Array Resizing for Scientific Code Debugging, Maintenance and Reuse

Corinne ANCOURT
CRI - ENSMP
35 rue Saint Honoré
77305 Fontainebleau Cedex, France
ancourt@cri.ensmp.fr

Thi Viet Nga NGUYEN
CRI - ENSMP
35 rue Saint Honoré
77305 Fontainebleau Cedex, France
nguyen@cri.ensmp.fr

ABSTRACT

Software debugging, maintenance and reuse have to deal with many problems from Fortran scientific codes that do not fully respect the standard specification. Our study on Linpack, PerfectClub and SPEC95 benchmarks and several industrial software reveals a large number of unprecise variable declarations that prevent program analysis, verification and parallelization. Furthermore, they decrease the readability of programs and make reverse-engineering more difficult.

This paper presents two different methods to compute the exact size of arrays in Fortran codes that have pointer-type `REAL A(1)` or assumed-size `REAL A(*)` declarations.

The first method uses the relationship between actual and formal arguments from parameter-passing rules. New array declarations in the called procedure are computed with respect to the declarations in the calling procedures.

The second approach is based on an array region analysis that gives information about the set of array elements accessed during the execution of code. This approach to array resizing could be applied to other languages without array declaration such as MATLAB and APL in order to reduce the execution overhead of dynamic test and resizing.

Our two approaches are combined to yield very good results for Linpack, PerfectClub and SPEC95 benchmarks.

Keywords

program analysis, program comprehension, debug, reuse, reverse-engineering, array declaration, array resizing, array region.

1. INTRODUCTION

In many programming languages, array declarations are very important for program analyses such as array aliasing, array bound and array initialization checking. It is the responsibility of the programmer to allocate a large enough storage space for arrays since any attempt to read from or write into a location outside the declared ranges is an error, and may result in unexpected results, security holes or failures.

In many languages, some array dimension declarators of formal arguments are implicit. In C and Java, the programmer can replace the first array dimension declarators by `[]`. In Fortran, it is the last array dimension that can be declared with an assumed-size declarator `(*)`, because arrays are allocated in column-major order. Array argument declarators with `1` (which do not respect Fortran 77 standard) instead of `*` are also encountered.

Our study of scientific applications from Linpack, PerfectClub [?] and SPEC CFP95 [?] benchmarks shows a large number of unnormalized array argument declarations. Many of them have 1 as the upper bound for the last dimension of the array declaration although array references in the corresponding procedure are outside the defined extent of the array. This causes premature aborts due to bound violations when the programs are compiled with an array range checking option. In [?], the declarations had to be fixed by hand, which is not an easy process. Like the assumed-size array declarator, this unnormalized feature in Fortran prevents array bound checking and alias analysis that are critical for code safety and debugging.

Furthermore, when these kinds of array declarations are used by programmers, code understanding, one of the core software engineering activities, becomes more difficult. Program comprehension is really needed to maintain, reuse, re-engineer and enhance software systems.

The numbers of unnormalized and assumed-sized arrays in Table 1 (fifth column: Total) show how often this kind of declarations occurs, in 18 out of 23 programs (the benchmark *adm* from PerfectClub is not used because it is exactly the same as benchmark *apsi* from SPEC CFP95). The average percentage of these declarations is 59.66% out of the total array argument declarations in the 18 programs. To ease code maintenance and debugging, an automatic phase is essential if we do not want to infer thousands precise declarations manually.

Array declarations can raise problems not only in Fortran but also in other class of programming languages that do not have variable declarations. A large execution time overhead for dynamic array allocation and resizing in MATLAB [?] is due to repeated reallocation and copying. As matrix and vectors sizes are not declared by the programmer, the MATLAB interpreter allocates storage for these vari-

PASTE'01, June 18-19, 2001, Snowbird, Utah, USA..

ables on demand, during program execution. An attempt to write into a matrix element outside the bounds of the matrix element causes the system to reallocate storage for the entire matrix, copying over all elements from the old storage to newly allocated space. This becomes very expensive if these operations are done within loops. In JAVA, the dynamic allocation of the Vector class exhibits the same problem. In this case, an implicit appropriate preallocation, when possible, would be interesting to improve performance.

A judicious use of declarations can significantly improve the quality of code that can be generated by an APL compiler as shown in [?]. Precise array declarations are also important for data distribution in Fortran compilers for message-passing machines as in [?]. Furthermore, our technical report [?] shows that the normalization of array declaration is unavoidable for array range checking.

As a result of these experiments, we developed two new methods to find out automatically the proper upper bound for the unnormalized and assumed-size array declarations, a process we call *array resizing*. Both methods are implemented in PIPS [?, ?, ?], a free, open and extensible workbench for automatically analyzing and transforming scientific applications. The goals of PIPS are program compilation, reverse-engineering, program verification, source-to-source program optimization and parallelization. Its interprocedural analyses help with program understanding and with checking the legality and the impact of automatic program transformations.

The first approach to array resizing is a top-down analysis based on the *association rules of arguments* in Fortran standard [?]. This method, which is a whole program compilation, tries to find the exact array argument declarations in the called procedure with respect to the declarations in the calling procedures.

The second approach is a bottom-up analysis using *array region* analysis [?, ?]. New array sizes are computed from the sets of actually accessed array elements during the execution of the program. The main program is not always necessary and this approach can also be used for library maintenance.

The paper is organized as follows. Section 2 contains a running example to describe the problems and the two approaches. Section 3 and Section 4 present in detail, respectively, the top-down and the bottom-up approaches. The experimental results and the comparison between the two algorithms are discussed in Section 5. Conclusions are given in the last section.

2. RUNNING EXAMPLE

The example in Figure 1 is extracted from the Linpack benchmark. It contains 7 unnormalized and assumed-size array declarations that should be resized such as `IPVT(1)`, `A(LDA,1)` in `DGESL` and `DX(*)`, `DY(*)` in `DAXPY`.

By propagating information down the call tree, the procedure call `CALL DGESL(A,LDA,N,IPVT,B)` in the main program shows that the actual array `MAIN:A(201,200)` is associated with the formal array `DGESL:A(LDA,1)` and as we have $LDA = 201$, the second dimension of the array `DGESL:A` could be replaced by `1:200`. In the same way, we have new array declarations: `DGESL:B(200)`, `DGESL:IPVT(200)`, `DGEFA:B(200)` and `DGEFA:IPVT(200)`.

On the other hand, the intraprocedural array region analysis in subroutine `DAXPY` shows that array region `DX(1:M)` must be read, and that `DY(1:M)` must be read and written, assuming $M \geq 1$, which can be proven by using interprocedural information. So new declarations `REAL DX(M)` and `REAL DY(M)` can be induced by finding the real array accesses in this subroutine.

```
      PROGRAM MAIN
      REAL A(201,200),B(200)
      INTEGER IPVT(200)
      LDA = 201
      N = 100
C     P(LDA,N) {LDA==201, N==100}
      CALL DGESL(A,LDA,N,IPVT,B)
      CALL DGEFA(N,IPVT,B)
      END

      SUBROUTINE DGESL(A,LDA,N,IPVT,B)
      INTEGER LDA,N,IPVT(1)
      REAL A(LDA,1),B(1)
      DO 10 K = 1, N-1
        L = IPVT(K)
C     P(LDA,N,K) {LDA==201, N==100, 1≤K, K≤99}
        CALL DAXPY(N-K,L,A(K+1,K),B(K+1))
10    CONTINUE
      END

      SUBROUTINE DGEFA(N,IPVT,B)
      INTEGER N,IPVT(1)
      REAL B(1)
      DO 20 K = 1, N-1
        L = IPVT(K)
C     P(N,K) {N==100, 1≤K, K≤99}
        CALL DAXPY(N-K,L,B,B(K+1))
20    CONTINUE
      END

      SUBROUTINE DAXPY(M,DA,DX,DY)
      REAL DX(*),DY(*),DA
C     <DX(PHI1)-EXACT-{1≤PHI1, PHI1≤M}>
C     <DY(PHI1)-EXACT-{1≤PHI1, PHI1≤M}>
      DO 30 I = 1,M
        DY(I) = DY(I) + DA*DX(I)
30    CONTINUE
      END
```

Figure 1: Running example, excerpt from Linpack

More details about the two above approaches are given in the following sections.

3. TOP-DOWN APPROACH

This approach is based on the association rules of formal and actual arguments[1] in Section 15.9.3.3 of the Fortran 77 standard [?]. In a program, the number and the size of dimensions in an actual argument array declaration may be different from those in an associated formal argument array declaration (*array reshaping*). A formal array can be associated with an actual array or with an actual array element.

In the first case, the size of the formal argument array must not exceed the size of the actual argument array. The size of an array is equal to the number of elements in the array: $\prod_{i=1}^{n} d_i$ where n is the number of dimensions of array, $d_i = u_i - l_i + 1$ is the size of the i-th dimension in which l_i and u_i are respectively the corresponding lower and upper bounds.

[1]The actual Fortran terminology is "dummy" and "actual" arguments

In the second case, the size of the formal argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element. As Fortran language allocates array in column-major order, the subscript value of an array reference $\mathrm{A}(\mathrm{s}_1, \mathrm{s}_2, \cdots, \mathrm{s}_n)$ is $1 + \sum_{i=1}^{n} \left((\mathrm{s}_i - \mathrm{l}_i) \prod_{j=1}^{i-1} \mathrm{d}_j \right)$. Note that $\prod_{j=1}^{0} \mathrm{d}_j = 1$.

```
procedure Top_Down_Array_Resizing(p)
  /* p:  current procedure */
begin
  for each a1 ∈ set of 1 and * arrays in p
    V := ∅ /* set of new bound values */
    for each q ∈ set of callers of p
      for each c ∈ set of calls for p in q
        a2 := corresponding_argument(a1,c)
        k := number_equal_dimensions(a2,a1,c,p,q)
        s2 := size_of_actual_array(a2,k,c,q)
        s2':= translate_to_callee_frame(p,s2,c,q)
        if (s2' != s2) then
          s1 := size_of_formal_array(a1,k,p)
          new_value := s2'/s1
        else
          new_value := *
        endif
        V := V ∪ {new_value}
      endfor
    endfor
    if |V|=1 then
      last_upper_bound(a1) := element(V)
    else
      last_upper_bound(a1) := *
    endif
  endfor
end
```

Figure 2: The top-down array resizing algorithm

Our algorithm is a top-down traversal of the call graph of the program that is described in Figure 1. We always begin by the main program and the new sizes of array arguments in the called procedure are computed with respect to the actual declarations in the calling procedures.

Function $number\_equal\_dimensions(a2, a1, c, p, q)$ computes k, the number of first equal dimensions of the actual array $a2$ and the formal array $a1$. When the actual argument is an array element, k is also the number of first subscripts that are equal to their corresponding lower bounds. This step simplifies the computation of array sizes and the subscript value expressions. The size of the actual array in procedure q is calculated by using all dimensions but the first k ones. If the actual argument is an array element, their subscript value expression is evaluated only from the subscript $k + 1$. The size of the formal array is computed in the same way, omitting the last dimension which has an unnormalized or assumed-size bound to be replaced.

Function $translate\_to\_callee\_frame(p, s2, c, q)$ tries to translate all variables in the size expression of the actual argument from the caller's name space into the callee's name space, by using the relations between formal and actual arguments, and between the declarations of global variables in both routines. If the new values contain variables that cannot be translated in the scope of the callee, or if these values are different for different call sites, we have to keep an assumed-size declarator for the array. If not, we have the same values for different call sites and they are in the callee's frame, thus the array can be resized with this new upper bound value.

Preconditions [?], an auxiliary analysis in PIPS, are also taken into account. They are affine predicates over scalar integer variable values that hold true before the execution of the corresponding statement. Preconditions are propagated from the module entry point down to the abstract syntax tree leaves. Unstructured programs [?] are also handled in PIPS.

The precondition of the current call site gives more information for the simplification and the translation steps.

```
SUBROUTINE DGESL(A,LDA,N,IPVT,B)
INTEGER LDA,N,IPVT(200)
REAL A(LDA,200),B(200)
END

SUBROUTINE DGEFA(N,IPVT,B)
INTEGER N,IPVT(200)
REAL B(200)
END

SUBROUTINE DAXPY(M,DA,DX,DY)
REAL DX(*),DY(M+100),DA
END
```

Figure 3: New declarations with top-down approach

For the running example, array element passing in DGESL: $\mathtt{CALL\ DAXPY(N-K,L,A(K+1,K),B(K+1))}$ and DGEFA: $\mathtt{CALL\ DAXPY(N-K,L,B,B(K+1))}$ makes it more difficult to find the right sizes for arrays DAXPY:DX and DAXPY:DY.

For DX, the corresponding actual array size in the caller DGESL is $\mathtt{LDA} * 200 + 1 - (1 + (\mathtt{K} + 1 - 1) + (\mathtt{K} - 1) * 201)$ and in the caller DGEFA is 200. By using the preconditions

$$\mathtt{DGESL} : \mathtt{N} = 100$$

and

$$\mathtt{DGESL} : \mathtt{LDA} = 201$$

shown in Figure 1, and the binding information

$$\mathtt{DGESL} : \mathtt{N} - \mathtt{DGESL} : \mathtt{K} = \mathtt{DAXPY} : \mathtt{M}$$

after the translation to the scope of DAXPY, we have two different values: $20201 + 202 * \mathtt{M}$ and 200. So array DX still has an assumed-size array declaration.

However, all actual array sizes of array DY can be translated to $100 + \mathtt{DAXPY} : \mathtt{M}$ and the new declaration is $\mathtt{DY}(100 + \mathtt{M})$. 6 out of 7 arrays have been resized with appropriate bounds (Figure 3).

4. BOTTOM-UP APPROACH

The bottom-up approach has been designed to normalize programs when initial declarations are not explicit in the main program or when the main program is missing e.g for numerical libraries. A typical example is a program where arrays are declared as pointers and dynamically allocated by *malloc*-like functions. In its subroutines, these arrays are passed as arguments but their actual sizes are not always in the argument list.

One solution is to analyze the program behavior and to try to extract information from the dynamic allocation function arguments in order to find the actual array size and then use the top-down approach. However, the allocation space computation often depends on the type of the elements and operators so this solution seems unrealistic. Furthermore, when the main program itself is missing or for programming languages with dynamic resizing, we need another method to find actual array accesses.

The bottom-up approach computes, without knowledge of the initial array declarations, adequate declarations. It is based on a convex array access analysis whose purpose is to identify the set of array elements accessed during the execution of a given section of code. An **convex array region**, as defined in [?, ?], is a set of array elements described by a set of affine equalities and inequalities. These constraints link the region parameters that represent the array dimensions to the values of the program integer scalar variables.

A region has the approximation MUST if every element in the region is certainly accessed, and the approximation MAY if its elements are simply potentially accessed. In fact, they are under- and over-approximation of EXACT which means that the region exactly represents the accessed set of array elements. A major source of inexactness comes from the lack of structure of programs when **goto**s are used instead of structured tests and loops. The aggressive restructuring phase implemented in PIPS is necessary to gather more information about array references. Another limitation is due to nonlinear expressions or some convex region operators. For instance, the region

$$< \mathtt{A}(\phi_1, \phi_2) - \mathtt{EXACT} - \{\phi_1 = \mathtt{I}, \phi_1 = \phi_2, 1 \leq \mathtt{I} \leq \mathtt{N}\} >$$

corresponds to an array element access A(I, I). The region parameters ϕ_1 and ϕ_2 respectively represent the first and second dimensions of A.

Regions are built bottom-up, intraprocedurally, from the deepest nodes to the largest compound statement nodes in the hierarchical control flow graph. At each meet point of a control flow graph, the region information pieces from different control branches are merged with a meet operator. The approximation of regions is conservative. Intraprocedural region analysis has to deal with elementary statements like assignments, basic blocks, tests, loops and the control flow graph.

The interprocedural propagation of regions is a bottom-up analysis of the program call graph. At each call site the summary regions of the called subroutine are translated from the callee's name space into the caller's name space, using the relations between actual and formal parameters, and between the declarations of global variables in both routines. *Intra*procedural array regions are not sufficient when an array is passed as an actual argument by nested subroutines, but no array elements are accessed in these subroutines except the last one. The *inter*procedural propagation is needed to find the access region for this array.

In each routine, for each array argument with non-precise declaration, the following steps are performed:

1. Compute the regions for read and written accesses to the array in the routine using [?].

2. Fuse read and written accesses to the array.

3. Project unnecessary variables from the region: $\phi_j, \forall 1 \leq j < n$ where n is the last array dimension.

4. Eliminate redundant constraints to get an accurate upper bound.

5. Compute the maximum value of the last dimension variable ϕ_n from the remaining constraints. If the system has several upper bounds and their relations are known at compile time, the preconditions are taken into account to find this maximum value.

As in the first approach, the implementation returns * when the maximum operator fails. By applying the bottom-up algorithm, array regions permit to resize 6 out of 7 arrays for the running example (Figure 4). However, in subroutine DGESL, as the reference B(L) is not affine (L = IPVT(K)), information about array regions is lost and we cannot find out a new value for the upper bound of B.

```
SUBROUTINE DGESL(A,LDA,N,IPVT,B)
INTEGER LDA,N,IPVT(99)
REAL A(LDA,*),B(100)
END

SUBROUTINE DGEFA(N,IPVT,B)
INTEGER N,IPVT(99)
REAL B(100)
END

SUBROUTINE DAXPY(M,DA,DX,DY)
REAL DX(M),DY(M),DA
END
```

Figure 4: New declarations with bottom-up approach

Results of this approach depend on the accuracy of the array region analysis. The execution time is longer than with the first approach because of feasibility tests on linear systems. But once regions are computed, execution times for the two approaches are similar. The global execution time could be improved using more sophisticated implementation of array regions, such as guarded regions, list of regions [?] or linear memory access descriptors [?]. However, in most practical cases, the required time is acceptable and good results for array resizing are reported in Section 5.

5. EXPERIMENTAL RESULTS

The percentages of arrays resized by the two methods for 18 benchmarks from Linpack, PerfectClub and SPEC CFP95 are shown in Table 1. The fifth column presents the total number of unnormalized and assumed-size array declarations. The 7th and 9th columns respectively show the numbers of actual resized arrays (* is not included) by the top-down and bottom-up approaches.

Over 18 benchmarks, the first approach gives 12 better, 1 equal and 5 worse results. It is specially good for *bdna, flo52, ocean, su2cor, hydro2d* and *applu* with more than 80% resized arrays. In benchmarks *spice, trfd* and *apsi*, it is not a whole array but array elements that are passed in most of the procedure calls. There are variables in the array size and subscript value expressions that cannot be translated into the callee's space name. Or we may have several calls to the same procedure from different callers and the new upper values are different, as in the running example. Furthermore, information is lost right after the beginning of the top-down traversal in the callgraph so the results for these three benchmark are not so good. In fact, the variables cannot be translated into the callee's frame because they are

Bench	Array	1	*	Total	Percen	TD	Percen	BU	Percen	TD+BU	Percen
linpack	16	13	3	16	100.00%	10	62.50%	11	68.75%	14	87.50%
bdna	228	0	191	191	83.77%	172	90.05%	46	24.08%	186	97.38%
dyfesm	81	0	19	19	23.46%	6	31.58%	1	5.26%	6	31.58%
flo52	73	18	0	18	24.66%	18	100.00%	3	16.67%	18	100.00%
mdg	40	29	0	29	72.50%	19	72.50%	12	41.38%	21	72.41%
mg3d	84	68	0	68	80.95%	54	79.41%	29	42.65%	62	91.18%
ocean	60	55	0	55	91.67%	47	85.45%	17	30.91%	50	90.91%
qcd	32	0	11	11	34.38%	5	45.45%	8	72.73%	10	90.91%
spc77	286	11	0	11	3.85%	8	72.73%	8	72.73%	11	100.00%
spice	38	34	0	34	89.47%	1	2.94%	19	56.00%	20	58.82%
trfd	22	0	13	13	59.09%	3	23.08%	5	38.46%	6	46.15%
su2cor	47	0	5	5	10.64%	5	100.00%	1	20.00%	5	100.00%
hydro2d	28	0	7	7	25.00%	7	100.00%	1	14.29%	7	100.00%
applu	20	0	18	18	90.00%	17	94.44%	13	72.22%	18	100.00%
turb3d	60	43	15	58	96.67%	45	77.59%	35	60.34%	45	77.59%
apsi	633	145	292	437	69.04%	8	1.83%	211	48.28%	219	50.11%
fpppp	26	19	0	19	73.08%	8	42.11%	3	15.97%	8	42.11%
wave5	153	1	69	70	45.75%	39	55.71%	26	37.14%	62	88.57%
Average					**59.66%**		**62.80%**		**40.65%**		**79.18%**

Table 1: Numbers of total array argument declarations, unnormalized array declarations, assumed-size array declarations, number and percentage of unnormalized and assumed-size array declarations, number and percentage of resized arrays by the top-down approach, number and percentage of resized arrays by the bottom-up approach, number and percentage of resized arrays by the combined approach.

neither in the argument list nor in the common blocks and the preconditions for non-linear expressions, indirections,... are not available. The average percentage of resized arrays is 62.80% with the current implementation.

The top-down approach could be improved by using more accurate precondition analyses or **code instrumentation**. One of our future work is to instrument the code by adding new parameters that are array sizes or subscript values into the argument list. By using this additional technique, 100% unnormalized and assumed-sized array declarations are resized but there is a trade-off between static and dynamic analyses. For example, the stack sizes may be augmented due to additional parameters.

The second approach is more time consuming and the average number of resized arrays is smaller than the first one, 40.65%. The reasons for these and possible improvements are mentioned in Section 4. However, we have about 70% resized arrays for *linpack, qcd, spc77, applu* and much better results for *spice, apsi* where the first algorithm does not work. Furthermore, this approach is more effective on programs having non-explicit initial declarations and it is the only solution for libraries.

In fact, the return values of array bounds given by the two approaches can be different because they are based on different assumptions. The first approach is an *interprocedural analysis* that uses the association between two program units. The new array declarations in the called procedure are computed with respect to the declarations in the calling procedure. As a consequence, these new declared ranges can be too large for the actual array accesses or, by contrast, can detect *intraprocedural array bound violations* in the execution of the called procedure.

On the other hand, the second approach is rather an *intraprocedural analysis* because it is based on actual array references in one program unit. This method trusts the execution of code and there will be no more out-of-bound errors for array arguments in the current procedure. But we can always check for *interprocedural array bound violation* [?] in the whole program.

So if the considered program is supposed to be correct, there is no danger if either the first or the second approach to array resizing is applied. Otherwise, if our purpose is to verify the validation of program, both approaches can be applied and they must be followed by the array bound checking phase.

Since neither approach is always superior, it is desirable to combine the two approaches to have as many as possible proper array declarations. A composed solution has been implemented: the less expensive, top-down analysis, is used first, then the second approach is applied only for subroutines in which precise declarations have not been found. About 80% unnormalized and assumed-size arrays are resized (two last columns in Table 1).

6. CONCLUSION

To our knowledge, there has not been much related work done with array resizing. Our two methods have solved the problems of unprecise array declarations in Fortran scientific codes as discussed in the introduction section. In the worst case, the pointer-type declarations `A(1)` that disable many program analyses and optimization are replaced by the assumed-size array declarations by both algorithms.

The first approach is appropriate for Fortran with its assumed-size array declarations but it is not always applicable to other programming languages. We get very good results for Linpack, PerfectClub and SPEC CFP95 benchmarks by using this approach. A higher percentage of resized arrays could be achieved with code instrumentation but there will be a trade-off between the static and dynamic techniques. Our future work is to improve the static analyses and to study the impact of code instrumentation in order to combine these techniques.

However, when the array is allocated dynamically and its real size is difficult to find, or when dealing with the class of languages without variable declarations, the second approach is needed to find the explicit declarations. Furthermore, this method can also be applied to other programming languages such as MATLAB, APL and JAVA to improve performance by reducing the execution overhead of dynamic resizing. And, of course, other program analyses and verifications will be facilitated.

Fewer than 1000 lines of C code are sufficient to implement both approaches in our research compiler PIPS. Encouraging experimental results are achieved with not only these three scientific benchmarks but also other large scale industrial applications, over 100 000 lines. The PIPS software and documentation as well as our array resizing implementations are available on *http://www.cri.ensmp.fr/pips*.

7. REFERENCES

[1] *Programming Language FORTRAN,ANSI X3.9-1978, ISO 1539-1980*. American National Standard Institute, 1983.

[2] T. Budd. *An APL Compiler*. Springer-Verlag, 1988.

[3] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Lecture Notes in Computer Science - Languages and Compilers for Parallel Computing*, pages 46–60. Springer-Verlag, August 1995.

[4] J. J. Dujmovic and I. Dujmoviv. Evolution and evaluation of SPEC benchmarks. In *ACM:SIGMETRICS*, pages 2–9, 1998.

[5] J. Gu and Z. Li. Efficient interprocedural array data-flow analysis for automatic program parallelization. *IEEE Transactions on Software Engineering*, 26(3):244–261, March 2000.

[6] F. Irigoin. Interprocedural analyses for programming environments. In *Environments and Tools for Parallel Scientific Computing*, pages 333–350. Elsevier Science Publisher, 1993.

[7] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[8] R. Keryell, C. Ancourt, F. Coelho, B. Creusile, F. Irigoin, and P. Jouvelot. PIPS: A workbench for building interprocedural parallelizers, compilers and optimizers. Technical Report A/289/CRI, Ecole des Mines de Paris, May 1996.

[9] A. Kubota, I. Miyoshi, K. Maeyama, S. Goto, S. Mori, H. Nakashima, and S. Tomita. TINPAR: A parallelizing compiler for message passing multiprocessors. In *International Symposium on Parallel and Distributed Supercomputing*, pages 214–223, September 1995.

[10] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In *ACM SIGPLAN 2nd Conference on Domain-Specific Languages*, volume 35, pages 53–65, Austin, Texas, USA, October 1999.

[11] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[12] T. V. N. Nguyen, F. Irigoin, C. Ancourt, and R. Keryell. Efficient intraprocedural array bound checking. Technical Report A/316/CRI, Ecole des Mines de Paris, November 2000.

[13] Y. Paek, J. Hoeflinger, and D. Padua. Simplication of array access patterns for compiler optimizations. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[14] A. V.Aho, R. Sethi, and J. D.Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[15] R. W.Hockney. *The Science of Computer Benchmarking*. SIAM, 1996.