AN OVERVIEW OF SYMBOLIC ANALYSIS TECHNIQUES NEEDED FOR THE EFFECTIVE PARALLELIZATION OF THE PERFECT BENCHMARKS®*

William Blume and Rudolf Eigenmann
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
blume@csrd.uiuc.edu and eigenman@csrd.uiuc.edu

Abstract: *We have identified symbolic analysis techniques that will improve the effectiveness of parallelizing Fortran compilers, with emphasis upon data dependence analysis. We have done this by comparing the automatically and manually parallelized versions of the Perfect Benchmarks®. The techniques include: symbolic data dependence tests for nonlinear expressions, constraint propagation, array summary information, and run time tests.*

Keywords: Automatic parallelization, symbolic analysis, dependence analysis, Perfect Benchmarks

INTRODUCTION

Parallelizing compilers are necessary to support standard programming languages on parallel machine architectures. This is very important because new high-performance machines and their parallel programming environments are changing rapidly. Also, the availability of standard programming languages is indispensable for non-computer experts who wish to use high-performance machines. Parallelizing compilation tools are commercially available and their technology is well documented [1, 23].

Unfortunately these commercial tools are not yet very effective. We have quantified and analyzed this situation in previous reports [4]. There have also been studies of potential improvements by manually parallelizing real programs and reporting the necessary transformation techniques [12, 11]. Table 1 shows some of these results. It compares automatically and hand-parallelized versions of the Perfect Benchmarks® that we ran on the Cedar multiprocessor.

For these new transformation techniques, the role of symbolic analysis is very important. This paper gives an overview of such analysis techniques and illustrates their applicability by code examples taken from time-consuming sections of the Perfect Benchmarks®. A more detailed description of these techniques can be found in [3].

ROLE OF SYMBOLIC ANALYSIS IN DATA DEPENDENCE TESTING

Typically, data dependence tests for parallelizing compilers demand that loop bounds and array subscripts are a linear (affine) function of loop index variables; that is, they are of the form $c_0 + \sum_{j=1}^{n} c_j i_j$ where c_j are integer constants and i_j are loop index variables. Unfortunately, array subscripts of real programs may include coefficients that are not integer constants, or the subscripts or array bounds may contain loop variant variables or subscript array references. We can deal with these situations by either eliminating the offending variable or subexpression from the subscript expression or by extending data dependence analysis to cope with symbolic expressions.

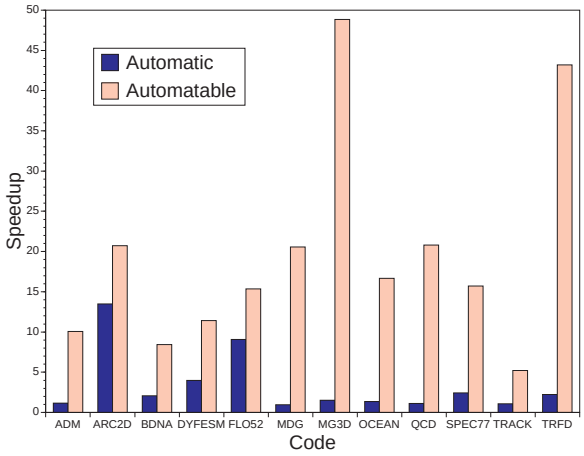

Figure 1: Speedups of automatically and manually parallelized versions of the Perfect Benchmarks® on Cedar

The techniques *constant propagation* and *induction variable substitution* are the most common methods used to transform array subscripts into testable linear expressions. Symbolic simplification of expressions is also important for canceling common terms and eliminating complex expressions. However in our analysis of the Perfect Benchmarks® we have seen examples of array subscripts that could not be transformed into the above form or that contained loop-variant expressions or subscript array references that could not be eliminated. Additionally, some common compiler transformations can introduce nonlinearities or non-constant terms to subscript expressions; two examples of such transformations are the linearization of arrays, which is often needed for inlining or interprocedural analysis, and the substitution of induction variables in multiply nested loops.

ANALYSIS OF THE PERFECT BENCHMARKS®

In our comparisons of the automatically and manually parallelized programs, we assume that the parallelizing compiler can do certain transformations well, (although these

*This work was supported by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the U.S. Army or the government.

transformations may not yet exist in current commercial compilers); that is, the compiler can privatize arrays [20], parallelize loops containing reduction statements, and parallelize loops with function calls [7, 6]. We also assume that the compiler is capable of performing symbolic analysis techniques that already have been well covered by others, including constant propagation of symbolic expressions [21], elimination of induction variables [15, 22], and symbolic simplification of expressions [8, 15]. However, we will mention cases where the transformations or analysis techniques above need minor modifications or more accurate information.

In our analysis of the Perfect Benchmarks®, we have found a variety of symbolic analysis techniques needed by the transformations described above to achieve the speedups of the manually parallelized versions. Except for *FLO52*, every code required some sort of symbolic analysis technique to improve its performance. However, the distribution of these techniques was quite uneven. Some codes, such as *BDNA* and *MDG*, required only a few techniques to allow parallelizing compilers to match the speedups attained from the manually parallelized versions. Other codes, most notably *QCD* and *TRACK*, need a long succession of complex analysis techniques just to parallelize a single important loop nest.

Rather than examining each code and describing what additional techniques will be needed to effectively parallelize it, we present the symbolic analysis techniques we have identified, and give examples showing why these techniques are important.

SYMBOLIC, NONLINEAR DEPENDENCE ANALYSIS

As mentioned above, current data dependence tests have difficulties in handling subscript terms that include non-constant coefficients. Very often, these program patterns consist of a multiply-nested loop where the inner loops access an array section, as in the following example. It is relatively easy to see that the array sections do not overlap in adjacent iterations of the outer loop.

```
do i = 0, n
   do j = 1, m
      x(m*i + j) = ...
   end do
end do
```

In some of these cases, the variable coefficients were introduced by the compiler. For example in the programs *MDG* and *TRFD*, such subscript expressions appeared in array subscripts after the elimination of an induction variable of a multiply nested loop. In *ADM*, non-constant subscript coefficients were the result of the linearization of a two dimensional array when a subroutine was inlined in an important loop. For example, the following important loop nest in *MDG* has an induction variable in the following doubly nested loop:

```
do i = 1, nt
   jj = i
   do j = 1, nor1
      var(jj) = var(jj) + ...
      jj = jj + nt
   end do
end do
```

After induction variable substitution, the loop is transformed into the following form, which includes a non-constant term:

```
do i = 1, nt
   do j = 1, nor1
      var(nt*j + i - nt)
.          = var(nt*j + i - nt) + ...
   enddo
enddo
```

Again, the above test for accessed array sections reveals independence.

In *OCEAN* we have found a more complex example, which takes 44% of the serial program execution time. The array sections accessed in this pattern can be detected as non-overlapping, similarly to the previous examples. However, the test now also has to consider array sections that are interleaved.

```
do jl = 1, i2k
   exj = ...
   do jj = jl, 64, 2*i2k
      do mm = 1, 129
         js = 129*jj + mm - 129
         js2 = js + 129*i2k
         h = data(js) - data(js2)
         data(js) = data(js) + data(js2)
         data(js2) = h * exj
      end do
   end do
end do
```

The need for a test that recognizes this situation was discussed by Eigenmann [12]. Maslov [17] presented the delinearization algorithm, which can handle any subscript expression $c_0 + \sum_{j=1}^{n} c_j i_j$ with symbolic loop-invariant expressions for the c_j's. Essentially, the delinearization algorithm partitions the array expression into several independent subexpressions and tests these partitions separately for dependences. Haghighat [14] describes how to prove that a subscript expression is strictly increasing or decreasing. Although it cannot be used to prove independence between two unequal subscript expressions, it is able to handle a more general class of symbolic expressions. For example, it can prove that there are no output dependences for the array write $a((i*i-i)/2+j) = \cdots$, where $1 \leq j \leq i$. We are currently developing a symbolic dependence test that can handle the examples we have seen [2].

CONSTANT PROPAGATION

By propagating constant values along all execution paths the constant value of program variables can be determined. This aids the analysis of subscript expressions by eliminating symbolic terms. Similarly, propagating symbolic expressions may remove loop-variant variables from the subscript expressions or may allow the compiler to determine additional relationships between these variables.

Interprocedural Constant Propagation with Procedure Cloning

In our code analysis we have often found a need to propagate values across procedure boundaries, sometimes with

the aid of procedure cloning[6]. For example, the code *OCEAN* requires both techniques to parallelize seven of its most important loop nests, accounting for 60% of the program's serial execution time. One of those loops is shown below. We have applied loop normalization, induction variable elimination, forward substitution, and dead code elimination to transform the loop into this form.

```
      do j = 0, mtrn-1
         work(1) = c1 * ac(n + q*j + 1)
         do i = 2, m/2, 1
            temp1 = ac(p*(2*i - 1) + q*j + 1)
     .            - ac(p*(2*i - 3) + q*j + 1)
            temp1 = c2 * temp1
            temp2 = ac(p*(2*i - 2) + q*j + 1)
            work(i) = temp1 + temp2
            work(m-i+2) = temp1 - temp2
         enddo
         work(m/2 + 1) = c3
     .        * ac(p*(2*(m/2) - 1) + q*j + 1)
         do i = 1, m, 1
            ac(p*(i-1) + q*j + 1) = work(i)
         enddo
      enddo
```

The symbolic index expressions `p*i + q*j` for array `ac` would disable traditional dependence tests, and symbolic tests fail because the values for `p`, `q`, `mtrn`, and `m` are not known and thus not comparable. The array privatization pass would also have difficulties in proving that array `work` is privatizable because the variable `m` must be evenly divisible by 2 for the entire array to be defined in the loop. Interprocedural constant propagation can resolve all these problems.

Guarded Constant Propagation

This technique deals with propagating values of variables that can take on one of several constant values, dependent upon the values of one or more boolean variables or expressions. We have found one code *(ARC2D)* where propagating guarded constants are essential for parallelizing a subroutine (`filerx`) that takes about 10% of the code's parallel execution time. The definition of these constants are:

```
L1    do j = 1, jmax
         jplus(j) = j+1
         jminu(j) = j-1
      enddo
S1    if (.not. peridc) then
         jplus(jmax) = jmax
         jminu(1) = 1
         jlow = 2
         jup = jmax-1
      else
         jplus(jmax) = 1
         jminu(1) = jmax
         jlow = 1
         jup = jmax
      endif
```

And the simplified body of routine `filerx` is:

```
L2    do n = 1, 4
L3       do j = jlow, jup
            work(j) = ...
         enddo
S2       if (.not. peridc) then
            work(1) = ...
            work(jmax) = work(jmax-1)
         endif
L4       do j = jlow, jup
            ... = work(jplus(j)) - 2*work(j)
     .            + work(jminu(j))
         enddo
         ...
      enddo
```

If the constant propagation pass takes control flow into account, it can see that `jlow` and `jup` take one of two constant values, depending upon the the constant boolean variable `peridc`. More specifically, `jlow = peridc ? 1 : 2` and `jup = peridc ? jmax : jmax-1` (borrowing the `?:` expression from the C language). Using this information, the compiler can determine that the range `work(1:jmax)` is defined at the start of `L4`. Thus, the definitions of `work` cover every use in the same iteration and array `work` can be privatized.

We have seen further examples in important sections of the programs *ARC2D*, *MDG*, and *QCD* where control flow must be taken into account in array def/use analysis for array privatization. An algorithm that deals with these situations was developed in a related project by Tu and Padua [20].

SYMBOLIC CONSTRAINT PROPAGATION

Symbolic constraint propagation gathers information from the program that can determine equalities and inequalities between program variables (e.g., $a < b$). This information can then be used to find the relationship between two arbitrary expressions. We have found this to be very useful for a variety of compiler passes, including data dependence analysis, array privatization, and dead code elimination.

Constraint propagation is needed by symbolic data dependence analysis for several purposes, such as determining whether certain variables are non-zero. For example, the following loop has no cross-iteration output dependences from `S1` to `S1` if and only if $n \neq 0$.

```
      do i = 1, 100
S1       a(n * i + c) = ...
      end do
```

Constraint propagation can also be very useful at identifying that there is no dependence for array subscripts containing loop variant variables. The code *TRACK* has several important loops that need such an analysis. A greatly simplified version of one of these loops is:

```
      ntrold = lsttrk
      do k1 = 1, nm1
         do kt = 1, ntrold
 S1         if (k1 .ne. ihits(kt)) then ...
         end do
         if (...) then
            lsttrk = lsttrk + 1
 S2         ihits(lsttrk) = ...
         endif
      end do
```

Current data dependence tests are unable to compare the use of `ihits(kt)` at statement `S1` with the definition of `ihits(lsttrk)` at statement `S2` and would have to assume that a dependence exists. However, using the fact `lsttrk` > `ntrold` at `S2`, gathered by constraint propagation, a compiler can determine that there is no dependence between `S1` and `S2`. Haghighat [13] describes how a popular data dependence test, (i.e., Banerjee's inequalities test), can be extended to use symbolic constraint information.

The array privatization technique needs to determine whether a range of array elements that is defined (e.g., `a(1:m)`) covers another range of elements used (`a(1:n)`). This example involves the comparison of the bounds of the ranges (i.e., is `m` $\geq$ `n`?). Constraint propagation can improve the accuracy of these tests. For example, in the following time-consuming loop of the code *BDNA*

```
      do i = 1, n
         do k = 1, i-1
S1          xdt(k) = ...
         end do
         l = 0
         do j = 1, i-1
            if (...) then
               l = l + 1
S2             ind(l) = j
            end if
         end do
         do j = 1, l
S3         ... = xdt(ind(j))
         end do
      end do
```

the presence of the subscript array `ind` would prevent the privatizer from determining any relationship between the array ranges accessed in statements `S1` and `S3`. However, the fact that `ind(1:l)` $\leq$ `i-1` found at `S2` can be propagated to `S3`, and hence the variable `xdt` can be privatized.

Dead code elimination of conditional statements has turned out to be very useful in the context of last value assignments generated by induction variable substitution. The following excerpt of program TRFD has an induction variable (`mijkl`) in a loop nest that is nested four deep.

```
      do mi = 1, morb
         do mj = 1, mi
            ...
            do ml = mj, mi
               xijkl(mijkl + ml - mj + 1) = ...
            end do
            if (mj - 1 .le. mi)
S1             mijkl =  mijkl + mi - mj + 1
            end if
            ...
            do mk = mi + 1, morb
               ...
               do ml = 1, mk
                  xijkl((mk*mk - mi*mi
     .                    - mi - mk)/2
     .                   + ml + mijkl) = ...
               end do
            end do
            if (mi .le. morb) then
S2             mijkl = mijkl + morb
     .                 + (morb*morb - mi*mi
     .                    - mi - morb)/2
            end if
         end do
      end do
```

The code example is taken after the induction variable has been substituted in the inner two loops. The conditional statements around `S1` and `S2` were inserted in this process, which now prevents induction variable substitution from eliminating `mijkl` entirely from the loop nest. However, by using constraint propagation, the compiler can determine that both tests are always true, which allows the outermost loop to be parallelized.

There has been some work in the determination of variable constraints. Much work has been done in determining the possible range, or interval, of values that variables can take, for the purpose of array bounds checking or program verification [16, 5]. These algorithms, however, only propagate integer ranges. Cousot and Halbwachs [10] offer a powerful algorithm for determining symbolic linear constraints between variables. Their algorithm is based upon the calculation, intersection, and merging of convex polyhedrons in the n-space of variable values. However, their algorithm cannot handle nonlinear expressions such as $a < b * c$. Although it is not too common, we have seen cases where nonlinear bounds must be propagated or nonlinear expressions must be compared. Because of this, we will be looking into methods that can handle such nonlinear expressions as well.

SUBSCRIPT ARRAY ANALYSIS

In our analysis of the Perfect Benchmarks®, we have found that a small but significant fraction of subscripts include arrays (e.g., `x(index(i))`, which disable all known data dependence tests. In a few of these cases the index arrays are initialized to constant symbolic expressions at the beginning of the program and never modified. Hence, any use of these arrays can be replaced with their constant expressions. To our knowledge, there has not been any previous published work in this area. One example is a loop nest in *TRFD* which, if not parallelized, accounts for 25% of the parallel execution time. The loop uses a subscript array `IA` that has the value `IA`$(i) = (i * (i-1))/2$. This knowledge can be used to parallelize the loop successfully.

In cases where subscript arrays are non-constant, there can still be useful information to gather from the program. Information whether the array is singly valued or monotonically increasing or decreasing is very useful for eliminating false dependences [18]. Knowing the difference between adjacent index array values also aids dependence analysis. The minimum or maximum value of the array is very useful for array privatization, as shown previously in the example for the usefulness of constraint propagation in *BDNA*.

One code that would benefit greatly from subscript array information is *DYFESM*. Most of the time-consuming loops in this program include subscripted subscripts. Using the described information, we believe that many of these loops can be identified as parallel automatically.

RUNTIME TESTS AND OTHER TECHNIQUES

Runtime Tests. Sometimes, proving that a loop is parallelizable at compile time is impossible or too expensive. In these cases, runtime tests with two version loops may parallelize them. For example, suppose that a given loop cannot be parallelized unless a certain condition is true. To handle this, the compiler inserts a conditional statement that tests this condition. If it is true, a parallelized version of the loop nest is executed. Otherwise, the program executes the sequential version. We have seen situations where such techniques are applicable in the programs *DYFESM, ADM,* and *MG3D*. The situations range from simple tests, such as whether or not a variable is greater than a threshold, to tests for more complex symbolic conditions. Details and examples are given in [3].

There were a few other important symbolic analysis techniques that were necessary for the effective parallelization of the Perfect Benchmarks®. Although these techniques are computationally expensive or have limited applicability, we do believe they are worth mentioning. Again we refer to [3] for more details.

Compile Time Interpretation. One important, but very expensive technique is the compile time interpretation of programs. Essentially, the idea is to execute the program without input data; that is, to perform abstract interpretation [9] where the abstractions in the analysis are kept to a minimum. One example is the determination of whether an array is filled with the factors of some scalar, which we have seen in program *ADM*. Another example occurs in *QCD*, where much of the code cannot be parallelized unless the control flow of a specific routine can be determined. This flow is defined by the content of an array that can be seen as an instruction stream. The routine interprets this stream to determine what operations it should perform on other inputs.

Algorithm Recognition. Another expensive technique required by some codes is algorithm recognition. Basically, the compiler must recognize that a given code fragment implements a certain algorithm so that it may be replaced with a parallel version. Algorithm recognition and replacement can be feasible if the code fragments are small and relatively simple. For example, some commercial parallelizing compilers can replace matrix multiplies or recurrences with library calls. When the algorithms to be replaced are longer and more complicated, performing algorithm recognition becomes more challenging. We have met such situations in the programs *QCD* and *SPEC77*, where the compiler must recognize a random number generator and a linear search in a sorted array, respectively.

SUMMARY

The techniques required to parallelize the Perfect Benchmarks® and their importance are shown in Table 1. A number in a cell for a specific code and technique is the estimated slowdown incurred from the manually parallelized code if the technique could not be used; that is, the slowdown equals t_e/t_a, where t_e is the estimated time taken with the technique disabled and t_a is the time taken by the manually parallelized version. The value t_e was calculated by assuming that all important loop nests[1] that use the given technique could not apply the transformations that allowed the faster execution times for the nest (i.e., such nests will have an execution time equal to the time taken by the automatically parallelized version of the loop nest). Other important loop nests, which did not use the technique, have an execution time equal to the manually parallelized versions. An empty cell indicates that no important loops use the given technique. The last column, *Automatable speedups*, displays the speedup from the automatically parallelized codes to the manually parallelized codes using only techniques that could be implemented in a compiler, which is the ratio between the two bars of each code in Figure 1.

One caveat to Table 1 is that the techniques are not orthogonal. First, some techniques are dependent upon information provided by others. For example, symbolic, nonlinear expression data dependence tests almost always need the information provided by constraint propagation so that they can effectively compare expressions. Secondly, some important loop nests can be parallelized by using one symbolic analysis technique or another. The only examples that suffer from this problem in Table 1 are the runtime test and compile time interpretion techniques for *ADM* and *MG3D*. For these slowdowns, either of the two techniques can be used to parallelize the important loop(s).

CONCLUSION

Other work has shown that current commercial parallelizing compilers perform poorly on real codes [4], and that a compiler can theoretically achieve good speedups for these codes [12, 11]. Motivated by this, we examined the Perfect Benchmarks® to determine what symbolic analysis techniques are required to get the observed good speedups. The techniques that we identified ranged from minor extensions of current techniques to complex and expensive transformations. The most interesting of these techniques are: symbolic, nonlinear expression data dependence tests, constraint propagation, guarded constant propagation, constant array propagation, subscript array analysis, and the generation of run time tests. We are currently implementing these techniques within the Polaris parallelizing compiler [19], which is being developed at the University of Illinois.

We believe that the symbolic analysis techniques that we have identified, along with other powerful techniques such as interprocedural analysis, array privatization, improved handling of induction variables, and reduction parallelization, can significantly improve the effectiveness of parallelizing compilers on real codes. A preliminary implementation of the nonlinear, symbolic data dependence test and constraint propagation algorithm supports this; it has been able to identify all the parallel loops for all the examples in the symbolic, nonlinear data dependence section. We also believe that some of these symbolic analysis techniques will be beneficial for other kinds of optimizing compilers.

[1] We consider a loop nest as *important* if its parallelization may significantly affect the speedup of the entire program.

Code	Nonlinear Depend. Anal.	Inter. Const. Prop.	Guard. Const. Prop.	Constraint Prop.	Const. Array Prop.	Array Anal.	Run Time Tests	Compile Time Interp.	Alg. Recog.	Auto-Matable Speedup
ADM	1.1	3.3		1.8			4.3†	4.3†		7.8
ARC2D		1.1	1.1	1.1		1.1				1.5
BDNA				2.1		2.1				4.1
DYFESM	3.6	3.1				3.6	3.6			3.9
FLO52										1.7
MDG	1.2	18.5		1.2						20.3
MG3D		36.9		36.9		36.9	36.9†	36.9†		36.9
QCD	11.3			19.0	19.0	19.0		19.0	11.4	19.1
OCEAN	8.3	11.5		8.3					1.5	12.2
SPEC77									2.4	6.4
TRACK				2.9			3.3			5.3
TRFD	18.2			18.2	1.4					18.2

†For *ADM* and *MG3D* the given slowdown is incurred only if neither runtime tests nor compile time tests can be used.

Table 1: Estimated amount of slowdown on Cedar from manually parallelized codes if a specific symbolic analysis technique could not be used.

REFERENCES

[1] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2), February 1993.

[2] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Nonlinear Expressions. Technical report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., April 1994. CSRD Report No. 1345.

[3] W. Blume and R. Eigenmann. Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. Technical Report 1332, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., January 1994.

[4] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions of Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[5] F. Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Design and Implementation*, pages 46–55, June 1993.

[6] P. Briggs, K. D. Cooper, M. W. Hall, and L. Torczon. Goal-Directed Interprocedural Optimization. Technical report, Rice University, November 1990. TR90-147.

[7] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural Constant Propagation. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, June 1986.

[8] T. E. Cheatham Jr., G. H. Holloway, and J. A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, SE-5(4):402–417, July 1979.

[9] P. Cousot and R. Cousot. Abstract Interpretation: A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[10] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[11] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res, & Dev., April 1992.

[12] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[13] M. Haghighat and C. Polychronopoulos. Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 310–330. MIT Press, 1991.

[14] M. Haghighat and C. Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs. *Presented at the sixth Annual Languages and Compilers for Parallelism Workshop, Portland, OR*, August 12–14, 1993.

[15] M. Haghighat and C. Polychronopoulos. Symbolic Program Analysis and Optimization for Parallelizing Compilers. *Presented at the fifth Annual Workshop on Languages and Compilers for Parallel Computing, New Haven, CT*, August 3–5, 1992.

[16] W. H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[17] V. Maslov. Delinearization: an Efficient Way to Break Multiloop Dependence Equations. *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 152–161, June 1992.

[18] K. S. McKinley. Dependence Analysis of Arrays Subscripted by Index Arrays. Technical report, Rice University, June 1991. TR91-162.

[19] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A New-Generation Parallelizing Compiler for MPP's. Technical Report 1306, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June 1993.

[20] P. Tu and D. Padua. Automatic Array Privatization. *Presented at the Sixth Annual Languages and Compilers for Parallelism Workshop, Portland, OR*, August 12-14, 1993.

[21] M. N. Wegman and K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[22] M. Wolfe. Beyond Induction Variables. *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, June 1992.

[23] H. P. Zima and B. M. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1991.