Automatic Detection of Uninitialized Variables

Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt, and Fabien Coelho

Ecole des Mines de Paris, 77305 Fontainebleau, France
{nguyen,irigoin,ancourt,coelho}@cri.ensmp.fr

Abstract. One of the most common programming errors is the use of a variable before its definition. This undefined value may produce incorrect results, memory violations, unpredictable behaviors and program failure. To detect this kind of error, two approaches can be used: compile-time analysis and run-time checking. However, compile-time analysis is far from perfect because of complicated data and control flows as well as arrays with non-linear, indirection subscripts, etc. On the other hand, dynamic checking, although supported by hardware and compiler techniques, is costly due to heavy code instrumentation while information available at compile-time is not taken into account.
This paper presents a combination of an efficient compile-time analysis and a source code instrumentation for run-time checking. All kinds of variables are checked by PIPS, a Fortran research compiler for program analyses, transformation, parallelization and verification. Uninitialized array elements are detected by using imported array region, an efficient inter-procedural array data flow analysis. If exact array regions cannot be computed and compile-time information is not sufficient, array elements are initialized to a special value and their utilization is accompanied by a value test to assert the legality of the access. In comparison to the dynamic instrumentation, our method greatly reduces the number of variables to be initialized and to be checked. Code instrumentation is only needed for some array sections, not for the whole array. Tests are generated as early as possible. In addition, programs can be proved to be free from used-before-set errors statically at compile-time or, on the contrary, have real undefined errors. Experiments on SPEC95 CFP show encouraging results on analysis cost and run-time overheads.

1 Introduction

Used-before-set refers to the error occurring when a program uses a variable which has not been assigned a value. This uninitialized variable, once used in a calculation, can be quickly propagated throughout the entire program and anything may happen. The program may produce different results each time it runs, or may crash for no apparent reason, or may behave unpredictably. This is also a known problem for embedded software. Some programming languages such as Java and C++ have built-in mechanisms that ensure memory to be initialized to default values, which make programs work consistently but may not give intended results.

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 217–231, 2003.

To detect this kind of error, two approaches can be used: compile-time analysis and run-time checking. However, compile-time analysis is far from perfect because complicated data and control flows result in a very large imprecision. Furthermore, the use of global variables and arrays with non-linear, indirection subscripts, etc sometimes makes static checking completely ineffective, leading to many spurious warnings. In addition, some other program analyses such as points-to analysis [1], alias analysis and array bound checking [2] are prerequisite for the detection of uninitialized variable uses.

On the other hand, pure dynamic checking is costly due to heavy code instrumentation while information available at compile-time is not taken into account. The slowdown between instrumented and uninstrumented codes has been measured to be up to 130 times in [3]. Dynamic checking is not so effective that, as shown in a report comparing Fortran compilers [4], only some Lahey/Fujitsu and Salford compilers offer run-time checking for all kinds of variables. The other compilers such as APF version 7.5, G77 version 0.5.26, NAS version 2.2 and PGI version 3.2-4 do not have this option. Intel Fortran Compiler version 6.0 and NAGWare F95 version 4.1 only check for local and formal scalar variables; array and global variables are omitted. The code instrumentation degrades the execution performance so it can only be used to create a test version of the program, not a production version. In addition, run-time checking only validates the code for a specific input.

With the growth of hardware performance - processor speed, memory bandwidth - software systems have become more and more complicated to solve better real application problems. Debugging several million lines of code becomes more difficult and time-consuming. Execution time overheads of dynamic checking or a large number of possibly undefined variable warnings issued by static checking are not highly appreciated. Efficient compile-time analysis to prove the safety of programs, to detect statically program errors, or to reduce the number of run-time checks is necessary. The question is, by using advanced program analyses, i.e interprocedural array analysis, can static analysis be an adequate answer to the used-before-set problem for the scientific codes? If not, can a combination of static and dynamic analyses reduce the cost of uninitialized variable checking? The goal of our research is to provide a more precise and efficient static analysis to detect uninitialized variables, and if sufficient information is not available, run-time checks are added to guarantee the program correctness.

The paper is organized as follows. Section 2 presents some related work on uninitialized variable checking. Section 3 describes the imported array regions analysis. Our used-before-set verification is presented in Section 4. Experimental results obtained with the SPEC95 CFP benchmark are given in Section 5. Conclusions are drawn in the last section.

2 Related Work

To cope with the used-before-set problem, some compilers silently initialize variables to a predefined value such as zero, so that programs work consistently, but

give incorrect results. Other compilers provide a run-time check option to spot uses of undefined values. This run-time error detection can be done by initializing each variable with a special value, depending on the variable type. If this value is encountered in a computation, a trap is activated. This technique was pioneered by Watfor, a Fortran debugging environment for IBM mainframes in the 70's, and then used in Salford, SGI and Cray compilers.

For example, the option *trap_uninitialized* of SGI compilers forces all real variables to be initialized with a NaN value (Not a Number - IEEE Standard 754 Floating Point Numbers) and when this value is involved in a floating-point calculation, it causes a floating-point trap. This approach raise several problems. Exception handler functions or compiler debugging options can be used to find the location of the exception but they are platform- and compiler-dependent. Furthermore, the IEEE invalid exception can be trapped for other reasons, not necessarily an uninitialized variable. In addition, when no floating-point calculation is done, e.g in the assignment `X = Y`, no used-before-set error is detected which makes tracking the origin of an error detected later difficult. Other kinds of variables such as integer and logical are not checked for uninitialization. In conclusion, the execution overhead of this technique is low but the used-before-set debugging is almost impossible.

Other compilers such as Lahey/Fujitsu compilers, SUN dbx debugger, etc use a *memory coloring algorithm* to detect run-time errors. For example, Valgrind, an open-source memory debugger (*http://devel-home.kde.org/sewardj*) tracks each byte of memory in the original program with nine status bits, one of which tracks the addressability of that byte, while the other eight track the validity of the byte. As a result, it can detect the use of single uninitialized bits, and does not report spurious errors on bit-field operations. The object code instrumentation method is used in Purify, a commercial memory access checking tool. Each memory access is intercepted and monitored. The advantage of this method is that it does not require recompilation and it supports libraries. However, the method is instruction set and operating system dependent. The memory usage is bigger than other methods. Furthermore, the program semantics is lost. An average slowdown of 5.5 times between the generated code and the initial code is reported for Purify in [5].

The plusFORT toolkit (*http://www.polyhedron.com*) instruments source code with probe routines so that uninitialized data can be spotted at run-time using any compiler and platform. There are functions to set variables to undefined, and functions to verify if a data item is defined or not. Variables of all types are checked. The amount of information about violations provided by plusFORT is precise and useful for debugging. The name of the subprogram, the line where the reference to an uninitialized variable occurred is reported in a log-file. However, the instrumentation is not so effective because of inserted code. Fig. 1 shows that plusFORT can detect bugs which depend on external data, but the execution time is greatly increased with such dynamic tests.

To reduce the execution cost, illegal uses of an uninitialized variable can be detected at compile-time by some compilers and static analyzers. LCLint [6] is

```
      SUBROUTINE QKNUM(IVAL, POS)
      INTEGER IVAL, POS, VALS(50), IOS
      CALL SB$ENT('QKNUM', 'DYNBEF.FOR')
      CALL UD$I4(IOS)
      CALL UD$AI4(50, VALS)
      READ (11, *, IOSTAT = IOS) VALS
      CALL QD$I4('IOS', IOS, 4)
      IF (IOS .EQ. 0) THEN
         DO POS = 1,50
            CALL QD$I4('VALS(POS)',VALS(POS),6)
            CALL QD$I4('IVAL', IVAL, 6)
            IF (IVAL .EQ. VALS(POS)) GOTO 10
         ENDDO
      ENDIF
      POS = 0
10    CALL SB$EXI
      END
```

Fig. 1. Example with plusFORT probe routines

an advanced C static checker that uses formal specification written in the LCL language to detect instances where the value of a location may be used before it is defined. Although few spurious warnings are generated, there are cases where LCLint cannot determine if a use-before-definition error is present, so a message may be issued for a non-existing problem. In other cases, a real problem may go undetected because of some simplified assumptions.

The static analyzer ftnchek (*http://www.dsm.fordham.edu/ftnchek*) gives efficient warnings about possible uninitialized variables, but the analysis is not complete. Warnings about common variables are only given for cases in which a variable is used in some routine but not set in any other routine. It also has the same problems as LCLint about non-existing or undetected errors, because of, for example the simplified rule about equivalenced arrays.

Reps et al. [7] consider the possibly uninitialized variable as an IFDS (*interprocedural, finite, distributive, subsets*) problem. A precise interprocedural data flow analysis via graph reachability is implemented with the Tabulation Algorithm to report the uses of possibly uninitialized variables. They compare the accuracy and time requirement of the Tabulation Algorithm with a naive algorithm that considers all execution paths, not only interprocedurally realizable ones. The number of possibly uninitialized variables detected by their algorithm ranges from 9% to 99% of that detected by the naive one. However, this is only an over-approximation that does not give an exact answer if there are really use-before-set errors in the program or not. The number of possibly undefined variables is rather high, 543 variables for a 897 line program, 894 variables for a 1345 line program.

PolySpace technologies (*http://www.polyspace.com*) apply abstract interpretation, the theory of semantic language approximations, to detect automatically

read accesses to non-initialized data. This technique predicts efficiently run-time errors and information about maybe non-initialized variables can be useful for the debugging process of C and Ada programs. But no data is given by the PolySpace group so we cannot compare with them.

Another related work of Feautrier [8] proposes to compute for each use of a scalar or array cell its *source function*, the statement that is the source of the value contained therein at a given instant of the program execution. Uninitialized variable checking can be done by verifying in the source the presence of the sign $\perp$, which indicates access to an undefined memory cell. Unfortunately, the input language in this paper is restricted to assignment statements, `FOR` loops, affine indices and loop limits.

The main difficulties encountered by static analysis for the used-before-set verification are complicated data and control flows with different kinds of variables. This explains why only a small set of variables such as scalar and local variables is checked by some static analyzers and compilers. Our motivation is to develop a more efficient program analysis to the used-before-set problem by using imported array regions.

3 Imported Array Region Analysis

Array region analyses collect information about the way array elements used and defined by programs. A *convex array region*, as defined in [9,10], is a set of array elements described by a convex polyhedron [11]. Its constraints link the region parameters that represent the array dimensions to the values of the program integer scalar variables. A region has the approximation `MUST` if every element in the region is accessed with certainty, `MAY` if its elements are simply potentially accessed and `EXACT` if the region exactly represents the requested set of array elements. There were two kinds of array regions, `READ` and `WRITE` regions, that represent the effects of program statements on array elements. For instance, `A-WRITE-EXACT-{PHI1==1,PHI2==I}` is the array region of statement `A(1,I)=5`. The region parameters `PHI1` and `PHI2` respectively represent the first and second dimensions of `A`.

The order in which references to array elements are executed, array data flow information, is essential for program optimizations. `IN` array regions are introduced in [10,12] to summarize the set of array elements whose values are *imported* (or *locally upward exposed*) by the current piece of code. One array element is imported by a fragment of code if there exists at least one use of the element whose value has not been defined earlier in the fragment itself. For instance, in the illustrative example in Fig. 2, the element `B(J,K)` in the second statement of the second J loop is read but its value is not imported by the loop body because it is previously defined by the first statement. On the contrary, the element `B(J,K-1)` is imported from the first J loop. The propagation of `IN` regions begins from the elementary statements to compound statements such as conditional statements, loops and sequences of statements, and through procedure calls. The input language of our analysis is Fortran.

```
K = FOO()
DO I = 1,N
   DO J = 1,N
      B(J,K) = J + K
   ENDDO
   K = K + 1
   DO J = 1,N
      B(J,K) = J*J - K*K
      A(I) = A(I) + B(J,K) + B(J,K-1)
   ENDDO
ENDDO
```

Fig. 2. Imported array region example

Elementary Statement. The `IN` regions of an assignment are all read references of the statement. Each array reference on the right hand side is converted to an elementary region. Array references in the subscript expressions of the left hand side reference are also taken into account. These regions are `EXACT` if and only if the subscripts are affine functions of the program variables. To save space, regions of the same array are merged by using the union operator.

The `IN` regions of an input/output statement are more complicated. The input/output status, error and end-of-file specifiers are handled with respect to the Fortran standard [13]. The order of variable occurrences in the input list is used to compute the `IN` regions of an input statement. For example, in the input statement `READ *,N,(A(I),I=1,N)`, `N` is not imported since it is written before being referenced in the implied-`DO` expression `(A(I),I=1,N)`.

Conditional Statement. The `IN` regions of a conditional statement contain the `READ` regions of the test condition, plus the `IN` regions of the true branch if the test condition is evaluated true, or the `IN` regions of the false branch if the test condition is evaluated false. Since the test condition value is not always known at compile-time, the `IN` regions of the true and false branches, combined with the test condition, are unified in the over-approximated regions.

Loop Statement. The `IN` regions of a loop contain array elements imported by each iteration but not previously written by the preceding iterations. Given the `IN` and `WRITE` regions of the loop body, the loop `IN` regions contain the imported array elements of the loop condition, plus the imported elements of the loop body if this condition is evaluated true. Then, when the loop is executed again, in the program state resulting from the execution of the loop body, they are added to the set of loop imported array elements in which all elements written by the previous execution are excluded.

Sequence of Statements. Let s be the sequence of instructions $s_1; s_2; ..s_n;$. The `IN` regions of the sequence contain all elements imported by the first statement s_1, plus the elements imported by $s_2; ..s_n;$ after the execution of s_1, but not written by the latter.

Control Flow Graph. Control flow graphs are handled in a very straightforward fashion: the `IN` regions of the whole graph are equal to the union of the

IN regions imported by all the nodes in the graph. Every variable modified at a node is projected from the regions of all other nodes. All approximations are decreased to MAY.

Interprocedural Array Region. The interprocedural propagation of IN regions is performed by a reverse invocation order traversal on the program call graph: a procedure is processed after its callees. For each procedure, the summary IN regions are computed by eliminating local effects from the IN regions of the procedure body. Information about formal parameters, global and static variables are preserved. The resulting summary regions are stored in the database and retrieved each time the procedure is invoked. At each call site, the summary IN regions of the called procedure are translated from the callee's name space into the caller's name space, using the relationships between actual and formal parameters, and between the declarations of global variables in both routines.

Fig. 3 shows the IN regions computed for the running example. In the body of the second J loop, array elements A(I), B(J,K) and B(J,K-1) are imported by the second statement. Since B(J,K) is defined by the first statement, only A(I) and B(J,K-1) are imported by the loop body. The IN regions of the second J loop are <B(PHI1,PHI2)-IN-EXACT-{1<=PHI1,PHI1<=N,PHI2==K-1}> and <A(PHI1)-IN-EXACT-{PHI1==I}>. After propagating upward these IN regions through statement K=K+1, region of array B becomes <B(PHI1,PHI2)-IN-EXACT-{1<=PHI1,PHI1<=N,PHI2==K}>. Once again, all array elements in this region are defined by the first J loop, so only array elements of A are imported by the code fragment in this example.

Array analysis is also studied in many papers [14,15,16,17,18,19]. The convex array regions implemented in PIPS are based on the Regions method [20] where IN region, the set of imported array elements, is somewhat similar to ExposedRead region in [15], UE set in [16], USE(s) in [17] and input effects in [19]. However, our IN region and the others differ. For example, in [15], array element sets are represented by lists of polyhedra and there is no exact representation, only under- and over-approximations. The ExposedRead sets contain array elements which are used in the continuation of the whole program before being defined, while IN regions are restricted to the current level in the hierarchical control flow graph. In [16], an array element set is a list of Regular Section Descriptors with bounds and step, guarded by predicates derived from IF conditions. When insufficient information is available, our MAY regions should be more accurate because we can keep more information about PHI variables. Input effects [19] give for each array element its first use in the considered fragment of code. This is similar to our IN regions but the precise statement instance in which the reference is performed is kept in the summary. The implementation choice really is the trade-off between efficiency and precision. The IN regions developed by [12] are used in this paper, with some improvements.

```
       K = FOO()
C      <A(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=N}>
       DO I = 1,N
C      <A(PHI1)-IN-EXACT-{PHI1==I}>
          DO J = 1,N
             B(J,K) = J + K
          ENDDO
C      <B(PHI1,PHI2)-IN-EXACT-{1<=PHI1, PHI1<=N, PHI2==K}>
C      <A(PHI1)-IN-EXACT-{PHI1==I}>
          K = K + 1
C      <B(PHI1,PHI2)-IN-EXACT-{1<=PHI1, PHI1<=N, PHI2==K-1}>
C      <A(PHI1)-IN-EXACT-{PHI1==I}>
          DO J = 1,N
C      <B(PHI1,PHI2)-IN-EXACT-{PHI1==J, PHI2==K-1}>
C      <A(PHI1)-IN-EXACT-{PHI1==I}>
             B(J,K) = J*J - K*K
C      <B(PHI1,PHI2)-IN-EXACT-{PHI1==J, K-1<=PHI2, PHI2<=K}>
C      <A(PHI1)-IN-EXACT-{PHI1==I}>
             A(I) = A(I) + B(J,K) + B(J,K-1)
          ENDDO
       ENDDO
```

Fig. 3. Computed IN regions

4 Used-Before-Set Analysis

Our used-before-set analysis is directly based on `IN` array regions. These regions are computed for arrays, but scalar variables also carry the same kind of information which is cheaper to compute. In fact, the region of a scalar has an empty predicate, i.e `<V-IN-EXACT-{}>`. Information about imported array elements and scalar variables are propagated interprocedurally, from the elementary statements to the compound statements. We traverse the program call graph in the invocation order, in which a procedure is processed after all its callers.

procedure Used_Before_Set_Analysis(p)
  p : current procedure
begin
  s := entry statement of p
  l := list of variables having IN region at s
  for each $v \in l$
    if local_variable(v, p) or global_variable_in_main_program(v, p) **then**
      if the `IN` region of v at s is `MUST` or `EXACT` **then**
        error: "Variable v is used before set"
      else /* `MAY IN` region */
        insert an initialization on v before s
        go_down_and_verify(v, s)
    else /* v is a formal parameter or a global variable in a called procedure*/
      if must_be_checked(v, p) **then**
        go_down_and_verify(v, s)

```
end
procedure go_down_and_verify(v, s)
begin
    for each sub-statement s_i of s
        if the IN region of v at s_i is EXACT then
            insert a verification on v before s_i
        else /* MAY IN region*/
            if call_statement(s_i) then
                mark must_be_checked for the corresponding
                formal or global variables in the called procedure
            else
                go_down_and_verify(v, s_i)
end
```

The list of IN regions at the module entry statement gives us the set of all possibly undefined variables of the module, and vice-versa, only variables in this list may be used before set. So at the entry statement, if the list of IN regions is empty, there is no used-before-set error in this module. Otherwise, each variable in the list is checked. In Fortran, a variable scope is always a module. The scope of a global variable declared in a module is that module but the scope of the common block where the variable is located is the whole program. So the IN regions of all global variables are propagated to the main program, although the variables are not declared in it. Depending on the variable type (local, formal or global) and the current module, we have two cases:
Case 1: Local or global variable in the main program. Depending on the region approximation, we have two sub-cases:

- If the IN region has the approximation MUST or EXACT, the variable must be used somewhere in the module before being defined and an error is detected.
- Otherwise, the region is MAY; we instrument the code by inserting an initialization function before the entry statement and go down to the sub-statements where the IN region is propagated from. Before each statement where we know with certainty that the variable must be imported, a verification function is inserted. We continue to go down for each statement with MAY regions. If this statement is a procedure call, information is added to mark that the corresponding formal parameters of the actual variable, or the corresponding global variables must be checked at the callee's level. To help the debugging process, information about the call path is added to locate the run-time error.

Case 2: Formal parameter or global variable in a called procedure. If this variable is marked as *must be checked*, we repeat the process as for local variables, but no initialization is needed since it has been performed earlier in one of the callers.

To trap premature usage, the initialization is implemented by assigning a special value to the maybe uninitialized scalar variable or array element. The verification checks whether the variable value is equal to this special value, and reports the error if one has occurred. If the variable is an array, the instrumented

code is a DO loop associated to the corresponding IN region, whereas it is a simple assignment and test for the scalar variables. All array elements in the MAY IN region are marked uninitialized and all array elements in the EXACT IN region are checked. We use the algorithm described in [21] to compute loop bounds from the region predicate, which is a polyhedron. This algorithm scans the polyhedron and uses the Fourier pairwise elimination to find loop bounds for each dimension. The generated loops have the following form:

```
IF (condition) THEN
  DO PHI1 = LOWER1, UPPER1
    A(PHI1) = special_value
  ENDDO
ENDIF
```

a. Marking initialization

```
IF (condition) THEN
  DO PHI1 = LOWER1, UPPER1
    IF (A(PHI1).EQ.special_value) STOP
  ENDDO
ENDIF
```

b. Initialization verification

However, there are some implementation problems related to the special value and the type of variable. We can use a SNaN (Signaling Not a Number) for floating variables but it is not evident for integer and logical variables. Currently, we choose the maximal integral value for integer variables, a value different from 0 and 1 for logical variables. This may raise false positive warnings when program computations really involve these special values. This is also a problem for other compiler implementations. Some memory coloring techniques can be used to avoid this problem, but at the expense of memory usage.

The efficiency of our analysis depends on the accuracy of array region analyses. The more precise the imported array regions are, the smaller the number of variables to be checked is, and code instrumentation is only used when we do not have enough information. Only array elements in the MAY IN regions are initialized and checked. Initialization and verification statements are inserted in the source code and the program is then compiled and executed normally. The executable code appears to the user to operate as the original, but if a used-before-set error is detected, the program is stopped with a message to indicate the name of the variable, the module and the call path where this error occurred. Another implemented option is that when a use before definition takes place, we do not stop the program but write details to a log-file for later analysis in order to catch several bugs in one run.

To illustrate the used-before-set analysis, we use the example of plusFORT. Fig. 4 shows the IN regions computed for module QKNUM. At the module entry, there is one IN region for variable IVAL which means that only this formal variable may be used without initialization. The variable POS is not imported by the loop because it is defined as the loop index. Neither is IOS imported by the module because its value has been defined by the READ statement, before being used in the test condition.

Array regions of the input statement are computed by taking into account the Fortran standard [13]: if the input/output status equals to zero, neither an error condition nor an end-of-file condition is encountered by the processor, all data in the input/output list are transfered. If the input/output status is not

```
      SUBROUTINE QKNUM(IVAL, POS)
      INTEGER IVAL, POS, VALS(50), IOS
C     <IVAL-IN-MAY-{}>
      READ (11, *, IOSTAT = IOS) VALS
C     <IOS-IN-EXACT-{}>
C     <IVAL-IN-MAY-{}>
C     <VALS(PHI1)-IN-MAY-{1<=PHI1, PHI1<=50, IOS==0}>
      IF (IOS .EQ. 0) THEN
C     IF (IVAL.EQ.MAXINT) STOP "IVAL is undefined in module QKNUM ..."
C     <IVAL-IN-EXACT-{}>
C     <VALS(PHI1)-IN-MAY-{1<=PHI1, PHI1<=50}>
         DO POS = 1,50
C     <IVAL-IN-EXACT-{}>
C     <POS-IN-EXACT-{}>
C     <VALS(PHI1)-IN-EXACT-{PHI1==POS}>
            IF (IVAL .EQ. VALS(POS)) GOTO 10
         ENDDO
      ENDIF
      POS = 0
10    END
```

Fig. 4. Used-before-set analysis example

equal to zero and there is no error or end-of-file specifier, execution of the executable program is terminated, no array element is defined. This is a language implementation feature but it must be respected to detect the uninitialized errors correctly. The set of array elements written by the input statement is exactly: `<VALS(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=50, IOS==0}>` and when propagating the `IN` region of array `VALS` backward, we have: `<VALS(PHI1)-IN-MAY-{1<=PHI1, PHI1<=50, IOS==0}> - <VALS(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=50, IOS==0}> = <VALS(PHI1)-IN-EXACT-{}>`. So all the elements of array `VALS` are well defined before they are used. There is no used-before-set error for the local variables in this module. By using static analysis, we prove that no instrumentation is needed, which is a big advantage with respect to the code generated by plusFORT (Fig. 1). Since we do not have any calling context, it is not possible to conclude whether the variable `IVAL` is already initialized by the callers of `QKNUM`. If the whole program was given, and following some call paths, `IVAL` may not be initialized, a verification would be inserted before the loop and inside the conditional statement, as shown in the fifth comment line in Fig. 4.

5 Experimental Results

We used the SPEC95 CFP benchmarks [22] that contain all kinds of variables: scalar and array, local, formal and global, with complicated data and control flow graphs. The experiments consist of two steps: `IN` array region computation and used-before-set analysis. Table 1 summarizes relevant information for each

Table 1. SPEC95 CFP: number of lines, modules, scalar variables (total, maybe uninitialized and percentage), array variables (total, maybe uninitialized and percentage), compilation time (total and the used-before-set phase) and execution slowdown.

			Scalar			Array			Compilation		
Bench	**Line**	**Mod**	**Tot**	**May**	**Percen**	**Tot**	**May**	**Percen**	**Total**	**UBS**	**Slowdown**
tomcatv	190	1	24	0	0.00%	9	5	55.56%	0:08	0:01	4.42
swim	429	6	42	0	0.00%	14	13	92.86%	0:15	0:01	4.19
su2cor	2332	35	276	12	4.35%	118	63	53.39%	5:35	0:02	5.43
hydro2d	4292	42	226	11	4.87%	34	7	20.59%	1:47	0:05	5.07
mgrid	484	12	49	0	0.00%	10	4	40.00%	1:50	0:14	3.77
applu	3868	16	200	1	0.50%	33	10	30.30%	20:48	1:32	6.06
turb3d	2101	23	246	14	5.69%	32	31	96.88%	2:03	0:17	6.41
apsi	7361	96	1035	125	12.08%	19	11	57.89%	21:05	0:05	1.06
fpppp	2784	38	919	331	36.02%	40	26	65.00%	6:39	0:29	12.92
wave5	7764	105	1192	74	6.21%	162	33	20.37%	26:46	2:95	UBS

benchmark. We report the total numbers of scalar and array variables (Columns 4 and 7), the numbers of maybe uninitialized variables detected by the static analysis (Columns 5 and 8) and the corresponding percentages (Columns 6 and 9). On average, the percentages of maybe uninitialized variables to be checked at run-time are 3.1% for scalar variables and 37.96% for array variables. No used-before-set error is detected at compile-time, which is expected for benchmarks. All scalar variables in *tomcatv, swim* and *mgrid* are proved to be well initialized. One initialization and several verifications (single tests for scalar variables and loops for arrays) are added for each maybe uninitialized variable.

Column 10 shows the total compilation time (in minutes and seconds) required by PIPS to parse, compute imported array regions, analyze and generate code with used-before-set checks. The used-before-set analysis phase only takes a very small fraction of this compilation time, which is shown in Column 11. These times are measured on a UltraSparc II 440MHz, 256 Mo RAM. The code instrumented with PIPS initializations and verifications is then compiled with the SUN Workshop F77 version 5.0 compiler to generate executable files. This experiment is reported with the optimizing options turned on, using the SPEC95 CFP measurement guidelines (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch). Uninitialized variables are detected in *wave5*. In subroutine `PARTBL`, the local and static variables `LCMMAX` and `LCMR` are used before initialization.

The execution time slowdown with the standard input data for other SPEC95 CFP benchmarks is shown in the last column. We did not measure the slowdown when all references have to be instrumented, without help of static analysis, to show the contribution of the combined analysis, because we think that the slowdown given in this paper is more important in order to compare with the other work. On average, the instrumented code is 5.48 times slower than the initial code. There is only a 6% overhead for *apsi*. The overhead is rather high for *fpppp* (about 13 times) because of irreducible control flow graphs in this benchmark. Information is lost and the approximation of array regions becomes `MAY`.

It corresponds to 36.02% of checked scalar variables and 65% of checked array variables . To improve the results, we can use a more sophisticated treatment on irreducible control flow graphs when computing array regions. On the other hand, some program transformations are needed for several benchmarks. For instance, we can reduce the total number of variables to be checked from 18 to 14 for *hydro2d* by cloning the subroutine `ADLEN` which has two totally different behaviors for two parameter values: "half" and "full" steps. This optimization makes the array region analysis more precise, since interprocedural information helps to narrow down the scope of possible effects of the called procedure.

Other solutions are used in different compilers such as the undefined bit pattern or the NaN floating point trap, which can reduce greatly the execution time overheads. The verification on real variables can be omitted and replaced by the NaN exception trap. However, the origin of the uninitialized errors is not easy to locate with this method. Since the primary objective of this work is to reduce the number of possibly undefined variables by using static analysis, we do not intend to implement these techniques at the moment. Experimental results show that SPEC95 CFP are in general well-debugged programs. Used-before-set errors have been found in only one benchmark with its given input.

6 Conclusion

Static and dynamic analyses complement each other. Static analysis can discover automatically run-time errors and reduce the instrumentation or debugging cost. Dynamic checking takes into account program control flows and real input data that sometimes make static checking completely ineffective. Our used-before-set analysis combines these two approaches in order to reduce the overall cost while assuring the correctness of program.

PIPS is a source-to-source compiler that can be used for program analyses, transformation, parallelization and verification. By reusing advanced interprocedural analyses, the verification task becomes more efficient. `READ` and `WRITE` regions are already used to analyze program dependencies. `IN` regions are exploited for program optimizations such as array privatization, compile-time optimization of local memory or cache behavior in hierarchical memory machines, etc. Their precision is improved to target the verification. Only about 600 additional lines of C code are needed to implement the used-before-set analysis phase.

By using the `IN` region analysis, the static phase can improve one's confidence of the program correctness by showing that the program is free from used-before-set errors. Or, an error can be detected statically and the bug can be fixed right after the analysis. A small number of maybe uninitialized variables pointed by our compile-time analysis can help the testing and validation process to save debugging time. Run-time checks are generated only when information is not available to monitor the verification process. When executing the code, if a used-before-set error happens, the message error provides information about what occurred prior to the error, which can be of great help when trying to identify the fragment of code that actually caused the error. Furthermore, we could

have an appropriate exception handling mechanism which is very important for safety-critical systems.

Experimental results also show that poor code quality can make static analysis insufficient; run-time checks remain and run-time failures cannot be eliminated. In addition to SPEC95 CFP suite, our analysis is applied to some large scale industrial applications to enhance the debugging process. We have encountered other problems with used-before-set checking, e.g the type mismatch. The actual variable is of integer type and the corresponding formal variable is declared of real type. In the called procedure, we verify if the formal variable is initialized by a NaN value check, which is in fact a check on integer variable and this may give false results. In addition, the compiler and platform dependences of initialization and verification functions are also implementation problems.

To obtain better results with static analysis, we are planning to improve the accuracy of `IN` array regions on an arbitrary control flow graph by using a more precise analysis, based on the control flow graph restructuring algorithm of Bourdoncle [23]. In addition, other approaches of array analysis such as input effects of Leservot [19] can give the exact statement where the used-before-set error occurs. Or, as in [24], the list of complementary array sections can be kept when performing some convex operators on array regions, in order to have more precise analysis on array usage. The source function [8] can also be applied to the used-before-set checking problem. The question is to study the trade-off between precision and summarization, as well as the complexity in space and time. Our method can be applied to other programming languages, with appropriate language construct handling. For example, the procedure call recursion can be handled with fixed point analysis on the call graph when computing imported array regions. Other problems such as pointer analysis must be studied to improve the precision of the static analysis in order to have an effective combined approach. The PIPS software and documentation as well as the used before set checking are available on *http://www.cri.ensmp.fr/ pips*.

References

1. Steensgaard, B.: Points-to analysis in almost linear time. In: ACM Symposium on Principles of Programming Languages, (1996) 32–41
2. Nguyen, T.V.N.: Efficient and Effective Software Verifications for Scientific Applications using Static Analyses and Code Instrumentation. PhD thesis, Ecole des Mines de Paris (2002)
3. Loginov, A., Yong, S.H., Horwitz, S., and Reps, T.W.: Debugging via run-time type checking. In Fundamental Approaches to Software Engineering (2001) 217–232
4. Appleyard, J.: Comparing Fortran compilers. ACM SIGPLAN – Fortran Forum **20** (2001) 6–10
5. Hasting, R., Joyce, B.: Purify: fast detection of memory leaks and access errors. In: Winter USENIX Conference (1992) 125–136
6. Evans, D., Guttag, J., Horning, J., Tan, Y.M.: LCLint: A tool for using specifications to check code. In: ACM SIGSOFT Symposium on Foundations of Software Engineering (1994) 87–96

7. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: ACM Symposium on Principles of Programming Languages (1995) 49–61
8. Feautrier, P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming **20** (1991) 23–53
9. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: an overview of the PIPS project. In: International Conference on Supercomputing (1991) 144–151
10. Creusillet, B., Irigoin, F.: Interprocedural array region analyses. In: International Workshop on Languages and Compilers for Parallel Computing. Volume 1033 of Lecture Notes in Computer Science, Springer-Verlag (1995) 46–60
11. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons, Chichester (1986).
12. Creusillet, B.: IN and OUT array region analyses. In: Workshop on Compilers for Parallel Computers. (1995) 233–246
13. ANSI: Programming Language FORTRAN, ANSI X3.9-1978, ISO 1539-1980. American National Standard Institute, New York (1983).
14. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical data flow framework for array reference analysis and its application in optimization. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (1993) 68–77
15. Hall, M.W., Amarasinghe, S.P., Murphy, B.R., Liao, S.W., Lam, M.S.: Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. Super-Computing (1995)
16. Gu, J., Li, Z., Lee, G.: Symbolic array dataflow analysis for array privatization and program parallelization. In: Supercomputing (1995)
17. Tu, P., Padua, D.A.: Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In: International Conference on Supercomputing (1995)414–423
18. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. In: ACM Symposium on Principles of Programming Languages (1995) 37–48
19. Leservot, A.: Analyses interprocédurales du flot des données. PhD thesis, Université Paris VI (1996)
20. Triolet, R., Feautrier, P., Irigoin, F.: Automatic parallelization of Fortran programs in the presence of procedure calls. In: European Symposium on Programming (1986)
21. Ancourt, C., Irigoin, F.: Scanning polyhedra with DO loops. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (1991) 39–50
22. Dujmovic, J.J., Dujmovic, I.: Evolution and evaluation of SPEC benchmarks. ACM SIGMETRICS **26** (1998) 2–9
23. Bourdoncle, F.: Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite. PhD thesis, Ecole Polytechnique, France (1992)
24. Manjunathaiah, M., Nicole, D.A.: Precise analysis of array usage in scientific programs. Scientific Programming **6** (1997) 229–242
25. Ami, T.L., Reps, T., Sagiv, L., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: International Symposium on Software Testing and Analysis, (2000) 26–38
26. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2001) 168–179