Journal of Universal Computer Science, vol. 9, no. 3 (2003), 270-297
submitted: 30/9/02, accepted: 27/1/03, appeared: 28/3/03 © J.UCS

Alias Verification for Fortran Code Optimization

Thi Viet Nga Nguyen
(Ecole des Mines de Paris, France
nguyen@cri.ensmp.fr)

François Irigoin
(Ecole des Mines de Paris, France
irigoin@cri.ensmp.fr)

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 produce incorrect answers, or behave unpredictably. The results may depend on the compiler and the optimization level.

To guarantee the standard conformance of programs in order to make program analyses exact and program optimizations safe, precise alias information, i.e the determination of overlaps among arrays is studied in this paper. Static analyses and code instrumentation are used to find all violations of the aliasing rules 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 obtain compact representation and to enhance the precision of alias analysis. Modifications on the dependence graph created by aliasing are also studied to show the impact of aliases on the correctness of some program optimizing transformations. Experimental results on SPEC95 CFP benchmark are presented and some related issues are also discussed.

Key Words: alias, dummy aliasing, verification, optimization

Category: D.2.4 Software Engineering - Software/Program Verification [Assertion checkers] D.2.5 Software Engineering - Testing and Debugging [Debugging aids, Symbolic execution] D.3.4 Programming Languages - Processors [Compilers, 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 key to data dependence analysis and hence is also important for program analyses, transformation, parallelization, verification, debugging and understanding. For example, a good precision of alias information increases the degree of instruction-level parallelism by a factor of 4 to 8 times with respect to the conservative alias information [Wal91].

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 constructs in C or `EQUIVALENCE` in Fortran. *Interprocedural* aliases are 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 [Muc97]: *may* versus *must* information and *flow-sensitive* versus *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 among 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* [RHS95] by maintaining the relationship between procedure calls and procedure returns.

In Fortran, parameters are passed by reference and, as long as the actual argument is associated to 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 are created among formal parameters if the 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 block 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 [ANS83]) 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 a few ways and be pin-pointed at compile-time. Mostly, they would not impact on data dependence analysis and the optimizations based on it, nor on maintenance and re-use.

However, established programming practice often violates the Fortran 77 standard [ZC90]. 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 to end up using different optimization levels for each module of an application. A contrived example of such aliasing is given in Figure 1.

The assignment to `X(1)` in the first loop iteration modifies `Y` which is loop invariant and stored in a 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

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

```
SUBROUTINE SUB(X,Y)
INTEGER I,X(5),Y
DO I = 1, 5
  X(I) = Y*X(I)
ENDDO
END
```

Figure 1: *Formal parameter aliasing example*

greater than 2 and if 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 among *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-alias 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 [App01], no compiler provides this standard violation check, one of the most common Fortran pitfalls. Only Forcheck (*http://www.forcheck.nl*), a commercial Fortran verifier, spots violation only on aliased scalar dummy arguments and only at run-time.

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. In parameter passing cases, such as in Figure 2, 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.

Programmers sometimes pass overlapping array regions purposefully, thinking that it will be safe because they know in which order the subroutine deals with the data. However, bugs related to this programming practice are very difficult to track, since compilers can reorder the subroutine's code, which results in incorrect output.

```
SUBROUTINE SUB1                     SUBROUTINE SUB2(V1,V2,N1,N2)
REAL A(100)                         REAL V1(N1), V2(N2)
READ *,I,J,M1,M2                    ...
CALL SUB2(A(I),A(J),M1,M2)          END
END
```

Figure 2: *Array overlapping example*

Our alias verification is broken down into three steps. Firstly, interprocedural aliases are computed for the whole program. Then, this information is used to decide statically if the program violates the standard restrictions on alias or not. There are three cases: no possibility of an alias, certainty of an alias, and possibility of an alias. In the last case, tests are inserted to check violations at execution time. Finally, the impact of alias violation on program optimizations 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. These three steps are implemented in PIPS (*Paralléliseur Interprocédural de Programmes Scientifiques*), a research compiler developed at Ecole des Mines de Paris [IJT91, Iri93].

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 discusses the impact of alias violations and points out some optimizations whose correctness depends crucially on alias information. Section 6 studies empirical results on alias checking on SPEC95 CFP benchmark. Finally, Section 7 presents conclusions and ideas for future work.

2 Related Work

A lot of work about alias analysis has been carried out over the past 25 years. Alias computation is usually divided into two parts [Muc97]: *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 which 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 computational complexity and accuracy.

Pointer alias analysis algorithms use varying degrees of flow-sensitivity, calling - context and alias representation and have empirically been studied in a lot of research papers [Cou86, LR92, CBC93, Deu94, BCCH95, Ruf95, WL95, ZRL96, Ste96, SH97, HBCC99, LH01, LPH01, CRL01, DLFR01, GLS01, HT01a, HT01b, HP01, Hin01, MDCE01, RLS+01]. Ruf [Ruf95] compares the precision of a simple, efficient, context-insensitive points-to analysis for the C programming language with that of a maximally context-sensitive version of the same analysis. He found that context-insensitive techniques can be implemented quite efficiently, and can produce information that is surprisingly precise, as well as useful in improving the efficiency of context-sensitive techniques. Hind et al. [HP01] provide the comparison of the effectiveness (compile-time efficiency and precision) of six pointer analysis algorithms: four flow-insensitive, one flow-sensitive, one flow-insensitive but uses precomputed flow-sensitive information. The results of their paper suggest that the Steensgaard's flow-insensitive algorithm [Ste96] computing one solution set for the entire program and using a union-find data structure to avoid iteration is usable in production compilers. If better precision is required, the analyses of Andersen [And94] or Burke et al. [BCCH95] can be used. Further precision may be obtained by the interprocedural analyses of Choi et al. [CBC93] and Hind et al. [HBCC99]. The efficiency of these analyses is improved by identifying realizable execution paths and by identifying different instances of dynamically allocated objects created along different call paths. Extensions of points-to analysis algorithms for object-oriented languages such as Java and C++ are studied in [LPH01, CRL01, WFPS02].

However, pointer analysis is not really within the scope of this paper. Alias analyses for programming languages without pointers such as Fortran 77 are discussed earlier in the literature [Bar77, Ban79, ABC+88, Coo84, CK89, CK88, MW93]. These analyses deal with aliases arising from the renaming effects at call sites in languages with call-by-reference parameter passing. 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 [Ban79] presents an aliasing analysis that follows parameter binding chains through the program in a depth first fashion to compute all possible aliases. Cooper and Kennedy [CK89] 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 $\mathtt{O}(\mathtt{N}^2 + \mathtt{NE})$ steps, where `N` and `E` are the number of nodes and edges of the program's call graph, respectively. An alias detection analysis is implemented by Mayer and Wolfe in the Nascent compiler [MW93], by improving Cooper and Kennedy's algorithm. The number and cardinality of maybe aliased variable sets as well as their relationship with the number of global variables and formal parameters per procedure on PerfectClub and Riceps benchmarks are reported in the paper. They also study the interaction of dependence analysis and aliasing by considering the parallelization of loops with formal array arguments. However, these methods treat arrays as atomic objects. This granularity is not fine enough and imposes alias restrictions too strict to be useful. Moreover, the case where an actual array element can be associated to a formal array parameter seems omitted in [MW93], so aliases between two formal array parameters may go undetected. 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 the `PTRAN` compiler [ABC$^+$88, HBCM92, HBCM94]. 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 [HBCM94], a precise interprocedural array analysis (`FIDA`) is used to avoid the conservative aliasing assumption and to improve the number of parallelized loops.

The parameter-induced aliasing in Ada has recently been studied in [GP01]. They implement the alias analysis developed by [Coo84] and examine manually the introduced alias pairs. The experimental results reveal a rare occurrence of aliasing that may explain the reliability of Ada programs.

However, to our knowledge, no work has been done on the verification of restrictions on alias use, especially 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. Current compilers could detect the violations dynamically but run-time checks are still an overhead and they can catch only those violations that actually happen during a particular run. The objective of this work 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, and then to study the impact of alias violation on other program optimizations.

3 Interprocedural Alias Propagation

ANSI Fortran 77 standard [ANS83] defines several ways to create aliases and most are detectable exactly during compilation. The `EQUIVALENCE` statement is used to specify how two or more entities in the same program unit do 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 for this invocation.

Formal parameters may become aliased in several ways. Two formal parameters are aliased if the same actual argument is passed to both of them. Also, if a global variable is used as an 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 parameter 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.

3.1 Storage Sequence Association

In our approach, arrays are not treated as atomic variables. To compute the overlap between arrays, the following definitions are needed to describe relationships that exist among variables, based on Section 17.1 [ANS83].

Definition 3.1 *A storage sequence of a variable is a sequence of storage units that the variable represents. The size of a storage sequence is the number of storage units in the storage sequence.*

For example, an integer type scalar variable or array element has a storage sequence of one storage unit; a double precision type has a storage sequence of two storage units. An array has a storage sequence consisting of the storage sequences of the array elements in the order determined by the array element ordering. The storage sequence of a common block consists of the storage sequences of all variables in the block in the order of their appearance.

Definition 3.2 *Two storage sequences are associated if they share some storage units.*

Definition 3.3 *In a procedure, two formal parameters are aliased by a call chain if and only if their corresponding storage sequences for this call chain are associated.*

Definition 3.4 *In a procedure, a formal parameter and a common variable are aliased by a call chain if and only if their corresponding storage sequences for this call chain are associated.*

3.2 Memory Location Naming

To decide when two storage sequences can be associated, we introduce a *memory location naming* technique that allows a compact representation of variable addresses. It provides all information required to locate a variable in the memory: the area, offset and call chain.

3.2.1 Actual Variable

Actual variables are variables that have a known address in some memory space. They include common and local variables. Since the storage sequences of two common blocks in a same program unit cannot share storage units, a unique name is given to each common block. This name is called an *area*. A local variable is stored in a static or dynamic area of the procedure where the variable is declared.

The *offset* of a variable is used to locate the variable in its area. The size of the storage sequence of this variable defines the offset of the next variable in the same area. If several variables are involved in an `EQUIVALENCE`, their offsets in the area are computed with respect to the association of the storage sequences. Since the subscript expressions in an `EQUIVALENCE` statement are known at compile-time, the computation of this information is straightforward.

3.2.2 Formal Variable

A *formal variable* does not have its own address. But when it is associated to an actual argument, it will have the address of the actual argument. This actual argument in turn may be a formal variable of the current caller, and in this case, we have to go up the call chain until we reach an actual argument that is an actual variable. So, depending on the call path, different addresses may be associated to one formal variable.

When an array is passed in a `CALL` statement, the starting address of the formal array argument is computed using the offset of the actual array argument, 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 $1 + \sum_{i=1}^{n} \left((s_i - l_i) \prod_{j=1}^{i-1} d_j \right)$ 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. l_i and u_i are respectively the corresponding lower and upper declaration bounds. Note that $\prod_{j=1}^{0} d_j = 1$. The size of an array element is the number of storage units required to store this element.

Definition 3.5 *The relationship between the offsets of the formal and actual array arguments is expressed by the equation:*

offset(formal_array) = offset(actual_array) + element_size(subscript_value-1)*

One important assumption for alias analysis is the absence of bound violations in the program, i.e to guarantee that all subscript values are bounded. The next two theorems specify the necessary conditions for an alias between a formal parameter and a common variable or two formal parameters in a procedure to exist. Their proofs are rather trivial because they are theorems of exclusion to enumerate all possible cases of aliasing. The treatments of formal parameters and common variables are separated.

Theorem 3.1 *In a procedure, if a formal parameter and a common variable are aliased by one call chain, then one of the following conditions must be satisfied:*

1. *The corresponding actual argument is an actual variable whose area is that of the common variable. This alias is created by only one call site, regardless of the call chain.*
2. *The corresponding actual argument is a formal variable of the caller and following the call chain, it is located in the area of the common variable.*

Proof. We prove that if the corresponding storage sequences of the formal parameter and the common variable are associated, either Condition 1 or 2 must be satisfied.

If the corresponding actual argument is an actual variable, it cannot be a local variable of the caller because a local variable of one procedure cannot be associated to a common variable of the other. This actual variable must be a common variable, and since the two common variables share the same storage units, they must be in the same common block, declared in two different procedures. So they are in the same area. The call chain has only one element, that is the direct call site.

If the corresponding actual argument is a formal variable of the caller, following the call chain, we reach an actual variable. Reasoning as in the first case, we come to the same conclusion.

Theorem 3.2 *In a procedure, if two formal parameters are aliased by a call chain, then one of the following conditions must be satisfied:*

1. *The same actual argument is passed to them. This alias is created by only one call site.*
2. *The corresponding actual arguments are in an* `EQUIVALENCE` *statement of the caller. This alias is also created by only one call site.*
3. *One actual argument is a common variable and the other actual argument is a formal variable of the caller, and following the call chain, the address of this formal variable has the same area as the common variable.*
4. *The corresponding actual arguments are formal variables of the caller and following the call chain, they have addresses which share the same area.*

Proof. Supposing that two formal parameters are aliased by a call chain, we have three possibilities:

1. Two corresponding actual arguments are actual variables of the caller. If they are different variables, the only way to associate their storage sequences within a program unit is to declare them in an `EQUIVALENCE` statement. So in this case, either Condition 1 or Condition 2 must be satisfied.

2. One actual argument is a formal variable and the other is an actual variable of the caller. This actual variable cannot be a local variable because the storage sequence of a local variable cannot be shared with that of another variable of another procedure. So the actual variable must be a common variable. Following the call chain, the formal variable will reach an actual variable. This actual variable in turn must be a common variable. And since the storage sequence of two common variables in two different subprograms are associated, they must be in the same common block, and therefore the same area. Condition 3 is satisfied.
3. Both actual arguments are formal variables of the caller. The simplest case is when they are the same variables (Condition 1) and the alias is created by only one call, which is the direct caller. These actual arguments cannot be equivalenced by the caller because they are formal parameters. If not, following the call chain, we will reach, not necessarily at the same time, the two corresponding actual variables.
   We have two sub-cases:
   - We reach the two actual variables at the same call site. As in the first case, these actual variables are the same variables or in an `EQUIVALENCE` statement of the current call site procedure. Their addresses must share the same area.
   - If one actual variable is reached before the other, which means that they are actual variables of two different subprograms and since their storage sequences are associated, they must be common variables in the same common block and therefore in the same area.

   Condition 4 is satisfied by these sub-cases.

The theorem is proved by enumeration.

3.3 Alias Propagation Algorithm

The basic idea for computing interprocedural aliases is to follow all the possible chains of parameter bindings at all call sites. The acyclic call graph is traversed in the *invocation order* that process a procedure before all its callees, and alias information is accumulated incrementally from the main program. In our interprocedural compiler PIPS, each analysis is performed only once on each procedure and produces a summary result that is used later at call sites. For each procedure, symbolic addresses 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.

Algorithm 3.1 shows the interprocedural alias propagation, based on Theorem 3.1 and Theorem 3.2. This propagation computes in fact an over approximated set of aliases in a procedure. Our analysis is context-sensitive since it distinguishes the different calls of a procedures by storing the call path that produces the alias.

Algorithm 3.1

procedure *Interprocedural_Alias_Propagation (p)*
  p : current procedure
begin
  for each *call site c to p*
    q := corresponding_caller(c)
    partition the actual argument list to groups of same and equival. variables
    for each *group g*
      for each *actual argument a in g*
        f = corresponding_formal_argument(a, c)
        if *g is a group of same variables* **then**
          area(f) = ALIAS_AREA_g
        else
          area(f) = area(a)
        endif
        call_chain(f) $= \{c\}$
        offset(f) = compute_and_translate_offset(a, c, p, q)
      endfor
    endfor
    for each *actual argument a in the argument list*
      if *a is a common variable* **then**
        f = corresponding_formal_argument(a, c)
        area(f) = area(a)
        call_chain(f) $= \{c\}$
        offset(f) = compute_and_translate_offset(a, c, p, q)
      endif
      if *a is a formal variable and one of its addresses has a common area or the same area as an address of another actual argument that is a formal variable* **then**
        f = corresponding_formal_argument(a, c)
        area(f) = corresponding_area(a)
        call_chain(f) = corresponding_call_chain(a) $\cup \{c\}$
        offset(f) = compute_and_translate_offset(a, c, p, q)
      endif
    endfor
  endfor
end

By dividing the list of actual arguments into groups of identical or equivalenced variables, all formal parameters in a same group have the same area. If the same variable is passed to different formal parameters, a special area called `ALIAS_AREA_g` and a call chain of one element are associated to the formal parameters (Condition 1 and 2 of Theorem 3.2). The advantage is that although the actual argument can be a formal parameter of the caller, we do not need to keep the whole call chain in which an actual variable is reached. If the actual argument a is a common variable, an alias may exist under Condition 1 of Theorem 3.1 if a is visible in the called procedure and Condition 3 of Theorem 3.2 if there exists another actual argument that is a formal variable of the caller and one of its addresses has the same area as a.

If a is a formal variable and one of its addresses has a common area which is visible in p, there may be an alias according to Condition 2 of Theorem 3.1. If this address has the same area as another common actual argument, an alias may exist under Condition 3 of Theorem 3.2. In the last case, if this address has the same area as an address of another actual argument that is a formal variable, an alias may happen following Condition 4 of Theorem 3.2.

The *compute_and_translate_offset* function in Algorithm 3.2 computes the offset of the formal parameter with respect to that of the actual argument and the subscript value. The offset is translated to the frame of the procedure p by using global variables information and the bindings between actual and formal parameters. If the translation is not possible, an unknown expression is returned and the alias verification phase treats this case differently, as discussed in the next section.

Algorithm 3.2
function *compute_and_translate_offset*(a, c, p, q)
begin
    off = *offset(a)* + *subscript_value*(a, c)
    if *off can be translated to the frame of* p **then**
        off = *translate_to_callee_frame(off,*$c, p, q)$
    else
        off = *unknown*
    endif
    return *off*
end

3.4 Alias Propagation Example

Our example to illustrate the alias propagation is given in Figure 3. Each formal parameter of subroutine `SUB2` has two addresses corresponding to two different call paths: `{MAIN:CALL SUB2(W,B,50)}` and `{MAIN:CALL SUB1(A,100), SUB1:CALL SUB2(V,V(M+1),M)}`.

There is no possible alias for `SUB1` but the two calls to `SUB2` may cause aliases in this module. In the first call, the common variable `W` that is visible in `SUB2` is passed as an actual argument to `V1`. An address is added for `V1` by the alias propagation. The area of this address is that of `W`, the offset is computed from zero and the call chain is this call site.

In the second call, array `V` is associated to two different formal parameters `V1` and `V2` by the second call. The offset of `V1` and `V2` is computed with respect to that of `V` and the subscript value in the actual argument list. A special area is used and we only need to keep the direct caller of `SUB2`. The element size of real variable is 4.

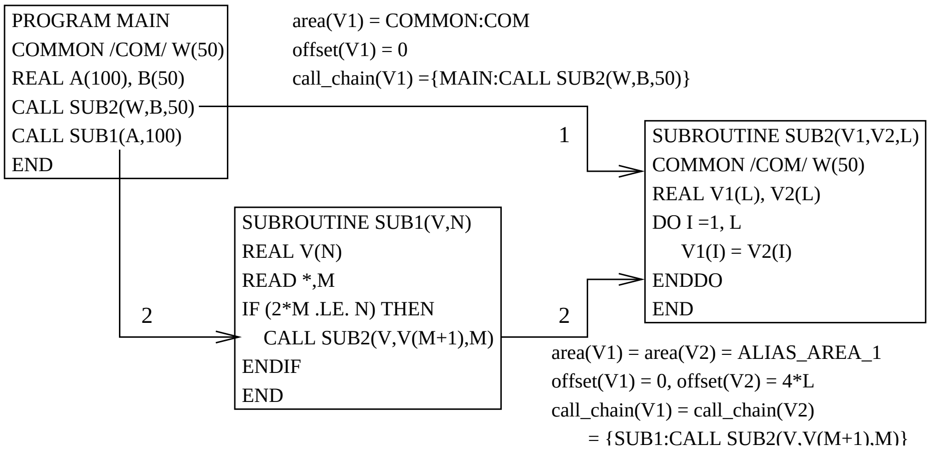

Figure 3: *Interprocedural alias propagation example*

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

4 Interprocedural Alias Verification

We enforce the Fortran standard about the restrictions on association of entities (Section 15.9.3.6 [ANS83]). However, the restrictions on association of array variables are not clear. 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 aliased elements, which identify the same storage units. Expecting users to follow this rule is not realistic because memory management is often performed by passing sections of huge workspace used as a user managed heap.

Arrays can also be treated element by element. Each array element is considered as a scalar variable. There is an alias violation if and only if an array element is defined and the aliased element of another array is referenced. That is data dependence arcs are generated between references to two different arrays. This interpretation is the most complicated because we must compare between all different referenced array elements during program execution, which is not often performed efficiently using static analysis. So we give other definitions of the so called writing violation on aliased variables in a subprogram, which is finer than the first interpretation but less complicated than the last one.

4.1 Alias Violation Condition

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 the execution of the subprogram.*

Note that if the storage sequences of variables concerned are disjoint, they share no storage unit. Therefore, we have:

Lemma 4.1 *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 that the alias violation depends on writing or not on the overlapping section of two aliased variables. The sequence of storage units that are defined during execution of the subprogram is called *defining sequence*.

Lemma 4.2 *In a subprogram, if the defining sequence of every formal parameter (every common variable) is disjoint with the storage sequence 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 4.3 *In a subprogram, if there exists a formal parameter (a common variable) whose defining 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 during the propagation phase) are excluded. To compute the defining 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* [KAC+96] of a procedure on its formal and common variables. This is an over approximation of the exact effects. By using this information, if a variable is never defined during the procedure execution, we do not have to treat it.

4.2 Alias Verification Algorithm

A necessary condition for two formal parameters or one formal parameter and one global variable to be aliased by a call chain is that their addresses have the same area. The alias verification applied for each procedure having at least one formal parameter and possible alias is given by Algorithm 4.1.

Algorithm 4.1
procedure *Interprocedural_Alias_Verification (p)*
    p : current procedure
begin
    for each *maybe_aliased_variables_by_call_chain*(v_1, v_2, cc)
        o_1 = *corresponding_offset*(v_1)
        o_2 = *corresponding_offset*(v_2)
        if $o_1 \neq$ *unknown and* $o_2 \neq$ *unknown* **then**
            alias_check_in_callee(v_1, v_2, o_1, o_2, p)
        else
            c = *direct_call_site*(cc)
            q = *direct_caller*(cc)
            o'_1 = *offset_in_caller*(v_1, c, q)
            o'_2 = *offset_in_caller*(v_2, c, q)
            if $o'_1 \neq$ *unknown and* $o'_2 \neq$ *unknown* **then**
                alias_check_in_caller$(v_1, v_2, o'_1, o'_2, q)$
            else
                direct_alias_check(v_1, v_2, p)
            endif
        endif
    endfor
end

For each maybe aliased pair of variables where at least one of them is present in the summary write effects of the procedure, the verification works as follow. If the offsets of both variables are known, depending on the type of variable, scalar or array, different checks are performed in the frame of the current procedure (Algorithm 4.2). As a matter of space, we only describe the most complicated case, when both variables are array variables. In the cases of Lemma 4.1 and Lemma 4.2, there is no alias violation between the two arrays. Otherwise, a test is inserted before each statement that writes on one variable. Flags are inserted before each call site in the call chain to mark if the execution reaches the current statement or not. If one clause $o_j \leq r_i$ or $r_i \leq o_j + s_j$ $(i, j = 1, 2\ i \neq j)$ is known to be true at compile-time, it can be removed from the test condition.

Algorithm 4.2
procedure *Alias_check_in_callee* (v_1, v_2, o_1, o_2, p)
begin
  ... both variables are array variables ...
  s_1 = *size_of_storage_sequence*(v_1)
  s_2 = *size_of_storage_sequence*(v_2)
  if $(o_2 + s_2 \leq o_1)$ *or* $(o_1 + s_1 \leq o_2)$ *is true* **then**
    no alias violation
  else
    for each *statement* s *in* p *that writes on* $v_i, i = 1, 2$
      r_i = o_i + *element_size*(v_i) * (*subscript_value*(v_i, s) - 1)
      if $r_i < o_j$ *or* $r_i > o_j + s_j$ *is true* **then**
        no alias violation
      else
        insert $flag_k$ = *.true. before each call site* k *in cc* $(k = 1, n)$
        insert IF ($flag_1$ *.and....*$flag_n$ *.and.* $(o_j \leq r_i)$ *.and.* $(r_i \leq o_j + s_j)$)
          STOP message before s
      endif
    endfor
  endif
end

If at least one offset is unknown because it cannot be translated to the frame of the called procedure in the propagation phase, we try to check for alias violation in the frame of the direct caller of the current procedure (Algorithm 4.3). The new offset of each variable is computed in the caller's frame by using function *offset_in_caller*, described as follows:

- 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 left hand-side expression in Definition 3.5. 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.

Algorithm 4.3
procedure *Alias_check_in_caller*(v_1, v_2, o'_1, o'_2, q)
begin
  ... both variables are array variables ...
  s_1 = *size_of_storage_sequence*(v_1)
  s_2 = *size_of_storage_sequence*(v_2)
  if s_1 *and* s_2 *can be translated to the frame of* q **then**
    s'_1 = *translate_to_caller_frame*(s_1, c, p, q)
    s'_2 = *translate_to_caller_frame*(s_2, c, p, q)
    if $(o'_2 + s'_2 \le o'_1)$ *or* $(o'_1 + s'_1 \le o'_2)$ *is true* **then**
      no alias violation
    else
      for each *statement* s *in* p *that writes on* $v_i, i = 1, 2$
        r_i = o_i + *element_size*(v_i) * (*subscript_value*(v_i, s) - 1)
        if r_i *can be translated to the frame of* q **then**
          r'_i = *translate_to_caller_frame*(r_i, c, p, q)
          if $r'_i < o'_j$ *or* $r'_i > o'_j + s'_j$ *is true* **then**
            no alias violation
          else
            insert $flag_k = .true.$ *before each call site* k *in cc* $(k = 1, n-1)$
            insert IF $(o'_j \le r'_i).and.(r'_i \le o'_j + s'_j)$ $flag_n = .true.$ *before* c
            insert IF $(flag_1 .and. ... flag_n)$ *STOP message before* s
          endif
        else
          direct_alias_check(v_1, v_2, p)
        endif
      endfor
    endif
  else
    direct_alias_check(v_1, v_2, p)
  endif
end

If both new offsets can be computed in the caller's frame, we repeat the alias violation checking between two variables as above, but in the frame of the caller. Each time the size of the storage sequence or the subscript value is needed, we have to translate it to the caller's frame, by using global variables information and the bindings between actual and formal parameters. Tests inserted before each statement defining a variable in the current procedure and before the direct call site are different from those in the first case.

If the translation is not possible, or one offset is unknown, we have to use the direct alias check version that inserts before each statement defining a variable, for example v_1, the procedure call $\mathtt{ALIAS\_CHECK}(\mathtt{message}, \mathtt{v_1}, \mathtt{r_1}, \mathtt{v_2}, \mathtt{s_2})$ where $\mathtt{r_1}$ is computed as in Algorithm 4.2, $\mathtt{s_2}$ is the size of the storage sequence of $\mathtt{v_2}$. $\mathtt{ALIAS\_CHECK}(\mathtt{char} * \mathtt{message}, \mathtt{void} * \mathtt{p_1}, \mathtt{int} * \mathtt{i_1}, \mathtt{void} * \mathtt{p_2}, \mathtt{int} * \mathtt{i_2})$ is a C function that takes the base address in the memory 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 exploited.

4.3 Alias Verification Example

In Figure 3 example, no alias violations are caused by `CALL SUB2(V,V(M+1),M)` because the intersection between [0:4*L-1] and [4*L:8*L-1] is empty (the storage sequence size of each array is 4*L). However, alias violations occur when writing on `V1(I)` which is aliased with an element in `W`. The instrumented code is shown in Figure 4.

```
PROGRAM MAIN
COMMON /COM/ W(50)
REAL A(100),B(50)
COMMON /ALIAS_FLAGS/ ALIAS_FLAG(1)
LOGICAL ALIAS_FLAG(1)
ALIAS_FLAG(1) = .TRUE.
CALL SUB2(W,B,50)
CALL SUB1(A,100)
END

SUBROUTINE SUB2(V1,V2,L)
COMMON /COM/ W(50)
REAL V1(L),V2(L)
COMMON /ALIAS_FLAGS/ ALIAS_FLAG(1)
LOGICAL ALIAS_FLAG(1)
DO I = 1,L
  IF (I.GE.1.AND.I.LE.50.AND.ALIAS_FLAG(1)) STOP "Alias violation:
    write on V1, aliased with W by CALL SUB2(W,B,50) in MAIN"
  V1(I) = V2(I)
ENDDO
END
```

Figure 4: *Interprocedural alias verification example*

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

5 Impact of Alias Violation on Program Optimizations

Although modifying an aliased variable may let the program produce incorrect results or behave unpredictably when using optimization, this violation on alias restrictions occurs occasionally in established programming practice. This section contains several examples of code optimizations which are not safe when alias conditions are violated by the user. Different optimization levels of the Sun WorkShop 6 FORTRAN 77 compiler are used to show different results obtained with each example.

Figure 5 shows that a trivial register allocation can be incorrect if alias information is not taken into account. The register allocation when using the optimization level 3 of the SUN compiler leads to incorrectly assigning the value 120 to `L`, instead of 144 because `M` is aliased with `N` by the call in the main program but its value is not recomputed after the assignment `N = 12`.

```
PROGRAM MAIN
CALL SUB(I,I)
END
```

```
SUBROUTINE SUB(M,N)
M = 10
N = 12
L = M * N
PRINT *,L
END
```

Figure 5: *Correctness of register optimization*

Figure 6 depicts another example about the correctness of a common subexpression elimination. According to [Muc97], an occurrence of an expression in a program is a common subexpression if there is another occurrence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations. The second occurrence of expression `M + L` is not a common subexpression because the last condition on the unchanged operands is not satisfied by the aliasing between `M` and `N` and the assignment `N = 6`. But with the optimization level 3 of the SUN compiler, the result is 6 instead of 7 (when no optimization is used).

```
PROGRAM MAIN
I = 5
CALL SUB(I,I)
END
```

```
SUBROUTINE SUB(M,N)
L = 1
K = M + L
N = 6
PRINT *, M + L
END
```

Figure 6: *Correctness of common subexpression elimination*

The invariant code motion in Figure 7 for the assignment `M = 2` is not correct if the alias between `M` and `N` is not taken into account. That is why with the optimization level 3, the SUN compiler gives the result 6 instead of the expected output 3.

```
PROGRAM MAIN
I = 5
CALL SUB(I,I)
END
```

```
SUBROUTINE SUB(M,N)
L = 1
DO WHILE (L.LE.N)
  M = 2
  L = L + 1
ENDDO
PRINT *, L
END
```

Figure 7: *Correctness of invariant code motion*

Figure 8 shows the impact of aliasing on loop parallelization. If `N` is greater than 30, there are aliases between array elements `X(I+30)` and `Y(I)` because they point to the same actual array element `A(I+30)`. The loop in module `COPY` can be unrolled as shown in Figure 9, as well as the corresponding computation on array `A`. So if this loop is executed in parallel, the result would be different than if it is executed sequentially, i.e the value of `Y(31)` is 31 instead of 1.

```
PROGRAM MAIN
REAL A(100)
DO I = 1,100
   A(I) = I
ENDDO
READ *, N
CALL COPY(A,A(31),N)
PRINT *,A
END
```

```
SUBROUTINE COPY(X,Y,N)
REAL X(N), Y(N)
DO I = 1,N
  Y(I) = X(I)
ENDDO
END
```

Figure 8: *Correctness of loop parallelization*

```
Y(1) = X(1)
...
Y(I) = X(I)
...
Y(31) = X(31)
...
```

```
A(31) = A(1) = 1
...
A(30 + I) = A(I) = I
...
A(61) = A(31) = 1
...
```

Figure 9: *Loop unrolling from Figure 8*

All the above examples show that alias analysis is essential to perform most optimization correctly or to warn the programmer. The aliases between variables create more dependences between program statements. As explained in [AK02], 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.

So the effect of new dependence arcs created by standard violation aliases on the data dependence graph is important because it decides whether other program transformations on the procedure are safe or not. If these new data dependence arcs are redundant with existing paths in the data dependence graph, the aliases have no impact on ordering-based optimizations. Take as an example a piece of code from the benchmark *hydro2d* in the SPEC95 CFP benchmarks [DD98] where calls to subroutine `FCT` make two formal parameters `UNEW` and `UTRA` aliased and they are defined by the called module. As shown in Figure 10, 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.

```
      SUBROUTINE FCT(UNEW,UTRA,UOLD)
      PARAMETER (MP = 402, NP = 160)
      DIMENSION UNEW(MP,NP), UTRA(MP,NP), UOLD(MP,NP)
      DO I = 1,MQ
         DO J = 1,NQ
s1          DZ1(I,J) = UOLD(I+1,J) - UOLD(I,J)
s2          AZ1(I,J) = UTRA(I+1,J) - UTRA(I,J)
         ENDDO
      ENDDO
      DO I = 1,MQ
         DO J = 1,NQ
s3          UTRA(I,J) = UTRA(I,J) + DZ1(I,J) - DZ1(I-1,J)
         ENDDO
      ENDDO
      DO I = 1,MQ
         DO J = 1,NQ
s4          UNEW(I,J) = UTRA(I,J) - AZ1(I,J) - AZ1(I-1,J)
         ENDDO
      ENDDO
      END
```

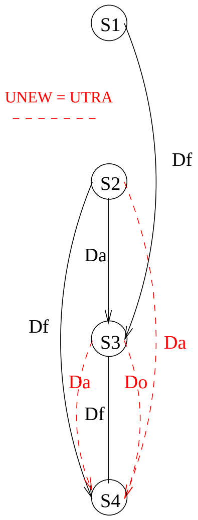

Figure 10: *Dependence graph of a code fragment from hydro2d*

Actually, this phase has been implemented in PIPS for scalar variables and it treats arrays as atomic variables. When aliases create more dependence arcs, the analysis is able to tell if these arcs modify the existing dependence graph, so the aliases have impacts on optimization or not. The alias violation checking as well as the effect of array alias violation is studied with the SPEC95 CFP benchmarks, that is shown in the next section.

6 Experimental Results

We use the SPEC95 CFP benchmarks that contain ten applications written in Fortran 77. 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 nine benchmarks, the number of modules 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 SPEC95 CFP.

Program	**Line**	**Mod**	**Call**	**Flag**	**Test**	**Check**	**Mod**	**Comp.**	**Viol.**	**AR**	**DG**
tomcatv	190	1	0	0	0	0	0	0s	No		
swim	429	6	5	0	0	0	0	1s	No		
su2cor	2332	35	166	0	0	0	0	14s	No		
hydro2d	4292	42	98	8	12	0	2	9s	Yes		*
mgrid	484	12	23	0	0	10	3	1s	Yes		*
applu	3868	16	27	6	6	0	2	8s	Yes		*
turb3d	2101	23	111	60	156	13	12	10s	Yes		*
apsi	7361	96	190	23	194	2945	66	92s	Yes	*	
fpppp	2784	38	27	0	0	250	1	30s	No		
wave5	7764	105	243	36	334	495	24	98s	Yes	*	

Table 1: *SPEC95 CFP: numbers of lines, modules and calls; numbers of inserted flags, tests, checks, modules with possible aliases; compilation times; alias violation results: array resizing and dependence graph categories*

Some array declarations conflict with alias analysis. The *assumed-size array* declaration (Section 5.1.2 [ANS83]) with an asterisk as the last dimension and array declaration with a final dimension specified as 1 prevent us from knowing the storage sequence size of some arrays in the called modules. These kinds of declarations may cause spurious array bound violations, inhibit program analyses such as array bound checking, uninitialized variable analysis, program debugging, etc. Unlike other compilers that ignore these assumed-size arrays, we deal with this by applying *array resizing* [AN01] to five benchmarks: *applu, turb3d, apsi, fpppp* and *wave5*. Assumed-size declarations also occur in two other benchmarks, *su2cor* and *hydro2d*, but the corresponding arrays are not involved in aliases so array resizing is not needed for these benchmarks. The array resizing phase tries to compute automatically the proper upper bound for the 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. All assumed-size arrays are resized.

The numbers of inserted flags, tests, direct alias checks, modules with possible aliases and the compilation times (in seconds) are also reported in Table 1. Compilation times for both alias propagation and verification phases are measured on an UltraSparc II 440MHz 256Mo RAM. Codes with generated checks are then compiled and executed using the standard input data for SPEC95 CFP benchmarks. Six out of ten 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 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 files. Runtime slowdown cannot be measured for the other benchmarks because the execution is interrupted at the first violation.

Column 8 shows the number of modules where there may be aliases between its formal parameters and common variables. It is important to see that, in comparison to Column 3, this number only takes a small percentage. For example, only one called module in *fpppp* must be compiled with the dummy aliasing assumption to assert the presence of aliases. Meanwhile, the total number of called modules in this benchmark is 37. The no alias assumption can be safely applied to the remaining modules in the program; hence the execution performance is improved.

Two kinds of alias violations appear in the six 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. Figure 11 gives an illustrating example.

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

Figure 11: Assumed-size array declaration example

Because Fortran uses the column-major scheme to store arrays, the address in memory of a given array reference is calculated from the base address of the array and the subscript value, 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 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 regions [AN01]. They give information about the set of array elements accessed during the code execution. In this case, array declarations are more precise and greatly reduce 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, etc. A good programming practice is to pass disjoint array sections to 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, MATLAB and APL [AN01].

The second category of alias violation includes real violations because two scalar variables or two array elements share the same memory location and one of them is written. This category includes *hydro2d*, *mgrid, applu* and *turb3d* benchmarks. However, after applying the phase of checking alias impact on dependence graph, the legal schedules in these programs are not changed by these aliases. Example on *hydro2d* in Section 5 illustrates this kind of violation. Aliases in *mgrid, applu* and *turb3d* are simpler than in *hydro2d* because one aliased variable is only defined once in the referenced procedure, and the other aliased variables are not used in this procedure.

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. We applied the analysis to a large scale industrial application, about 100.000 lines of code, and found several potential bugs.

7 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 perform program analyses under less restrict assumptions about the program quality. For example, without alias information, the used-before-set analysis can give false results because some variables may be initialized implicitly by aliasing. Other optimizations such as load and store reordering, partial redundancy elimination, constant propagation and copy propagation are hard or impossible to carry out because of aliasing. Our analysis can be applied to other programming languages with the 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 programmers to correct errors in order to gain performance by applying optimizations safely. Once alias checks are generated, the instrumented code respects the standard about aliasing. Other techniques such as code hoisting, partial redundancy elimination, induction variable optimization can be safely used to optimize the generated code. On the other hand, a program can be guaranteed to be free of dynamic alias errors by our static analysis.

An important issue is the impact of aliasing on the dependence graph and hence on code optimizations. New scheduling constraints cause real alarms for program transformations. The 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 well debugged codes are necessary.

The result can be applied to Fortran 90. Useless array copies can be suspended and useful procedure cloning can be performed to improve the handling of array sections if the alias information is available.

Another 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. The PIPS software and our alias checking implementations are available on http://www.cri.ensmp.fr/pips.

References

[ABC+88] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeane Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, San Francisco, 2002.

[AN01] Corinne Ancourt and Thi Viet Nga Nguyen. Array resizing for code debugging, maintenance and reuse. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 32–37, June 2001.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Denmark, May 1994.

[ANS83] ANSI. *Programming Language FORTRAN, ANSI X3.9-1978, ISO 1539-1980.* American National Standard Institute, New York, 1983.

[App01] John Appleyard. Comparing Fortran compilers. *ACM SIGPLAN - Fortran Forum*, 20(1):6–10, 2001.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar77] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *ACM Symposium on Principles of Programming Languages*, pages 119–131, January 1977.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer induced aliases and side effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CK88] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *ACM Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[Coo84] Keith D. Cooper. Analyzing aliases of reference formal parameter. In *ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1984.

[Cou86] Deborah S. Coutant. Retargetable high level alias analysis. In *ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[CRL01] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Transactions on Software Engineering*, 27(6):481–512, June 2001.

[DD98] Jozo J. Dujmovic and Ivo Dujmovic. Evolution and evaluation of SPEC benchmarks. *ACM SIGMETRICS*, 26(3):2–9, 1998.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fahndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 260–278. Springer-Verlag, 2001.

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–58, June 2001.

[GP01] Wolfgang Gellerich and Erhard Ploedereder. Parameter-induced aliasing in Ada. In *Ada-Europe*, volume 2043 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2001.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[HBCM92] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. Interprocedural array analysis: How much precision do we need? In *Workshop on Compilers for Parallel Computers*, pages 48–64, June 1992.

[HBCM94] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet ? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, June 2001.

[HP01] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39:31–55, 2001.

[HT01a] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, June 2001.

[HT01b] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, June 2001.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[Iri93] François Irigoin. Interprocedural analyses for programming environments. In *Environments and Tools for Parallel Scientific Computing*, pages 333–350. Elsevier, 1993.

[KAC+96] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS: A workbench for building interprocedural parallelizers, compilers and optimizers. In *European Parallel Tool Meeting*, October 1996.

[LH01] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 279–298. Springer-Verlag, 2001.

[LPH01] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, 1992.

[MDCE01] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, June 2001.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, 1997.

[MW93] Herbert G. Mayer and Michael Wolfe. Interprocedural alias analysis: implementation and empirical results. *Software - Practice and Experience*, 23(11):1201–1233, November 1993.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[RLS+01] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, 2001.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. *ACM SIGPLAN Notices*, 30(6):13–22, June 1995.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Wal91] David W. Wall. Limits of instruction-level parallelism. *ACM SIGPLAN Notices*, 26(4):176–188, 1991.

[WFPS02] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. Instance-wise points-to analysis for loop-based dependence testing. In *International Conference on Supercomputing*, pages 262–273, 2002.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[ZC90] Hans P. Zima and Barbara M. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, Reading, 1990.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 81–92, October 1996.