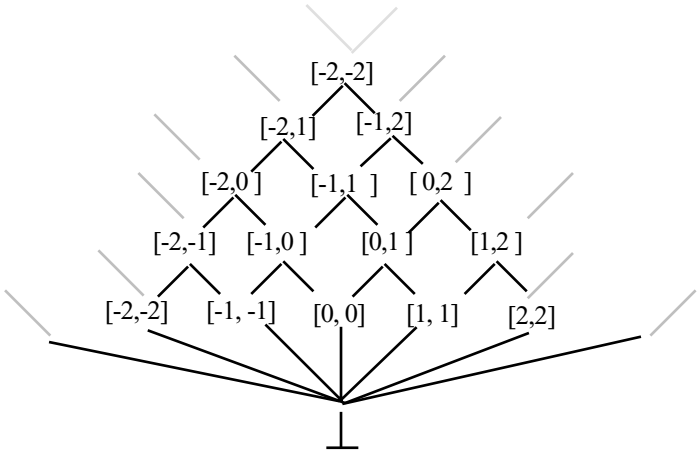


Figure 1. The Intervals Domain

The intervals returned by this analysis can be exploited to check array independence through the *Banerjiee Test* [10] whenever the loop indexes define a rectangular region. As a matter of fact, the analysis returns the same information of the Banerjee's inequalities. The analysis of triangular or trapezoidal loops, or of loops including symbolic variables, requires the definition of a new domain describing a set of simple linear restraints between the variables.

Let us consider a program that uses two variables only. In this case, the domain includes linear restraints which describe one of the regions produced by cutting the plane through bisecting lines: for this reason we call this domain the *Domain of Bisectors*.

For the sake of conciseness, we will show one subset of the domain only, that includes the restraints corresponding to the bisecting line $X = Y$. The domain is shown in Figure 2, where $X?Y$ means that there may be any kind of relationship between X and Y.

The specification of the complete *Bisectors Domain* is given in [7] and it includes the restraints corresponding to the bisecting line $X = -Y$ as well. The specification defines the domain corresponding to the two bisectors separately, and then combines them by defining their *reduced product*.

The abstract domain has been extended to an arbitrary number of variables, but, in order to reduce its complexity, the resulting analysis returns linear restraints only. Hence, the elements of the extended domain are a set of restraints, where each restraint belongs to the bisector domain and it involves only one pair of program variables.

Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)
0-7695-1730-7/02 $17.00 © 2002 IEEE

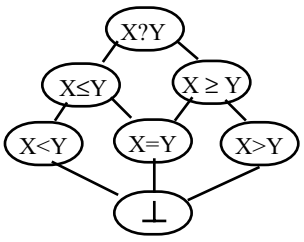


Figure 2. Bisectors Domain

A more refined approach combines the interval and the bisectors domain through their *Reduced Product*. This approach exploits a *reduction operator*: for each element (a_1, a_2) belonging to the Cartesian product of the original domains, the reduction operator may refine both a_1 and a_2 by considering the intersection of the concretization of a_1, rs. a_2 and by re abstracting the result through the corresponding domains. The reduced domain is produced by applying this operator to each element in the Cartesian product.

An *abstract state* includes, for each program point, an element of the reduced product, i.e. a numeric interval for each variable of the program and a set of relations between pair of variables. The abstract state where all the intervals are set to $[-\infty, +\infty]$ and the relations between any pair of variables is set to ? is associated with the initial point of the program.

Let us consider the reduction operator. Both this operator and the abstract operators of the following section will be defined with respect to the simplified bisector domain shown in this section. The reduction operator

$$\sigma : Bisector \times Int \times Int \rightarrow Bisector \times Int \times Int$$

is applied to a triple $(R_{x,y}, I_x, I_y)$, where $I_x = [m_x, M_x]$, $I_y = [m_y, M_y]$, and it may return a more precise relation or it may restrict the bounds of the intervals. We show only some cases of reduction. Let $m = max(m_x, m_y)$, $M = min(M_x, M_y)$

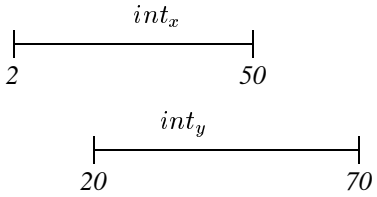
$$\begin{array}{lll}
\sigma(X = Y, I_x, I_y) = & \bot, \bot, \bot & \text{if } (I_x \cap I_y) = \emptyset \\
\sigma(X = Y, I_x, I_y) = & X = Y, [m, M], [m, M] & \text{if } (I_x \cap I_y) \neq \emptyset \\
\sigma(X > Y, I_x, I_y) = & \bot, \bot, \bot & \text{if } M_x \leq m_y \\
\sigma(X \geq Y, I_x, I_y) = & \bot, \bot, \bot & \text{if } M_x < m_y \\
\sigma(X \geq Y, I_x, I_y) = & X \geq Y, [m, M_x], [m_y, M] & \text{if } (I_x \cap I_y) \neq \emptyset \\
\sigma(X?Y, I_x, I_y) = & X > Y, I_x, I_y & \text{if } m_x > M_y \\
& \ldots &
\end{array}$$

The values returned by the operator when $X < Y$ or $X \leq Y$ can be trivially deduced from the previous cases.

Example 2.1 *Consider the reduction of the abstract state*

$$\sigma(X > Y, [2, 50], [20, 70])$$

the interval values of X and Y may be sketched as:



Since $X > Y$, all the values of X belonging to $[2, 20]$ can be discarded, because any value in this interval cannot be larger than any value of Y. Furthermore the values of Y belonging to $[50, 70]$ can also be discarded, because any value of Y in this interval is not lower than any value of X. Hence,the reduction operator returns the triple

$$\sigma(X > Y, [2, 50], [20, 70]) = X > Y, [21, 50], [20, 49]$$

Let us now consider the reduction of the triple:

$$X \geq Y, [17, 20], [3, 15])$$

Since the intersection of the intervals is empty, any value of X is larger than any one of Y. The reduction operator can now return a more accurate relation between the variables X and Y as follows
$\sigma(X \geq Y, [17, 20], [3, 15]) = (X > Y, [17, 20], [3, 15])$ $\diamondsuit$

The analysis is defined through a set of abstract operators corresponding to the constructs of the considered language. Each operator transforms the abstract state associated with the point of the program preceding the corresponding construct *(input abstract state)* into a new abstract state *(output abstract state)*, according to the semantics of the construct. The reduced product is applied to the abstract state returned by each operator. The analysis is iterated till a fix point is computed.

3. The Abstract Operators

The *abstract operators* to implement the analysis are fully described in [7]. This section shows the abstract operator corresponding to the assignment only. We will consider a general assignment involving n variables

$$X_j := c_1 X_1 + ... + c_i X_i + ...c_n X_n + c_{n+1} \qquad (1)$$

where the right hand side expression is *linear*, i.e. each c_i is constant. Non linear expressions are discussed in [7]. The assignment is *invertible* iff the variable on the left hand side also occurs on the right hand side. A relation between the new value of the left hand side variable and its previous value may be computed only if the assignment is invertible.

Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)
0-7695-1730-7/02 $17.00 © 2002 IEEE

An abstract assignment updates the interval corresponding to the left hand side variable and all the relations involving that variable, while all other relations are unchanged. Both domains introduced in the previous sections are exploited by the interval analysis and by the relational analysis described in the following. These analyses are applied *in parallel* to each program statement and both exploit any information in an input abstract state, i.e. the interval and the relations, to produce a component of the output abstract state. These components are then merged into a single output state by the reduction operator.

3.1. Relations Analysis

Let us consider assignment 1. The abstract operator inserts into the output state each relation in the input state involving any variable but X_j. If a relation involves X_j, it may be updated by the assignment, to take into account that the value of X_j is updated.

To compute the new relations, we exploit the results of the interval analysis as follows

- a new relation, between the left hand side variable and any right hand side variable can be computed through an analysis of the intervals of the right hand side variables.
- if the assignment is invertible, the relation between the new and the old value of the left hand variable can be exploited to update the relations of the abstract input state.

Let us consider the first case. To compute the relation between variables X_j and X_i, the assignment is transformed as follows

$$X_j := c_i X_i + [a_{exp}, b_{exp}] \quad (2)$$

where $[a_{exp}, b_{exp}]$ is the approximate interval of values of the expression exp resulting by removing the term $c_i X_i$ from the right hand side expression. This approximation is returned by the interval analysis described in section 2.

The new relation between X_j and X_i depends upon the signs of, respectively, c_i, the interval corresponding to X_i and $[a_{exp}, b_{exp}]$, as shown in following example.

Example 3.1 *Let us consider the assignment*

$$X_2 := 2X_1 + 3X_2 - X_3$$

Consider an abstract state including the intervals int_{X_1}*=[6,12],* int_{X_2}*=[2,10],* int_{X_3}*=[1,5] and the relations* $X_1 > X_2$, $X_2 > X_3$, $X_1 > X_3$, $X_3 < X_4$, *To compute the relation between* X_2 *and* X_1, *the variable* X_1 *is removed from the right hand side and the interval of the resulting expression is computed as follows:* $X_2 = 2X_1 + 3[2,10] - [1,5] = 2X_1 + [6,30] - [1,5] = 2X_1 + [1,29]$ *Since the value of* X_2 *is obtained by multiplying the value of* X_1, *which is positive, by a positive constant and by adding a positive value to the result, the analysis returns the relation* $X_2 > X_1$. ♢

This analysis is implemented by the function *NewRel* that is applied to an assignment statement a and a variable v and returns a relation between v and the left hand side variable of a. If v does not belong to the right hand side expression of a, *Newrel* returns the ? relation. The definition of *NewRel* [7], is straightforward, and is based upon an analysis of the possible combinations of the signs of the right hand side terms of 1. It is worth noticing that the function *NewRel* can be exploited to compute a relation between the old and the new value of the updated variable of an invertible assignment as well. This relation can be exploited to update the relation in the input abstract state as shown in the following example.

Example 3.2 *Let us consider again the assignment in example 3.1. The corresponding input abstract state includes the relation* $X_2 > X_3$. *Since* $NewRel$ *detects that the value of* X_2 *is increased by the assignment, the analysis can safely deduce that the relation* $X_2 > X_3$ *still holds after the assignment. Note that, in this case, Newrel cannot determine a relation between* X_2 *and* X_3 *because of the negative coefficient of* X_3. ♢

The function *Modrel* [7] implements the analysis shown in the previous example. It receives as input the relation R between the old and the new value of the updated variable X_j and a relation R' between the left hand side variable and any right hand variable and it checks if the latter is modified by the assignment.

The abstract operator *AbstrRel* exploits both the previous functions, as shown in Fig. 3. The function *NewRel* is initially invoked to compute a relation between the old value and the new one of the left hand side variable. If the assignment is not invertible, the function returns the ? relation. Then, each relation Rel in the input abstract state is considered: if it does not involve the left hand variable, it is inserted in the output abstract state. Otherwise, both *NewRel* and *ModRel* are invoked and the greatest lower bound of the results is returned. For instance, in previous example, the returned is the one computed by *Modrel*.

3.2. Intervals Analysis

This section shows that the relations defined in the *Bisector Domain* support a refinement of the numeric intervals. Let us consider again the assignment 1 and the abstract operator that, given an abstract state, returns an interval for the right hand side expression. The definition of this operator requires the solution of a linear programming problem, to compute the minimum and maximum value of

Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)
0-7695-1730-7/02 $17.00 © 2002 IEEE

```
AbstrRel(Assign,Relset);
begin
    R:=Newrel(Assign, LeftVar(Assign))

    // if LeftVar(Assign) ∉ RightVars(Assign) R = ⊤

    forall Rel ∈ Relset do
        if LeftVar(Assign) ∉ Vars(Rel) then
            Relout = Relout ∪ {Rel}
        else
        begin
            Var :=V ∈ { Vars(Rel)-LeftVar(Assign)}
            NR:= Newrel(Assign, Var);
            MR:= Modrel(Rel,R);
            New:=NR ⊓ MR;
            Relout := Relout ∪ New
        end
    endforall
end
```

Figure 3. Computing the Relations

the expression within the region defined by the numerical and the symbolic linear restraints belonging to an abstract state. While any classical linear programming algorithm may be exploited to solve this problem, these algorithms are not suitable for a static analyzer because of their complexity.

Our proposal is based upon an approximation of the *Fourier Motzkin* elimination method [10].

The function *AbstrInt*, implementing the refined interval analysis, is shown in Figure 4. For the sake of conciseness, we consider $\leq$ and $\geq$ relations only. The extension to the whole bisector domain is straightforward. *AbstrInt* receives as input an expression and an abstract state $S = (Relset, Inteset)$. It returns a safe approximation of the minimum value of the expression when it is evaluated in a state satisfying the constraints of S. A similar function has been defined for the computation of the maximum.

The function considers each variable v occurring in the expression and, if possible, it replaces v either with a symbolic lower bound or with an upper bound, according to the sign of the coefficient of v in the expression, which is returned by the function *Positive*. The function *choose* looks for an upper or a lower bound in the input abstract state. If no bound exists, the variable v is returned unchanged. If more than one bound is detected, one of them is chosen according to some *heuristics*. Currently, we exploit a straightforward heuristics which chooses a bound corresponding to any variable occurring in the expression. The definition of sophisticated heuristics is useless in our case, because most practical cases includes just one lower/upper bound for each variable. After replacing v, all the relations involving v are removed from the abstract state to avoid that v is inserted again in the expression by a following replacement. Furthermore, the variable x introduced by the substitution is marked and it will be not replaced before the simplification of the expression. As a matter of fact, since the expression can now include multiple occurrences of x it is not possible to decide whether x has to be replaced by either its lower or upper bound. When all variables have been considered, the expression is simplified through the function *simplify*.

Finally, the interval analysis introduced in section 2 is applied both to the simplified expression and to the original one and the tightest bound is returned. The following

```
AbstrInt(Expr,Relset,Intset)
substvar = ∅
forall v∈ Var(Expression)
    if v ∉ substvar then
        LB(v)={l:v≥l ∈ Relset }
        UB(v)={u:v≤u ∈ Relset }
        if Positive(v) then x=choose(LB(v))
                       else x=choose(UB(v))
        Expr = replace(x,v,Expr)
        substvar = substvar ∪ {x}
        Relset = Relset - Rels(v,Relset)
endforall
Simpexpr=simplify(Expr)
Int1 =interval(Simpexpr)
Int2 =interval(Expr)
Int = Int1 ⊓ Int2
return min(Int)
```

Figure 4. Computing the Intervals

example shows that the simplification does not always improve the accuracy of the analysis. In this case, the results of the interval analysis are returned.

Example 3.3 *Let us consider the following expression*

$$Y\text{-}X\text{+}K\text{-}T$$

and the following abstract state including the relations $X \leq Y$, $K \geq T$ *and the intervals* $X=[1, 10]$, $Y=[5, 20]$, $K=[5, 30]$, $T=[1, 20]$. *To compute the approximation of the minimum, the expression is scanned from left to right and the variable Y is initially considered. Since the coefficient of Y is positive, Y is replaced by its lower bound X and the first relation which involves Y is removed from the abstract state. Furthermore, X is marked to prevent any further substitution. Since K is then replaced by T, the resulting expression becomes:*

$$X\text{-}X\text{+}T\text{-}T$$

After the simplification of this expression, the returned minimum value is 0. Note that the analysis of interval applied to the original expression returns the value -20, which is a worse approximation. Consider the following expression

$$Y\text{+}X\text{+}K\text{+}T$$

Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)
0-7695-1730-7/02 $17.00 © 2002 IEEE

and suppose it is evaluated with respect to the previous abstract state. The same variables are replaced and

2X+2T

is returned, whose minimum value is 4. In this case, the result of the interval analysis, i.e. the value 12, is a better approximation. ♢

[7] shows that the operator always returns a safe approximation. The argument is that each elimination step corresponds to an elimination step of the *Fourier Motzkin method*, where some constraints are neglected.

Expressions including non linear terms require a more complex analysis. Before replacing a variable belonging to a non linear term, the signs of all the variables of the term have to be computed to decide if the variable can be replaced by an upper or a lower bound.

4. Applications

To prove the effectiveness of our approach, we have considered the *Perfect Benchmark*, a set of scientific programs widely exploited to test parallelizing compilers and applied our analysis to some kernel loops of this benchmark. This section shows that our analysis can prove the independence of these loops.

Let us consider the loop nest shown if Fig. 5, a simplified version of the *TRACK* code of the *Perfect Benchmark*.

```
ntrold = lsttrk                          S0
do k1 = 1, nm
    do kt = 1, ntrold
        if (k1 ≠ ihits(kt) then...       S1
    enddo
    if (...) then
        lsttrk = lsttrk + 1
        ihits(lsttrk) =...               S2
    endif
enddo
```

Figure 5. TRACK code: Symbolic Analysis

Let us consider the array *ihits* which is read in statement S_1 and modified in statement S_2. To prove that no dependency exists between these pair of references, the analysis computes the ranges of values accessed by each of these references and shows that they do not overlap. Notice that the values of the upper bounds of the loops and that of the variable *ntrold* are not known. For this reason, the interval analysis cannot infer a bounded interval neither for the reference S_1 nor for reference S_2. As a consequence, the interval analysis cannot prove independence for the loop nest.

Instead, the bisector analysis detects a symbolic upper bound, *ntrold*, for the set of values of the subscript S_1 and symbolic lower bound *ntrold+1* for the reference S_2. These symbolic bounds guarantee that the pair of references never overlap, independently of the value of *ntrold*. The upper bound is detected through an analysis of the bounds of the inner loop. To compute the lower bound, the analysis computes the relation $ntrold = lsttrk$, which hold after the execution of S_0. Since *lsttrk* is incremented in the outer loop, the relation $lsttrk > ntrold$ always holds before the execution of the statement S_2.

A more complex example is shown in Fig. 6. The code is one of the kernel loop of a *N-Body dynamics simulation* program. The program includes a set of complex subscripts, introduced by the elimination of *induction variables* from the original code. Let us consider the read access to v in S_0 and the write access to the same array in S_1. We want determine if any dependency carried by loop k exists between these references.

```
do i = 1,norder
    do k = 1,natmo3
        s=0.0
        do j = i, norder
        s=s+ c(i*norder+j-norder+1)*v(k+j*natmo3)          S0
        enddo
        v(i*natmo3+k-natmo3) = v(i*natmo3+k-natmo3)       S1
    enddo
enddo
```

Figure 6. MDG code: Symbolic Analysis

Since we are considering the dependencies carried by loop k, we have to analyze two different iterations k1 and k2 of the that loop. Following [12], at first we compute the integer range of the difference of the two subscripts, then we determine whether this range is always positive or always negative. The difference expression is

$$i * natmo3 + k1 - natmo3 - k2 - j * natmo3 \quad (3)$$

As in the previous example, an analysis taking into account only the interval returns an unbounded interval, because of the unknown value of the symbolic variables. Since expression 3 involves a pair of reference, we consider only the relations which holds *both* before statement S_1 *and* before statement S_2, that are

$$i \leq norder, k1 \leq natmo3, k2 \leq natmo3, i \leq j \quad (4)$$

These relations are exploited to refine the interval of values of expression 3. Let us consider the computation of the approximated maximum of this interval. The analysis can compute that both the value of the sub expression $i * natmo3 - j * natmo3$ and that of the sub expression $k1 - natmo3$ are always non positive because of, respectively, the relations $i \leq j$ and $k1 \leq natmo3$. Note

Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)
0-7695-1730-7/02 $17.00 © 2002 IEEE

that the analysis can prove that the non linear expression $i*natmo3-j*natmo3$ cannot be larger than 0, because of the interval $[1,\infty]$ inferred for the variable natmo3. Since the interval associated with k2 is $[1,+\infty]$, the maximum value of the expression is -1. This proves that the value of expression 3 is always negative and hence no dependency is carried by loop k.

5. Implementation

One analyzer has been automatically implemented in *PAG*. The specification language of *PAG* [1] includes a language for the definition of the domains and of the data types and one for the specification of the abstract operators. The former includes a set of constructs to define the elements and the ordering of the domain. It is possible to exploit some predefined domains or to define a new one. PAG automatically defines the reduction of two domains by exploiting the reduced product definition supplied by the user. The definition of the abstract operators is given through a monomorphical first order language, similar to Haskell [8]. The language may be exploited to define the reduction operator and the widening/narrowing operators as well. About 500 lines of code are necessary to specify our analysis.

6. Related Work

[12] proposes a *symbolic dependence tester* based on an analysis which presents some similarities with our approach. A single abstract structure, the *symbolic range*, describes both the numerical properties and the relations among the variables. This increases the complexity of the analysis. Furthermore, neither the abstract domain definition nor the correctness proofs are given. We recall that the definition of an abstract domain is essential to exploit tools for the automatic generation of the analyzers. Finally, the definition of the abstract operators is based on the manipulation and the simplification of arbitrary complex symbolic expressions. On the other hand, our analysis computes *integer ranges* for the variables and only a simple symbolic calculus is required when considering the relation between the symbolic domain and the numeric one. [9] presents a symbolic analysis for parallelizing compilers and performance estimators. An algorithm to estimate the number of integer solutions of a system of constraints is proposed. This algorithm support elimination of zero-trip loops, elimination of dead code, and performance prediction of parallel programs. [4] defines an abstract interpretation returning a set of linear restraints among the variables of a program. The analysis is very accurate, but its complexity is very high. A set of linear restraints is represented as convex polyhedron and the computation defined on the polyhedron requires the application of the pivot method.

7. Conclusions

This paper has presented an abstract interpretation for the parallelization of iterative constructs. We plan to extend the analysis both by refining the domains and by defining their composition with other abstract domains. For instance, the bisector domain can be extended to include bounds parallel to the bisecting lines. Furthermore, we plan to describe loops with different strides through the domain of arithmetical congruences.

References

[1] F.Martin. Pag, an efficient program analyzer generator. *Int.Jour. on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[2] I.Lari and L.Ricci. Array region analyses by abstract interpretation approaches. In *SBAC-PAD'2000 12th Symposium on Computer Architecture and high Performance Comp.*, pages 305–312, October 24-27 2000.

[3] M.Haghighat and C.D.Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

[4] P.Cousot and N.Halbwachs. Automatic discovery of liner restraints among variables of a program. In *6th ACM Conference on Principles of Programming Languages*, pages 84–95, 1978.

[5] P.Cousot and R.Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *5thACM Conference on Principles of Programming Languages*, pages 238–252, 1977.

[6] P.Cousot and R.Cousot. Systematic design of program analysis frameworks. In *6th ACM International Conference Principles of Programming Languages*, pages 269–282. ACM, 1979.

[7] R.Broccolucci. Interpretazione astratta per l'analisi dei descrittori di accesso. Master's thesis, Dipartimento di Informatica-Universita degli Studi di Pisa, February 2001.

[8] S.Thomson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.

[9] T.Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *The Journal of Supercomputing*, (12):1–29, 1998.

[10] U.Banerjee, R.Eigenmann, A.Nicolau, and D.Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–242, February 1993.

[11] W.Blume and R.Eigenmann. An overview of symbolic analysis techniques neede for the effective parallelization of the perfect benchmarks. In *Int. Conf. on Parallel Processing*, August 1994.

[12] W.Blume and R.Eigenmann. Non linear and symbolic data dependence testing. *IEEE Trans. on Parallel and Distributed Systems*, 9(12):1180–1194, 1998.

Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)
0-7695-1730-7/02 $17.00 © 2002 IEEE