Electronic Notes in Theoretical Computer Science 65 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html pages

Alias verification for Fortran code optimization

Thi Viet Nga NGUYEN François IRIGOIN

Centre de Recherche en Informatique - Ecole des Mines de Paris
77305 Fontainebleau Cedex, France
Email: nguyen,irigoin@cri.ensmp.fr - Fax: +33 1 64 69 48 47

Abstract

Alias analysis for Fortran is less complicated than for programming languages with pointers but many real Fortran programs violate the standard: a formal parameter or a common variable that is aliased with another formal parameter is modified. Compilers, assuming standard-conforming programs, consider that an assignment to one variable will not change the value of any other variable, allowing optimizations involving the aliased variables. Higher performance results but anything may happen: the program may appear to run normally, or may produce incorrect answers, or may behave unpredictably. The results may depend on the compiler and the optimization level.

To guarantee the standard conformance of programs and to maintain the referential transparency in order to make program analyses exact and program optimizations safe, precise alias information retrieval at a reasonable cost, especially the determination of overlaps between arrays are studied in this paper. Static analyses and code instrumentation are used to find all violations of the prohibitions against aliasing in Fortran code. Alias violation tests are inserted only at places where it cannot be proved statically that they are useless in order to reduce the number of dynamic checks at run-time. A specific memory location naming technique is used to compact representation and enhance the precision of alias analysis. Modifications on the dependence graph created by aliasing are also studied to show the impact of aliases on some program optimizing transformations. Experimental results on SPEC95 benchmark are presented and some related issues are also discussed.

Key words: Alias, dummy aliasing, verification, optimization

1 Introduction

Aliasing occurs when two or more variables refer to the same storage location at the same program point. Alias analysis is critical for performing most optimizations correctly because all the ways a location, or the value of a variable, may (or must) be used or defined must be taken into account. Compile-time alias information is also important for program analyses, parallelization, verification, debugging and understanding.

The sources of aliases vary from language to language. *Intraprocedural* aliases occur due to pointers in languages like LISP, C, C++ or Fortran 90, union construct in C or `EQUIVALENCE` in Fortran. *Interprocedural* aliases are generally created by parameter passing and by access to global variables, which propagate intraprocedural aliases across procedures and introduce new aliases. Alias analysis can be classified by its formal characterization [16]: *may* or *must* information and *flow-sensitive* or *flow-insensitive* analysis. The may alias information indicates what may occur on some path through a flow graph, while the must information indicates what must occur on all paths through the flow graph. Flow-insensitive information is independent of the control flow encountered in a procedure, while flow-sensitive aliasing information depends on control flow. Furthermore, interprocedural alias analysis can be classified *context-insensitive* or *context-sensitive*. The context-insensitive approach cannot distinguish between different call sites of a procedure. The information about calling states is combined for all call sites and the resulting information about return states is returned at all return points. By contrast, the context-sensitive approach considers interprocedurally *realizable paths* [18] by maintaining the relationship between procedure calls and procedure returns.

In Fortran, parameters are passed by reference in such a way that, as long as the actual argument is associated with a named storage location, the called subprogram can change the value of the actual argument by assigning a value to the corresponding formal parameter. So new aliases can be created between formal parameters if a same actual argument is passed to two or more formal parameters, or between a formal parameter and a common variable if an actual argument is allocated in a common which is also visible in the called subprogram or other subprograms in the call chain below it. Restrictions on association of entities in Fortran 77 (Section 15.9.3.6 [4]) state that neither aliased formal parameters nor variables in the common blocks may become defined during execution of the called subprogram or other subprograms in the call chain. If these rules were enforced by compilers, aliases would be created only in few ways and be detectable exactly at compile-time. Mostly, they would not impact data dependence analysis and the optimizations based on it.

However, established programming practice often violates the Fortran 77 standard. Compilers should follow practice at least to some degree so as not to place the burden of alias analysis on the programmer. This can cause programs to produce results depending on optimization levels and programmers end up using different optimization levels for each module of an application. A contrived example of such aliasing is:

```
PROGRAM ALIAS                      SUBROUTINE SUB(X,Y)
INTEGER I,A(5)                     INTEGER I,X(5),Y
DO I = 1, 5                        DO I = 1, 5
   A(I) = 2                           X(I) = Y*X(I)
ENDDO                              ENDDO
CALL SUB(A,A(1))                   END
```

```
WRITE(*,*),A
END
```

The assignment to `X(1)` in the first loop iteration modifies `Y` which is loop invariant and stored in register. With the Sun WorkShop 6 FORTRAN 77 5.1 compiler, the output is 4 4 4 4 4 instead of 4 8 8 8 8 if the optimization level is greater than 2 and the modules are compiled separately. Some Fortran compilers such as OpenVMS, DEC Unix, Ultrix and AIX from IBM have an option to assert the presence of aliases between *dummy* arguments (Fortran terminology for formal parameters). If this option is selected, program semantics requires frequent recomputation on dummy arguments and common variables that insures correct results but optimizations are inhibited. By default, no aliases between dummy arguments and common variables exist. One called module can be compiled with the dummy aliasing assumption and the other modules with the opposite setting to improve performance. The no-aliases assumption should only be used for source programs that strictly obey Fortran 77 rules for associations of variables, but how can the programmer know for sure if there are aliases between dummy arguments and common variables or not? As mentioned in a study comparing the diagnostic capabilities of Fortran compilers [5], no compiler provides this standard violation check, one of the most common Fortran pitfalls. Only Forcheck, a commercial Fortran verifier spots violation on aliased scalar dummy arguments as run-time error.

The most difficult problem in Fortran alias analysis is to compute exactly the overlapping memory locations between arrays. Overlapping for arrays in an `EQUIVALENCE` statement is known at compile-time because the subscript expressions are integer constant expressions. But in the case of parameter passing, such as in:

```
SUBROUTINE SUB1
REAL A(100)
CALL SUB2(A(I),A(J),M1,M2)
END
```

```
SUBROUTINE SUB2(V1,V2,N1,N2)
REAL V1(N1), V2(N2)
END
```

the worst-case assumption is: the whole arrays `V1` and `V2` are aliased. But, if two intervals $[\mathrm{I}, \mathrm{I}+\mathrm{M1}-1]$ and $[\mathrm{J}, \mathrm{J}+\mathrm{M2}-1]$ can be proved disjoint, `V1` and `V2` are not aliased, and so optimizations can be applied in `SUB2`, if this is the only call to `SUB2`. Furthermore, `V1` and `V2` can overlap, but if all the written array elements in `SUB2` are proved not to be in the overlapping section, the restriction on association of entities in Fortran is not violated.

Our alias verification relies on three steps. Firstly, interprocedural aliases are computed for whole program. Secondly, this information is used to decide statically if the program violates the standard restrictions on alias or not. When the information is not known at compile-time, tests are added to check violations at execution time. Thirdly, to avoid false alarms, the impact of violation is studied. If the new data dependence arcs due to aliases are redundant with existing paths in the data dependence graph, the aliases have no impact on optimization. The first two steps are implemented in PIPS, *Paralléliseur*

Interprocédural de Programmes Scientifiques [15,14], the third step is used for case studies for this moment.

The paper is organized as follows. Section 2 discusses some related work. Section 3 describes the interprocedural alias propagation and Section 4 the interprocedural alias verification. Section 5 studies empirical results on alias checking on SPEC95 CFP benchmark, as well as the impact of these aliases on dependence graphs. Finally, Section 6 presents conclusions and ideas for future work.

2 Related work

A lot of work about alias analysis has been carried out during the past 25 years. Alias computation is usually divided into two parts [16]: *alias gatherer* and *alias propagator*. Since the sources of aliases vary from language to language, the alias gatherer is a language-specific component which is provided by the compiler front end. Meanwhile, the alias propagator is a common component that performs a data-flow analysis using the aliasing relations discovered by the alias gatherer to combine the aliasing information at join points in a procedure and to propagate it to where it is needed. The various alias analyses offer different trade-offs between the computational complexity and the accuracy.

Pointer alias analysis algorithms use varying degrees of flow-sensitivity, calling-context and alias representation and are empirically studied in many researches. However, pointer analysis is not in the scope of this paper. Alias analyses for programming languages without pointer such as Fortran 77 are discussed earlier in the literature [7,6,1,9,10,8]. These analyses deal with aliases arising from the renaming effects at call sites in languages with call-by-reference formal parameters and they are formulated as a data-flow analysis problem. The static call graph of a program is built and used to find the potential aliases at every procedure entry point. Banning presents in [6] an aliasing analysis that follows parameter binding chains through the program in a depth first fashion to compute all possible aliases. Cooper and Kennedy [10] improve the alias analysis computation time based on the fundamental insight that significant advantages can be achieved by separating the treatment of reference formal parameters from the treatment of global variables. Their algorithm requires $\mathrm{O(N^2 + NE)}$ steps, where N and E are the number of nodes and edges of the program's call graph, respectively. However, these methods treat arrays as atomic objects. This granularity is not fine enough and imposes too strict alias restriction on programs. A more sophisticated analysis of arrays might produce useful information about the offsets and patterns of overlap in a program. Such an analysis is implemented in `PTRAN` [1,13] compiler. When an array is involved in an alias, it determines, when possible, the difference in starting address between the aliased variables. These offsets are used in dependence analysis by linearization between aliased arrays. In [13], a precise interprocedural array analysis (`FIDA`) is used to avoid the conservative aliasing assumption and to improve the number of parallelized loops.

However, to our knowledge, no work has been done for the verification of restrictions on alias use, special for array variables. This verification is critical for code safety, debugging and maintainability (referential transparency) because allowing writing on aliased variables may result in unpredictable behaviors and make optimizations impossible. Actual compilers could detect the violations dynamically but run-time checks are still a overhead and they can catch only those violations that actually happen during a particular run. The objective of this project is to check the whole program, to generate a minimum number of tests by using precise alias information for both kinds of variables: scalar and array variables, and then to study the effect of alias violation on other program analyses and transformations, such as dependence analysis.

3 Interprocedural alias propagation

ANSI Fortran 77 standard [4] defines several ways to create aliases and they are mostly detectable exactly during compilation. The `EQUIVALENCE` statement is used to specify how two or more entities in the same program unit to share storages units. The effects of aliases created by `EQUIVALENCE` statements are purely local and statically determinable, as long as the equivalenced variables are not also in common storage. The `COMMON` statement associates different variables in different subprograms to the same storage. Determining the full effects of variables in common storage requires interprocedural analysis as for aliases created by parameter passing. When a procedure is called, an association is established between the actual arguments and the corresponding formal parameters in the called procedure. The formal parameter has the storage location of the actual argument by this invocation.

Formal parameters may become aliased in several ways. Two formal parameters are aliased if a same actual argument is passed to both of them. Also, if a global variable is used as actual argument and a variable aliased to the global variable is passed as another actual argument, the two corresponding formal parameters are aliased. In addition, formal parameters aliases can be passed through call chain, creating more aliased formal parameters. A global variable can only become aliased to a formal parameter in a routine in which it is visible and only by its being passed as an actual argument to that formal parameter.

To compute aliases, we introduce a *memory location naming* technique that allows a compact representation of address of variables.

Ram variable has an address (or a storage location) in some memory space that is linked to a common or to a procedure. An address is specified by an *area* and an *offset*. A variable in a common block is stored in an area whose name is global to the whole program. A local variable is stored in a static or dynamic area of the procedure which is the scope of the variable. Each variable is located in its area by its *offset*. The *size* of a variable in its area is the amount of memory space expressed in *numerical storage unit* or in *character storage unit*, according to Fortran standard. The number of storage units of a variable is defined by its type (integer, real, logical, double precision or complex, ...). The size of an array is the number of elements multiplied by the number of storage units of its element. An array has $\prod_{i=1}^{n} d_i$ elements where n is the number of dimensions of the 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.

Formal variable does not have its own address. But when it is associated with an actual argument, it will have the storage location of the actual argument. This actual argument in turn may be a formal parameter of the current caller and in this case, we have to go up the call site chain until we reach an actual argument which is a ram variable. So depending on the call path, different storage locations are associated with one formal variable.

When an array is passed in a `CALL` statement, the starting address of the formal array is computed using the base address of the actual array, and the subscript expression if an array element is passed. The *subscript value* expression of an array element determines the order of that element in the array. As Fortran language allocates array in column-major order, the subscript value of an array reference $A(s_1, s_2, \cdots, s_n)$ is $1 + \sum_{i=1}^{n}\left((s_i - l_i)\prod_{j=1}^{i-1} d_j\right)$. Note that $\prod_{j=1}^{0} d_j = 1$.

The basic idea for computing interprocedural aliases is to follow all the possible chains of parameter bindings at all call sites. The call graph is traversed in *invocation order* that process a procedure before all its callees, and alias information is accumulated incrementally from the main program. In our interprocedural tool PIPS, each analysis is performed only once on each procedure and produces a summary result that will be used later at call sites. For each procedure, storage locations of formal parameters of its callers have already been computed. This information is available in the database of PIPS and is used to compute addresses of the formal parameters of the current procedure. Our analysis is a context-sensitive analysis since it distinguishes among different calls of a procedures by storing the call path that produces the alias. The alias propagation algorithm works as follows:

Algorithm 1 *For each procedure* `P` *and each call site* `C` *to* `P`*:*

Let a_i be the i^{th} actual argument in the argument list of `C`, we compute all possible addresses of the corresponding formal parameter f_i which is a 3-tuple of area, offset and call chain. Normally, we have:

- $\mathrm{area}(f_i) = \mathrm{area}(a_i)$
- $\mathrm{offset}(f_i) = \mathrm{offset}(a_i) + \mathrm{subscript\_value}(a_i(s_1, .., s_n))$
- $\mathrm{call\_chain}(f_i) = \{C\}$, if the alias is created by only one call, or $\{\mathrm{call\_chain}(a_i), C\}$, if the alias is created through chain of calls.

By separating the treatment of formal parameters from the treatment of global variables, we have the following cases:

Case 1 *Alias between formal parameter and common variable:*

- Alias created by only one call: if a_i is a common variable and visible in `P` or in at least one callee (direct or indirect) of `P`, add a new address for f_i.
- Alias created through chain of calls: if a_i is a formal variable of the current caller and by retrieving already computed information for a_i in this caller, a_i has a common area that is visible in `P` or in at least one callee (direct and indirect) of `P`, add a new address for f_i.

Case 2 *Alias between formal parameters:*

- Alias created by only one call: if a_i is also passed to other formal parameters or there are other actual arguments that are in a `EQUIVALENCE` statement with a_i, we can divide the argument list into groups of same or equivalence arguments. A new address is added for each corresponding formal parameter, all parameters in a same group are in the same memory area. Specially, for the same argument case, although a_i can be a formal parameter of the current caller, we can use a special area `ALIAS_AREA_i`, where i is an unique counter for different group of same arguments.
- Alias created through chain of calls:
  - · If a_i and a_j are formal variables of the current caller and by retrieving already computed information for a_i and a_j, they have same area from two included call chains (to assure the alias happens), add a new address for each corresponding formal parameter.
  - · If a_i is a formal variable of the current caller and by retrieving already computed information for a_i, it has a same area with other actual argument that is a common variable, add a new address for f_i and a new address for the corresponding formal variable of the common variable with call chain equals to {C}

Then, the offsets of f_i are translated to the frame of `P` by using the global variables information and the bindings between actual and formal parameters. If one offset cannot be translated, we replace it by an unknown expression and the alias verification phase will treat this case differently, as it is discussed in the next section □

To illustrate the alias propagation, consider the following code:

```
1 PROGRAM MAIN          SUBROUTINE SUB1(V,N)        SUBROUTINE SUB2(V1,V2,L)
2 COMMON /COM/ W(50)    REAL V(N)                   COMMON /COM/ W(50)
3 REAL A(100),B(50)     READ *,M                    REAL V1(L),V2(L)
4 CALL SUB2(W,B,50)     IF (2*M.LE.N) THEN          DO I=1,L
5 CALL SUB1(A,100)        CALL SUB2(V,V(M+1),M)       V1(I) = V2(I)
6 END                   ENDIF                       ENDDO
                        END                         END
```

There are two call sites: `CALL SUB2(W,B,50)` from the main program (line 4) and `CALL SUB2(V,V(M+1),M)` from procedure `SUB1` (line 5) that can cause aliases. The first call makes the formal parameter `V1` aliased to the common variable `W`. The area of `V1` is `COMMON:COM`, a global name to the whole program. Its offset is `W`'s, which is equal to 0 (the offset is counted from 0, not 1). The call chain is only the procedure call to `SUB2` in the main program, denoted by `{(MAIN:#4)}`. In the second call site, the array `V` is associated to two different formal parameters `V1` and `V2`. The area and offset of `V1` and `V2` are computed from those of `V` and the subscript values in the procedure call. So we have: `area(V1)=area(V2)=ALIAS_AREA_1`; `offset(V1)=0`, `offset(V2)=4*L`; `call_chain(V1)=call_chain(V2)={(SUB1:#5)}`

So for each procedure, the propagation phase produces a compact and precise alias information that is used in the next phase to check the restriction on aliased variables.

4 Interprocedural alias verification

We enforce the Fortran standard about the restrictions on association of entities (Section 15.9.3.6 [4]). In Section 17.1.3 [4], two variables, array elements or substrings are associated if their storage sequences are associated. The association between actual and formal argument implies association of storage sequences when the actual argument is the name of a variable, array element, array, or substring.

In fact, there is no mention of the association of two arrays. Arrays can be treated as units, that is, if two formal array parameters or one formal array parameter and one global array variable are aliased, assignment on any element of any array is forbidden. This granularity is not fine enough because we may not write on the really aliased elements, which identify the same datum. So we need a more exact definition of the so called writing violation on aliased variables in a subprogram.

Definition 4.1 *A call chain causes an alias violation of formal parameter (of common variable) in a subprogram if following this call chain, there exists two different formal parameters (a common variable and a formal parameter) sharing storage units that are defined during execution of the subprogram.*

One important assumption for alias verification is related to the absence of bound violations in the program. Note that if the storage sequences of concerning variables are disjoint, they will share no storage unit. Therefore, we have:

Lemma 4.2 *In a subprogram, if the storage sequences of all formal parameters (if the storage sequences of all formal parameters and those of all common variables) allocated by the execution of a call chain are disjoint, there is no alias violation of formal parameter (of common variable) caused by this call chain.*

The next two lemmas show the condition of alias violation that depends on writing or not on the overlapping section of two aliased variables.

Lemma 4.3 *In a subprogram, if the defined sequence of every formal parameter (every common variable) is disjoint with the storage sequences of all other formal parameters allocated by the execution of a call chain, there is no alias violation of formal parameter (of common variable) caused by this call chain.*

Lemma 4.4 *In a subprogram, if there exists a formal parameter (a common variable) whose defined sequence is not disjoint with the storage sequence of another formal parameter allocated by the execution of a call chain, there is an alias violation of formal parameter (of common variable) caused by this call chain.*

The alias verification on each procedure of the program is defined by these three lemmas. Procedures with no formal parameter and procedures with formal parameters but no possible alias (calculated by the propagation phase) are excluded. To compute the defined sequence of a variable in a procedure, we need information about write effects of statements on this variable. In general, the exact effect is only known at elementary statements such as assignments, because of different control paths. However, to reduce the number of analyzed pairs of aliased variables, we can use the *summary effect* of a procedure on its formal and common variables. This is an over approximation of the exact effects and by using this information, if a variable is never defined during the procedure execution, we do not have to treat it.

By a call chain, two formal parameters or one formal parameter and one global variable maybe aliased if and only if they are in the same area. So for each may be aliased pair where at least one of them is present in the summary write effects of the procedure, the following tests are performed:

Algorithm 2 *Check for alias violation between two variables:*

Case 1. If the offsets of both variables are known, depending on the type of variables: scalar or array, different check cases are performed. As a matter of space, we only describe the most complicated case: both variables are array variables. Let o_1 and s_1, o_2 and s_2 be the offset and size of the array a_1 and a_2, respectively.

- If $\mathrm{o}_2 + \mathrm{s}_2 \leq \mathrm{o}_1$ or $\mathrm{o}_1 + \mathrm{s}_1 \leq \mathrm{o}_2$ is evaluated true, according to Lemma 4.2, there is no alias violation.
- Else, without loss of generality, suppose that some elements of a_1 are defined in the procedure. For each statement that has write effect on a_1, let r_1 be the subscript value that determines the position of the defined array element.
  - · If $\mathrm{r}_1 < \mathrm{o}_2$ or $\mathrm{r}_1 \geq \mathrm{o}_2 + \mathrm{s}_2$ is evaluated true, according to Lemma 4.3, there is no alias violation.
  - · Else, according to Lemma 4.4, insert before the current statement: IF (flag_1 .AND. ... flag_n .AND. $(\mathrm{o}_2 \leq \mathrm{r}_1)$ .AND. $(\mathrm{r}_1 < \mathrm{o}_2 + \mathrm{s}_2)$) STOP message where flag_i is the flag inserted before the call site i in the call chain to mark if the execution reaches the current statement or not. If one clause $\mathrm{o}_2 \leq \mathrm{r}_1$ or $\mathrm{r}_1 < \mathrm{o}_2 + \mathrm{s}_2$ is known to be true at compile-time, it can be

removed from the test condition.

Case 2. If at least one offset is unknown because it cannot be translated to the frame of the called procedure in the propagation phase, the simplest solution is to generate directly checks for these variables. We can insert before each statement that defines a variable, for example a_1, the following procedure call: CALL ALIAS_CHECK$(\mathrm{message}, a_1, r_1, a_2, s_2)$ where r_1 is the subscript value of the written element of array a_1, s_2 is the size of the second variable a_2. ALIAS_CHECK$(\mathrm{char} * \mathrm{message}, \mathrm{void} * p_1, \mathrm{int} * i_1, \mathrm{void} * p_2, \mathrm{int} * i_2)$ is a C function that takes the base address of two variables, the referenced element of one variable and the size of the other to check if there is alias violation on the referenced element or not. This check is expensive because no static analysis is performed. We improve the algorithm by trying to check for alias violation in the frame of the direct caller of the current procedure if possible. The process is:

- Compute the offset of each variable in the caller's frame:
  - If the variable is a common variable, its offset in the common block does not change and is returned if the variable is visible in the caller. Otherwise, an unknown expression is returned.
  - If the variable is a formal parameter, we return the offset of the corresponding actual argument plus the subscript value of this actual argument. In fact, this is the value before the translation step in the propagation phase. If the offset of the actual argument is unknown because the actual argument is a formal parameter of the caller and information is lost somewhere earlier in the call chain, we return an unknown expression.
- If both offsets can be computed in the caller's frame, we repeat Case 1, but each time the size or the subscript value is needed, we have to translate it from the current procedure's frame to the caller's frame, by using the global variables information and the bindings between actual and formal parameters. Tests inserted before each statement defining a variable in the current procedure are of form:

  IF (flag_1 .AND. ... .AND. flag_n) STOP message

  and test inserted before the direct call site is of form:

  IF $((o_2 \leq r_1)$ .AND. $(r_1 < o_2 + s_2))$ $\mathrm{flag}_n =$ .TRUE.

  where $\mathrm{flag}_1, \cdots, \mathrm{flag}_{n-1}$ are flags set before the $n-1$ higher call sites in the call chain.
- If the translation is not possible, or one offset is unknown, we have to return to the direct check version. □

In Section 3 example, no alias violations are caused by `CALL SUB2(V,V(M+1),M)` because there is no intersection between [0:4*L-1] and [4*L:8*L-1]. However, there are alias violations when writing on `V1(I)` that is aliased with an element in `W`. The code is instrumented as follows:

```
PROGRAM MAIN                SUBROUTINE SUB2(V1,V2,L)
COMMON /COM/ W(50)          COMMON /COM/ W(50)
```

```
REAL A(100),B(50)            REAL V1(L),V2(L)
COMMON /FLAGS/ FLAG(1)       COMMON /FLAGS/ FLAG(1)
LOGICAL FLAG(1)              LOGICAL FLAG(1)
FLAG(1) = .TRUE.             DO I=1,L
CALL SUB2(W,B,50)             IF (I.GE.1.AND.I.LE.50.AND.FLAG(1)) STOP
CALL SUB1(A,100)              "Alias violation on V1,V2, line 4 of MAIN"
END                           V1(I) = V2(I)
                             ENDDO
                             END
```

These alias checks are expensive because they are left inside loop. It can be optimized by safely applying code hoisting since our instrumented code does not violate the standard anymore.

5 Experimental Results

We use the SPEC CFP95 benchmark [11] that contains 10 applications written in Fortran. They are scientific benchmarks with floating point arithmetic and many of them have been derived from publicly available application programs. We are not interested in *tomcatv* which is a single procedure program. For the other 9 benchmarks, the number of routines in a program varies from 6 to 105, the number of procedure and function calls is in the range from 5 to 243. Table 1 summarizes relevant information for all benchmark in SPEC CFP95.

Some array declarations conflict with alias analysis. The *assumed-size array* declaration (Section 5.1.2 [4]) with an asterisk as the last dimension and array declaration with a final dimension specified as 1 (*ugly assumed-size array* in GNU terminology) prevent the compiler to know the size of the array in the called routine. These kinds of declaration may cause spurious array bound violations, inhibit program analyses such as array bound checking, used before set analysis, program debugging, etc. Unlike other compilers that ignore these assumed-size arrays, we deal with this by applying *array resizing* [3] to five benchmarks, *applu, turb3d, apsi, fpppp* and *wave5*, where assumed-size declarations are used. The array resizing phase tries to find out automatically the proper upper bound for the one and assumed-size array declarations. It uses the relationship between actual and formal arguments from parameter-passing rules: the size of the formal argument array must not exceed the size of the actual argument array. New array declarations in the called procedure are computed with respect to the declarations in the calling procedures. Codes are instrumented to pass the array descriptors corresponding to each procedure call. 100% of ugly and assumed-size arrays are resized.

The numbers of inserted flags, tests, checks and the compilation times in second for alias verification are reported in Table 1. Codes with generated checks are then compiled and executed using the standard input data for SPEC95 benchmarks. 6 out of 10 benchmarks violate alias rules: *hydro2d, mgrid, applu, turb3d, apsi* and *wave5*. As *tomcatv* has only one routine, there are certainly no aliases for this benchmark. *swim* and *su2cor* are proved to be

Table 1
SPEC95 CFP: number of lines, routines, calls; number of inserted flags, tests, checks and compilation times; alias violation results

Prog.	Lines	Rou.	Calls	Flags	Tests	Checks	Time	Violat.	AR	DG
tomcatv	190	1	0	0	0	0	0	No		
swim	429	6	5	0	0	0	1	No		
su2cor	2332	35	166	0	0	0	14	No		
hydro2d	4292	42	98	8	12	0	9	Yes		*
mgrid	484	12	23	0	0	10	1	Yes		*
applu	3868	16	27	6	6	0	8	Yes		*
turb3d	2101	23	111	60	156	13	10	Yes		*
apsi	7361	96	190	23	194	2945	92	Yes	*	
fpppp	2784	38	27	0	0	250	30	No		
wave5	7764	105	243	36	334	495	98	Yes	*	

free of dummy aliased variables by our analysis. *fpppp* is instrumented with 250 checks, generating 2% execution slowdown and has no alias violations for its standard input.

Two kinds of alias violations appear in 6 benchmarks. The first category exists in *apsi* and *wave5*: assumed-size arrays are resized with new dimensions that are too large with respect to actual array accesses. Consider the illustrating example:

```
REAL  WORK(1000)                              SUBROUTINE RUN(X,Y,Z,L)
CALL RUN(WORK,WORK(L+1),WORK(2*L+1),L)        DIMENSION X(*),Y(*),Z(*)
```

Because Fortran uses the column-major scheme of storing arrays, the address in memory of a given array reference is calculated from the base address of the array and the subscript value (Section 3), which do not involve the last dimension. The upper bound can be left unspecified, and the compiler does not know the logical size of the formal array in the called routine. The physical size in the called routine is the size the array has in the caller. We only have to ensure that references to elements do not go past the end of the actual array. So with array resizing, we infer new declarations: `DIMENSION X(1000),Y(1000-L),Z(1000-2*L)`. These declarations are correct with respect to the standard but they may be too large for actual array accesses in the called routines. The intersection between the storage sequence of array `X`, computed from the array size, and the defined sequence of array `Y` or `Z` are not empty. So false alias alarms are raised because the actual accesses of array `X` are not taken into account. To cope with this problem, we could use another approach for array resizing based on array region [3]. It gives information about the set of array elements accessed during the code execution. In this case, array declarations are sharper and reduce greatly the number of alias violations. However, it is not always possible to compute the array region, due to non-linear expressions, indirection arrays, the lack of structure of programs... A good programming practice is to pass disjoint array sections to

```
    SUBROUTINE FCT(UNEW,UTRA,UOLD)
    PARAMETER (MP = 402, NP = 160)
    DIMENSION UNEW(MP,NP), UTRA(MP,NP), UOLD(MP,NP)
    DO 100 J = 1, NQ
       DO 100 I = 1, MQ
S1           DZ1(I,J) = UOLD(I+1,J) − UOLD(I,J)
S2           AZ1(I,J) = UTRA(I+1,J) − UTRA(I,J)
100 CONTINUE
    DO 200 J = 1, NQ
       DO 200 I = 1, MQ
S3           UTRA(I,J) = UTRA(I,J) + DZ1(I,J) − DZ1(I−1,J)
200 CONTINUE
    DO 400 J = 1, NQ
       DO 400 I = 1, MQ
S4           UNEW(I,J) = UTRA(I,J) − AZ1(I,J) − AZ1(I−1,J)
400 CONTINUE
    END
```

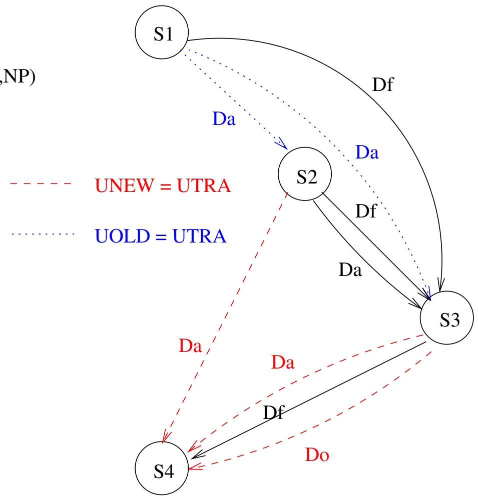

Fig. 1. Dependence graph of a code fragment from hydro2d

the called routine, i.e `DIMENSION X(L),Y(L),Z(L)`. This motivates us to use the alias information to derive proper array declarations, a problem not only in Fortran but also in other programming languages such as C and `MATLAB` [3].

The second category of alias violation includes real violations because two scalar variables or two array elements have exactly the same memory location and one of them is written; but the legal schedules are not changed by these aliases. As explained in [2], a transformation preserves the semantics of the program if the control and data dependences are respected. Any ordering-based optimization that does not change the dependences of a program is guaranteed not to change the results of the program. We take as example a piece of code from *hydro2d* where calls to subroutine `FCT` make two formal parameters `UNEW` and `UTRA` aliased. As shown in Figure 1, the new data dependence arcs (dashed arcs) created by the alias between `UNEW` and `UTRA` are redundant with existing paths in the dependence graph. So we can prove that all ordering-based optimizations for `FCT` are safe. If `UOLD` and `UTRA` were aliased, an anti-dependence arc (dotted arc) would be added between `S1` and `S2`, creating a new scheduling constraint. So the alias between `UOLD` and `UTRA` would cause a real alarm.

To sum up, although the standard specification is violated by some programs, we did not find any aliasing problem in the SPEC95 CFP benchmarks. This is no surprise since these benchmarks are well-debugged.

6 Conclusion

Our alias analysis is flow- and context- sensitive and gives efficient and precise alias information. This information lets us avoid the worst-case assumption about aliases and do program analyses more exactly. For example, without alias information, the used before set analysis can give false results because we risk to initialize some variables by aliasing without knowing that. Our analysis can be applied to other programming languages with call-by-reference mechanism.

We developed algorithms to check for violation of aliasing rules in Fortran, an option which is missing in most compilers. Violations are detected not only for scalar but also for array variables. This standard checking is useful to debug code, to help programmer to correct errors in order to gain performance by applying optimizations safely. Once alias checks are generated, the instrumented code will respect the standard about aliasing, other techniques such as code hoisting, partial redundancy elimination, induction variable optimization can be used to optimize the generated code. The correctness and completeness proofs of our algorithms can be found in the technical report [17].

Our experimental results show that the SPEC95 CFP programs do not suffer from effective aliasing errors. They also show that accurate alias detection depends on array resizing and dependence graph analysis, and that alias analysis cannot be fully evaluated with standard benchmark. Experiments on less debugged codes are necessary. We applied the analysis to a large scale industrial application, about 100.000 lines of code, and found several potential bugs.

An important perspective is that the alias information itself can be used to resize array, also an array declaration problem in other programming language such as C, `MATLAB` and `APL`. Another future work is to study automatically the effect of aliases on the program dependence graph. When aliases create more dependence arcs, the analysis should be able to tell if these arcs modify the existing dependence graph, so the aliases have impacts on optimization or not. The PIPS software and documentation as well as our alias checking implementations are available on *http://www.cri.ensmp.fr/pips*.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2002.

[3] C. Ancourt and T. V. N. Nguyen. Array resizing for code debugging, maintenance and reuse. In *ACM SIGLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering - PASTE01*, pages 32–37, Snowbird, Utah, USA, June 2001.

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

[5] J. Appleyard. Comparing Fortran compilers. *ACM SIGPLAN - Fortran Forum*, 20(1):6–10, 2001.

[6] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[7] J. M. Barth. An interprocedural data flow analysis algorithm. In *Symposium on Principles of Programming Languages*, pages 119–131, 1977.

[8] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[9] K. D. Cooper. Analyzing aliases of reference formal parameter. In *Symposium on Principles of Programming Languages*, pages 281–290, 1984.

[10] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Symposium on Principles of Programming Languages*, pages 49–59, 1989.

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

[12] W. Gellerich and E. Plodereder. Parameter-induced aliasing in Ada. In *Ada-Europe*, volume 2043 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2001.

[13] M. Hind, M. Burke, P. Carini, and S. Midkiff. An Empirical Study of Precise Interprocedural Array Analysis. *Scientific Programming*, 3(3):255–271, 1994.

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

[15] 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.

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

[17] T. V. N. Nguyen. Program analyses for software verification. Technical Report E/250/CRI, Ecole des Mines de Paris, 77305 Fontainebleau Cedex, France, 2002.

[18] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages*, pages 49–61, San Francisco,CA, USA, 1995.

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