Interprocedural Program Analyses for Efficient Array Bound Checking

Thi Viet Nga NGUYEN
François IRIGOIN
Ecole des Mines de Paris

Array bound violations occur when accessing array elements with indexes that are out of the array declared ranges in a specified program unit. *Interprocedural array bound violations* happen when the size of an actual array argument is exceeded through procedure calls. Both intraprocedural and interprocedural range checking are critical for software verification and validation. Commercial implementations are currently limited to intraprocedural checking and are not really fulfilling user expectations for execution speed and/or information about the violations.

In this paper, we study how interprocedural program analyses can be used to improve the elimination of unnecessary bound checking within procedures. Instead of designing a new specific algorithm, we implemented two algorithms representative of the main published approaches for *intraprocedural array bound checking* by re-using available *interprocedural analysis techniques* designed for automatic parallelization and code optimization. The first algorithm is based on redundant bound check elimination and the second one is based on insertion of unavoidable tests.

We also show how an *interprocedural array bound checker* can be seen as a whole program transformation. This is not covered in other studies. Program analysis helps software verification by guaranteeing automatically the correctness of program, detecting statically real array access errors and reducing the overhead of run-time array range checks. Not only compile-time and run-time performance but also debugging capability are important criterias to evaluate different compilers. Experiments with our array bound checkers and other compilers on three different platforms for SPEC95 CFP benchmarks show improvements in execution times, compilation times and information available about violations when they occur dynamically.

Categories and Subject Descriptors: 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*

General Terms: Algorithms, Performance, Verification

Additional Key Words and Phrases: Intraprocedural analysis, interprocedural analysis, range checking, array bound checking

---

This article is a revised version of a paper presented at the Second International Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), May 2001, expanded with a new contribution about the interprocedural array bound checking.
Address: Centre de Recherche en Informatique, Ecole des Mines de Paris, 35 rue Saint Honoré, 77305 Fontainebleau Cedex, France; email: nguyen@cri.ensmp.fr, irigoin@cri.ensmp.fr

1. INTRODUCTION

Array bound checking refers to determining whether all array references in a program are within their declared ranges. These array bound checks may be analyzed intraprocedurally or interprocedurally, depending on the need for safety. Such checking is desirable for any program, regardless of the programming language used, since bound violations are among the most common programming errors.

Subscripting arrays beyond their declared sizes may result in unexpected results, security holes or failures. For the safety of execution, some languages such as Java require that a program only be allowed to access elements of an array that are part of the defined extent of the array.

But on the other hand, bound checking can be very expensive because every array access must be preceded by two bound checks per dimension to assert the legality of the access. This increases the size of the executable file, the compilation time and the execution time. In fact, these two bound checks can be implemented as one unsigned comparison instruction but the overhead still remains, in part because other code transformations or optimizations are prevented by these bound checks.

Many compilers for other languages such as Pascal and Fortran solve this problem by providing the user with a compile-time option to enable or disable the checking. The purpose of this option is to allow users to enable checking in the development and debugging runs of the program, and then, once all the defects are supposedly found and fixed, to turn it off for the production version.

However, all software engineering studies of defects in programs indicate that versions of systems delivered to customers are likely to have bugs that were not even observed during testing phases. Users are nowadays more motivated by safety, and this approach to bound checking, therefore, is not highly appreciated. Rather, bound checking is just as important for delivered versions of programs as for development versions. Instead of providing a way to turn bound checking off, what is needed is to optimize it so that it has a minimal overall cost.

The objective of this work is to see if it is possible to perform efficient range checking by reusing interprocedural analysis techniques already implemented in commercial compilers. Many routines are written to manipulate arrays of arbitrary size, but are used in actual programs only on arrays whose sizes are determined in the main program. So interprocedural analyses that propagate information through procedure boundaries should allow us to eliminate more unnecessary bounds checking and may result in significant speedups. Moreover, interprocedural translation helps to prove the absence of interprocedural array bound violations or to detect them at either compile-time or run-time.

The paper is organized as follows. Section 2 discusses the related work on optimization of array range checking and explains what is missing in these results. An overview of PIPS [Irigoin et al. 1991], our research parallelizing compiler, and its existing analyses used in the following sections are briefly described in Section 3. Section 4 presents our first intraprocedural bound checking approach based on the elimination of redundant tests. Full information about the location of bound violation is preserved. Section 5 presents the second approach, based on the insertion of unavoidable tests. The precise location of the violation is lost but the array improperly accessed is known. Section 6 describes the interprocedural array bound checking. Results obtained with our intra- and inter-procedural techniques are reported and compared to three commercial compilers in Section 7. Conclusions are given in the last section.

2. RELATED WORK

Suzuki and Ishihata [1977] implemented a system that inserts logical assertions before array element accesses and then uses theorem proving techniques to verify the absence of array range violations. Such techniques are often expensive and are restricted to programs written in a structured manner, i.e without goto statements.

Another approach was developed by Markstein, Cocke, and Markstein [1982], M.Asuru [1992], Gupta [1990, 1993], Spezialetti and Gupta [1995], Kolte and Wolfe [1995]. They introduce algorithms to reduce the execution overhead of range checks through the elimination and propagation of bound checks by using data flow analysis. Their techniques became more and more sophisticated in order to improve results. In Gupta [1993], a bound check that is identical or subsumed by other bound checks is suppressed in a local elimination. In a global elimination, an algorithm first modifies bound checks to create additional redundant checks and then carries out the elimination of redundant checks by using the notions of *available* and *very busy* checks. These two algorithms use backward and forward data flow analyses to solve the problems. For the propagation of checks out of loops, R. Gupta identifies candidates for propagation which include invariants, loops with increment or decrement of one and variables of increasing or decreasing values. Then he uses check hoisting to move checks out of loops. However, the results are not very convincing because of the small size of his examples and because the optimizations have only been applied by hand.

In their article, Kolte and Wolfe [1995] rely on a check implication graph where nodes are sets of range checks in canonical form and edges denote implications between these families. Like [Gupta 1993], they also compute the *available* and *anticipatable* checks by solving forward and backward data flow problems. To create more redundant checks, there are five schemes to insert checks at safe and profitable program points: *no-insertion, safe-earliest, latest-not-isolated placement, check-strengthening and preheader insertion*. They use partial redundancy elimination after determining the most efficient places to move bound checks to.

Their implementation in the Fortran compiler Nascent [Kolte and Wolfe 1995] with different techniques in range check optimization leads to experimental results that show the necessity of range check optimization, the effectiveness and cost of these optimizations. However, high percentages, even 99.99%, of eliminated tests do not always mean faster execution times. This article lacks comparisons between the execution times of codes with and without optimized bound checks to show the impact of removed checks. Furthermore, there are no mentions of bound violations in PerfectClub (*mdg, spc77, trfd*) and Riceps benchmarks (*linpackd*) which are caused by formal array dimension declared 1.

The abstract interpretation approach proposed by Cousot and Cousot [1976], Cousot and Halbwachs [1978] considers array range checking as an example of the automated verification of execution properties of programs. Like [Schwarz et al. 1988], they use static data flow analysis information to prove at compile-time that an array bound violation cannot occur at run-time and that the test for this violation is unnecessary. Their algorithms for propagating and combining assertions depend on the different rules they use. Since the algorithms in the abstract interpretation and the program verification approaches do not perform any insertion of checks in the program to create more redundant checks, they could only take advantage of completely redundant checks. So the run-time overhead of the partial redundant checks that cannot be evaluated at compile-time still remains. Also present in this approach is the model checking group [Delzanno et al. 2000] who uses fix point acceleration techniques to help the automated verification of programs.

Other articles by Midkiff, Moreira, Snir and M. Gupta [P.Midkiff et al. 1998; Moreira et al. 2000] describe another approach to optimize array reference checking in Java programs based on code replication. All the optimizations are based on partitioning a loop nest, seen as an iteration space, into regions with different access violation characteristics. In unsafe regions, run-time tests are performed, whereas in other regions they are not necessary because all array indices are guaranteed to be within bounds. The optimizations differ on their level of refinement and practicality. These techniques are less complicated than the abstract interpretation approach while still being effective. However, they do not use any control-flow analysis to reduce code replication, and the optimizations here are mainly for Java applications, because of its precise exception semantics. Another approach for Java, based on an extended Static Single-Assignment graph representation, the Eliminating Array Bound Checks on Demand in [Bodik et al. 2000] can remove about 45% of dynamic bound checks of a representative set of Java programs.

Although there are many different techniques for array bound checking optimization, we can partition them into two main approaches. The first approach puts array bound checks at every array reference and removes a check if it is redundant [Markstein et al. 1982; Gupta 1990; M.Asuru 1992; Gupta 1993; Kolte and Wolfe 1995]. In the second approach, array bound checks are put at places where it is not possible to prove them useless [Suzuki and Ishihata 1977; P.Midkiff et al. 1998; Schwarz et al. 1988; Cousot and Cousot 1976; Cousot and Halbwachs 1978; Cousot 1990].

The first approach attempts to reduce the dynamic and static numbers of bound tests and the overhead induced by a test even if it cannot be eliminated. This is done by determining if a test is subsumed by another test, so that it can be eliminated. Hoisting range checks out of loops is also applied when it is possible. The analyses are simple or sophisticated depending on each technique. The question here is: is it worth performing complicated range check optimizations when the hoisting of array bound checks out from the innermost loop may be sufficient?

In the second approach, by using data flow information, if it is proven that no array bound violation will occur at run-time in some region of code, tests are unnecessary for this region. If it is proven that an access violation might occur, tests are generated as needed. The number of generated tests is limited; range checks are put only where there might be bound violations. But the difficulty of this approach is that the information needed to prove that no violation will occur may not be available at compile-time. Then tests may remain inside inner loops.

In addition, the amount of information about the violation is never discussed. It is often reduced to "a violation occured" with no information about the array accessed nor the statement where the array element was referenced. Especially with code hoisting, the information cannot always be preserved.

So both approaches have advantages and drawbacks when comparing the number of needed transformations and analyses as well as information about array violation. A goal of our work here is to compare the effectiveness and optimization costs of two different algorithms for intraprocedural array bound checking. The first one is based on test elimination without hoisting, and the second one is based on optimized test insertion without code replication.

To complete array range checking, we introduce in this paper an interprocedural phase to prove the absence of or to detect interprocedural bound violations. This kind of out-of-bound error occurs when the size of a formal array parameter exceeds that of its associated actual array parameter. If sufficient information is not available, run-time checks are generated before each call site to guarantee the program correctness. This interprocedural array bound check analysis is neither addressed in other work nor in commercial compilers. The three algorithms, two intraprocedural and one interprocedural, were implemented in PIPS, which is described in the next section.

3. PIPS OVERVIEW

PIPS is an interprocedural parallelizing tool designed on top of a database to avoid global recompilation whenever possible. Each analysis is performed only once on each procedure and produces a summary result that is later used at call sites. Forward and backward analyses are controlled by a *make*-file mechanism (*pipsmake*) which makes sure that necessary information is available before a new analysis phase is started.

PIPS consists of several analysis phases dealing with call graph computation, dependences, transformers, preconditions, use-def chains, array regions and of program transformations such as loop transformations, constant folding, dead code elimination. The running example in Figure 1 is used to illustrate three important analyses (transformers, preconditions, array regions) and our two intraprocedural bound check optimizers. This example is extracted from the program *swim*, a weather prediction program in the SPEC95 CFP benchmark with procedure calls and array references, which are used to show the effect of interprocedural analyses.

3.1 Transformers

A transformer abstracts the effects of a program statement upon the values of integer scalar variables by giving an affine approximation of the relations that exist between their values before and after the execution of the statement. Transformers are computed from elementary instructions to compound instructions such as basic blocks, tests, loops. To deal with test statements, we first build a predicate that is the transformer of the true branch, combined with the test condition if this test is a convex polyhedron [Cousot and Halbwachs 1978]. Another predicate is the transformer of the false branch, added with the negation of the test condition if it is a convex polyhedron. The transformer of the test statement is the convex hull of these two predicates. To handle loops, different kinds of fixed points are needed in order to provide a flexible choice between efficiency and precision. Fast computation uses primitive, non-iterative fixed point algorithm based on the transition function associated to the loop body and which only handles equations. Full computation

```
      PROGRAM SHALOW
      PARAMETER (N1=513, N2=513)
      COMMON U(N1,N2), UNEW(N1,N2)
      COMMON /CONS/ M,N
      CALL INITAL
      MN = MIN(M, N)
      UCHECK = 0.0
      DO 35 I = 1, MN
         DO 35 J = 1, MN
            UCHECK = UCHECK+ABS(UNEW(I,J))
   35 CONTINUE
      END

      SUBROUTINE INITAL
      PARAMETER (N1=513, N2=513)
      COMMON U(N1,N2), UNEW(N1,N2)
      COMMON /CONS/ M,N
      READ (5, *) M, N
      U(1,N+1) = U(M+1,1)
      END
```

a) Initial code

```
...
C T(MN) {MN<=M, MN<=N}
C P(M,N) {}
  MN = MIN(M, N)
C T() {}
C P(M,MN,N) {MN<=M, MN<=N}
  UCHECK = 0.0
C T(I,J) {1<=I}
C P(M,MN,N) {MN<=M, MN<=N}
  DO 35 I = 1, MN
C T(J) {1<=J}
C P(I,J,M,MN,N) {1<=I, I<=MN, MN<=M, MN<=N}
     DO 35 J = 1, MN
C T() {}
C P(I,J,M,MN,N){1<=I,I<=MN,1<=J,J<=MN,MN<=M,MN<=N}
        UCHECK=UCHECK+ABS(UNEW(I,J))
...
```

b) Code with transformers (T) and preconditions (P)

Fig. 1. Running example from benchmark *swim*, SPEC95 CFP

uses a *derivative* fixed point operator based on finite differences. The loop body transformer on variable values is projected onto their finite differences. Invariants, both equations and inequalities, are directly deduced from the constraints on the differences and after integration. Unstructured statements [Irigoin et al. 1991] are also handled in PIPS and can be accurately analyzed with non-iterative algorithm for fixed points.

A summary transformer of a procedure is the transformer computed for the procedure body after projecting its local variables. An interprocedural propagation of transformers is realized by traversing procedures in the *reverse invocation order*, which processes a procedure after its callees. Each time a procedure is called, its summary transformer is translated to the frame of the calling procedure to provide the transformer of the procedure call. This summary information is available and updated in the database thanks to the pipsmake mechanism. The translation into the calling frame uses the global variables information and the bindings between actual and formal parameters.

3.2 Preconditions

This analysis tries to discover the constraints holding among variables of the program. Preconditions are affine predicates over scalar integer variable that hold just before the execution of the corresponding statement. They are propagated from the module entry point down to the abstract syntax tree leaves. Transformers, which are computed by a previous phase, are applied to preconditions to obtain postconditions, which usually are the preconditions of the following statements.

The initial precondition of a procedure in the intraprocedural analysis is derived from `DATA` or `PARAMETER` statements or, if no information is available, we use an empty precondition `P() {}` that represents all possible values. In the interprocedural analysis that is discussed below, the initial precondition of a procedure is derived from its calling contexts. As for transformers, the postcondition of a test statement is the convex hull of two predicates that are propagated along the true and false branches. The accuracy of postconditions for loops and unstructured statements also depends on the type of fixed point operators used. The fastest operator is the projection along all variables that are used in the loop body and iterator or in the unstructured statement. The most accurate solution is the derivative one, because it handles both equalities and inequalities.

The interprocedural analysis of preconditions is performed in the *invocation order*, which processes a procedure before all its callees. Each time a procedure is invoked, the precondition of the current call site is available and is translated to the frame of the called procedure. The summary precondition of the called procedure is then replaced by its convex hull with the new translated precondition.

One main advantage of transformer and precondition analyses is that we can deduce from the program semantics the information that is not stated explicitly. In fact, transformers and preconditions are powerful symbolic analyses that abstract relations between program states with polyhedra, and encompass most standard interprocedural constant propagation as well as interval analyses.

3.3 Array Regions

Array region analysis collects information about the way array elements are used and defined by programs. Different representations of array element sets such as convex polyhedra, regular section descriptors, data access descriptors, guarded regular section descriptors are introduced in [Triolet et al. 1986; Callahan and Kennedy 1988; Feautrier 1991; Maydan et al. 1992; Gu and Li 2000]. As defined in PIPS [Creusillet and Irigoin 1995; Irigoin 1993], a *convex array region* is a set of array elements described by a convex polyhedron containing affine equalities and inequalities. These constraints link the region parameters that represent the array dimensions to the value of the program integer scalar variables (i.e `{PHI1==I, PHI2==J}` is the region of an array reference `A(I,J)` where the region parameters `PHI1` and `PHI2` respectively represent the first and second dimensions of `A`).

A region has the approximation `MUST` if every element in the region is certainly accessed, and the approximation `MAY` if its elements are simply potentially accessed. It is useful to distinguish the `MUST` versus `MAY` information because it tells us whether a property must or may hold, and hence can be relied upon or not. The approximation of a region is `EXACT` if the region exactly represents the requested set of array elements. `MAY` and `MUST`, respectively are an over- and under-approximation of `EXACT`: $\texttt{MUST} \subseteq \texttt{EXACT} \subseteq \texttt{MAY}$.

Since array region analysis is introduced to support dependence analyses on array structures, two kinds of effects on array elements are used: *read* region if they are used and *write* region if they are defined.

Regions are built bottom-up from the deepest nodes to the largest compound statement nodes in the *hierarchical control flow graph* [Irigoin et al. 1991]. It means that at each meet point of a control flow graph, the region information from different control branches are merged with a convex hull operator. The approximation of regions is conservative. Intraprocedural region analysis has to deal with statements like assignments, basic blocks, tests, loops and control flow graphs.

The summary region of a procedure is computed by masking local effects from the region for the procedure body. However, the interprocedural analysis of array regions is more complicated than that of transformers and preconditions. Summary region is translated from the callee's name space into the caller's name space at each call site but array reshaping and the lack of non-linear expression analysis can cause lost of accuracy during the translation.

4. ELIMINATION OF REDUNDANT TESTS

Our first implementation of an intraprocedural range check optimizer consists of two phases: *generation of bound checks* and *partial redundancy elimination*

In essence, a *partial redundancy* [Muchnick 1997] is a computation that is performed more than once on some path through a flow graph, i.e., some path through the flow graph contains a point at which the given computation has already been computed and will be computed again. To eliminate redundant bound checks, we use information provided by the preconditions in PIPS. Preconditions, seen as invariant assertions, are very useful to detect bound violations or to remove redundant checks. The algorithm Elimination of redundant tests works as follows:

(1) Generate non-trivial bound checks for every statement that has array references. Each bound check is accompanied with a stop message. If a bound violation is detected, the message tells the user in which array, on which dimension, and in which line the subscript is out of range. The test for the i-th dimension of array reference $A(..., s_i, ...)$ is: `IF` $((s_i.\mathtt{LT}.l_i).\mathtt{OR}.(s_i.\mathtt{GT}.u_i))$ `STOP` *message*, where A is declared as $A(..., l_i : u_i, ...)$.

(2) Compute transformers and preconditions for the new code with two options:
—Intraprocedural transformer and precondition analyses;
—Interprocedural transformer and precondition analyses;

(3) For each bound check, test the feasibility of the system built from the precondition of the current statement and the bound check. The feasibility test of a system of constraints is implemented in PIPS by using the linear programming Simplex and Fourier-Motzkin algorithms for integer and rational coefficients [Schrijver 1986].
   (a) If the bound check is true with respect to the precondition, a bound violation is detected at compile-time;
   (b) If the system is infeasible, the bound check is false and is removed;
   (c) Otherwise, the bound check is preserved;

Figures (2a,2b) and Figures (2a,2c) illustrate this approach with the running example in Figure 1 by using respectively, the intraprocedural and interprocedural options. The left hand side part of each figure is the code after the generation of bound checks. Preconditions of the new code are in comment lines. Some trivial checks that are never true such as `1.LT.1`, `1.GT.513`, or `N-1.GT.N` are not generated by our bound checker. In both options, the preconditions `{1<=I, 1<=J}` after entering the loop in `SHALOW` allow us to remove all lower bound checks. The difference between the intraprocedural and the interprocedural options of transformers and preconditions is that in Figure 2b, after the call to `INITAL`, we have an empty postcondition while, in Figure 2c, we have `P(M,N) {0<=M, M<=512, 0<=N, N<=512}`.

```
  SUBROUTINE INITAL
  READ (5, *) M, N
C P(M,N) {}
  IF (M+1.LT.1.OR.M+1.GT.N1) STOP
     "Violation:1st dim,array U,line 19"
C P(M,N) {0<=M, M<=512}
  IF (N+1.LT.1.OR.N+1.GT.N2) STOP
     "Violation:2nd dim,array U,line 19"
C P(M,N) {0<=M, M<=512, 0<=N, N<=512}
  U(1,N+1) = U(M+1,1)
  END
```

```
SUBROUTINE INITAL
READ (5, *) M, N
IF (M+1.LT.1.OR.M+1.GT.N1) STOP
   "Violation:1st dim,array U,line 19"
IF (N+1.LT.1.OR.N+1.GT.N2) STOP
   "Violation:2nd dim,array U,line 19"
U(1,N+1) = U(M+1,1)
END
```

a) Intraprocedural and interprocedural options for INITIAL

```
  PROGRAM SHALOW
C P() {}
  CALL INITAL
C P(M,N) {}
  MN = MIN(M, N)
C P(M,MN,N) {MN<=M, MN<=N}
  UCHECK = 0.0
C P(M,MN,N) {MN<=M, MN<=N}
  DO 35 I = 1, MN
C P(I,J,M,MN,N) {1<=I,I<=MN, N<=M,MN<=N}
     DO 35 J = 1, MN
C P(I,J,M,MN,N) {1<=I,I<=MN,1<=J,J<=MN,MN<=M,MN<=N}
        IF (J.LT.1.OR.J.GT.N2) STOP
        "Violation:2nd dim,array UNEW,line 10"
C P(I,J,M,MN,N) {1<=I,I<=MN,1<=J,J<=513,J<=MN,MN<=M,MN<=N}
        IF (I.LT.1.OR.I.GT.N1) STOP
        "Violation:1st dim,array UNEW,line 10"
C P(I,J,M,MN,N) {1<=I,I<=513,I<=MN,1<=J,J<=513,J<=MN, MN<=M,MN<=N}
        UCHECK=UCHECK+ABS(UNEW(I,J))
35 CONTINUE
  END
```

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
UCHECK = 0.0
DO 35 I = 1, MN
   DO 35 J = 1, MN
      IF (J.GT.N2) STOP
      "Violation:2nd dim,array UNEW,line 10"
      IF (I.GT.N1) STOP
      "Violation:1st dim,array UNEW,line 10"
      UCHECK=UCHECK+ABS(UNEW(I,J))
35 CONTINUE
   END
```

b) Intraprocedural option for SHALOW

```
  PROGRAM SHALOW
C P() {}
  CALL INITAL
C P(M,N) {0<=M, M<=512, 0<=N, N<=512}
  MN = MIN(M, N)
C P(M,MN,N) {0<=M,M<=512,MN<=M,MN<=N,0<=N,N<=512}
  UCHECK = 0.0
C P(M,MN,N) {0<=M,M<=512,MN<=M,MN<=N,0<=N,N<=512}
  DO 35 I = 1, MN
C P(I,J,M,MN,N) {1<=I,I<=MN,0<=M,M<=512,MN<=M,MN<=N,0<=N,N<=512}
     DO 35 J = 1, MN
C P(I,J,M,MN,N) {1<=I,I<=MN,1<=J,J<=MN,0<=M,M<=512,MN<=M,MN<=N,0<=N,N<=512}
        IF (J.LT.1.OR.J.GT.N2) STOP
C P(I,J,M,MN,N) {1<=I,I<=MN,1<=J,J<=MN,M<=512,MN<=M,MN<=N,N<=512}
        IF (I.LT.1.OR.I.GT.N1) STOP
C P(I,J,M,MN,N) {1<=I,I<=MN,1<=J,J<=MN,M<=512,MN<=M,MN<=N,N<=512}
        UCHECK=UCHECK+ABS(UNEW(I,J))
35 CONTINUE
  END
```

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
UCHECK = 0.0
DO 35 I = 1, MN
   DO 35 J = 1, MN
      UCHECK=UCHECK+ABS(UNEW(I,J))
35 CONTINUE
END
```

c) Interprocedural option for SHALOW

Fig. 2. Running example with Elimination of redundant tests (declarations omitted - see Fig. 1)

```
   COMMON ITAB(10),J
   REAL A(10)
C <A(PHI1)-W-EXACT-{PHI1==11}>
C <ITAB(PHI1)-W-MAY-{1<=PHI1}>
   READ *,M
   J = 11
C <ITAB(PHI1)-W-EXACT-{1<=PHI1, PHI1<=M, J==11}>
   DO I = 1,M
C <ITAB(PHI1)-W-EXACT-{PHI1==I, J==11, 1<=I, I<=M}>
      ITAB(I) = 1
   ENDDO
C <A(PHI1)-W-EXACT-{PHI1==J, J==11, 1+M<=I, 1<=I}>
   A(J) = 0
   a) Code with array regions
```

```
COMMON ITAB(10),J
REAL A(10)
READ *,M
IF (11.LE.M) STOP
   "Violation:1st dim,array ITAB"
STOP "Violation:1st dim,array A"
J = 11
DO I = 1,M
   ITAB(I) = 1
ENDDO
A(J) = 0

b) Code with unavoidable tests
```

Fig. 3. Array regions with incorrect code

This more exact postcondition helps to eliminate all upper bound checks (`I.GT.N1` and `J.GT.N2`) in the nested loop in `SHALOW`. Here we see the strength of the interprocedural precondition analysis because in the intraprocedural option, bound checks are left inside loops, as long as code hoisting has not been applied. However, for the subroutine `INITAL`, since we have no information about variables `M` and `N`, no test is removed.

After the partial redundancy elimination transformation, the number of generated tests is greatly reduced. PIPS translates Fortran programs into instrumented Fortran codes with bound checks which are then compiled and executed using their standard input data sets. The experimental results with the benchmark SPEC95 CFP are given in Section 7.

5. INSERTION OF UNAVOIDABLE TESTS

The second intraprocedural array bound checker is based on the array region analysis phase. The array region information as well as other analyses in PIPS are computed under the assumption that the code is correct. In the first approach, Elimination of redundant tests, bound checks are generated before applying other analyses. Array accesses are guaranteed to be within their bounds and transformations applied on this instrumented code are always safe. In the second approach, the insertion of unavoidable tests is based directly on analyses computed for the input code. The small example in Figure 3a shows how impact of the correct code assumption can lead to a false region of array `A`. As a bound violation in `ITAB` can modify the value of `J` (when $\texttt{M} \geq \texttt{11}$), the region for `A` is not correct any more and there is no overflow in array `A` but in `ITAB`. To cope with this problem, our analysis is based on the insight that it is safe to propagate an array region from a program point p_2 up to an earlier point p_1 if and only if on every execution path from p_1 to p_2, any written reference to any array is inside the declared range. In other words, an array region at point p_1 is said *safe to be used* if and only if all written array references before p_2 are checked. Only written references are taken into account because read references do not modify memory locations, so they have no effects on the correctness of the array region computation. The write order is used to decide which array region is checked firstly. We know that the region for array `A` becomes false only when an element outside the declared range of `ITAB` is written. So if bound checks for `ITAB` are always generated before testing regions of `A`, there is no problem. The code with unavoidable tests is shown in Figure 3b. This is very similar to code hoisting; we are only allowed to hoist code up to an earlier point if it is safe doing so. A complete correctness proof of our algorithm is given in [Nguyen and Irigoin 2001].

As mentioned above, regions are built bottom-up, from the elementary statements to the compound statements. Our analysis is a top-down analysis: it begins with the largest compound statement and if we have the answer about array element accesses for this statement, we do not have to go down into its substatements. The analysis can stop here, and bound checks can be inserted at the very beginning of the module entry and outside loops if we have sufficient information. The algorithm consists of two phases: *array region computation* and *insertion of unavoidable tests*.

To compute array regions, since transformers are used in the array region analysis to model the effects of state transitions and preconditions are used to improve the accuracy of over-approximated regions by filtering out some unreachable states, we have choices between two options:

—Intraprocedural transformer and precondition analyses;

—Interprocedural transformer and precondition analyses;

The insertion of unavoidable array bound checks works as follows:

At a compound statement, the read and write regions of each array are used to test the feasibility of the corresponding array bound checks. The regions of an array are considered only when every array whose assignment occurs at least once before another assigment on the concerning array has been already checked. In case the write order cannot be established for this compound statement, we have to go down to the substatements of the current statement (for instance, with the sequence $(s_1; s_2; s_3)$: `A(I) = ;` (s_1) `B(M,N) = ;` (s_2) `A(J) = ;` (s_3), neither `A` nor `B` is always written before the other, we have to go down and check the array region for `A` at (s_1), array region for `B` and then for `A` at $(s_2; s_3)$). Otherwise, for each array dimension i, we build two systems: s_1 from the array region and the lower bound check $\texttt{PHI}_i$ `<` l_i, and s_2 from the array region and the upper bound check $\texttt{PHI}_i$ `>` u_i. Then,

(1) If the region is a `MAY` region included in the declared dimensions of the array (s_1 and s_2 are infeasible), no bound check is needed for the compound statement and we stop the process for the array here;

(2) If the region is a `MUST` region that contains elements which are outside the declared dimensions of the array (at least one of the two systems s_1 or s_2 is feasible), there is certainly a bound violation. An error is detected at compile-time;

(3) If the region is an `EXACT` region and it is possible to project all unnecessary variables `PHI` from the systems s_1 and s_2, we have unavoidable tests to insert before the compound statement. Each bound check is accompanied with a stop message that tells the user in which array and on which dimension the subscript is out of range. The process stops here for the array;

(4) Otherwise, we go down to the substatements of the current compound statement, take the regions of the concerning array and repeat the above steps.

The algorithm terminates because we can always generate bound checks directly for array references of elementary statements in the control flow graph.

Figures (4a,4b) and Figures (4a,4c) show the running example with the Insertion of unavoidable tests approach, respectively for the intraprocedural and interprocedural options. For the procedure `INITAL`, we have two regions, a read and a write region for array `U` and these regions are treated separately. Bound checks remain for the first dimension `PHI1` of the read region and for the second dimension `PHI2` of the write region. For the procedure `SHALOW` in the intraprocedural version, the cumulated region of the nested loop allow us to generate an upper bound check that is outside the loop. Compared to the intraprocedural version of the first approach, this is a point in favor of the second approach because it may lift test out of loops automatically while the first one does not.

In the interprocedural version, after inserting unavoidable tests for `INITAL`, we have more information from interprocedural analyses in the region of array `UNEW` in `SHALOW`. Like the first approach, the interprocedural preconditions `{M<=512, N<=512}` decide that no check is needed in `SHALOW`. We have the same result for both approaches.

The purpose of Insertion of unavoidable tests is to generate a minimum number of bound checks using the available information from array regions. Bound checks are inserted outside loops and at the beginning of the program. The other advantage of this algorithm is that it detects the sure bound violations or indicates that there is certainly no bound violation as early as possible, thanks to the context given by the top-down analysis of insertion of tests. That is the goal of the second approach group as explained in Section 2. Our region-based algorithm can be parameterized with respect to different notions of array regions, not only convex polyhedra region. Guarded regions, list of regions [Gu and Li 2000] or dimension per dimension regions could be used to improve the computation time of convex regions. Furthermore, we can consider to merge the read and write regions of the same array, or detect arrays that have the same declarations and same regions in order to reduce redundant checks.

6. INTERPROCEDURAL ARRAY BOUND CHECK

Interprocedural array bound checking refers to checking the declared size of a formal array with respect to the range declaration of its corresponding actual argument. A formal array can be associated with an actual array or with an actual array element.

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

In the second case, the size of the formal argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element. The subscript value of an array element determines the order of that

```
   SUBROUTINE INITAL
   READ (5, *) M, N
C <U(PHI1,PHI2)-R-EXACT-{PHI1==M+1, PHI2==1}>
C <U(PHI1,PHI2)-W-EXACT-{PHI1==1, PHI2==N+1}>
   U(1,N+1) = U(M+1,1)
   END
```

```
   SUBROUTINE INITAL
   READ (5, *) M, N
   IF (513.LE.N) STOP "Violation:2nd dim,array U"
   IF (1+N.LE.0) STOP "Violation:2nd dim,array U"
   IF (513.LE.M) STOP "Violation:1st dim,array U"
   IF (1+M.LE.0) STOP "Violation:1st dim,array U"
   U(1,N+1) = U(M+1,1)
   END
```

a) Intraprocedural and interprocedural options for INITIAL

```
   PROGRAM SHALOW
C <UNEW(PHI1,PHI2)-R-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN, MN<=M, MN<=N}>
   DO 35 I = 1, MN
C <UNEW(PHI1,PHI2)-R-EXACT-{PHI1==I, 1<=PHI2, PHI2<=MN, 1<=I, I<=MN, MN<=M, MN<=N}>
      DO 35 J = 1, MN
C <UNEW(PHI1,PHI2)-R-EXACT-{PHI1==I, PHI2==J, 1<=I, I<=MN, 1<=J, J<=MN, MN<=M, MN<=N}>
         UCHECK = UCHECK + ABS(UNEW(I,J))
35 CONTINUE
   END

   PROGRAM SHALOW
   CALL INITAL
   MN = MIN(M, N)
   UCHECK = 0.0
   IF (514.LE.MN) STOP "Violation:1st dim,array UNEW"
   DO 35 I = 1, MN
      DO 35 J = 1, MN
         UCHECK = UCHECK + ABS(UNEW(I,J))
35 CONTINUE
   END
```

b) Intraprocedural option for SHALOW

```
   PROGRAM SHALOW
C <UNEW(PHI1,PHI2)-R-EXACT-{1<=PHI1,PHI1<=MN,1<=PHI2,PHI2<=MN,MN<=M,M<=512,MN<=N,N<=512}>
   DO 35 I = 1, MN
C <UNEW(PHI1,PHI2)-R-EXACT-{1<=PHI1,PHI1<=MN,1<=PHI2,PHI2<=MN,MN<=M,M<=512,MN<=N,N<=512}>
      DO 35 J = 1, MN
C <UNEW(PHI1,PHI2)-R-EXACT-{1<=PHI1,PHI1<=MN,1<=PHI2,PHI2<=MN,MN<=M,M<=512,MN<=N,N<=512}>
         UCHECK = UCHECK + ABS(UNEW(I,J))
35 CONTINUE
   END

   PROGRAM SHALOW
   CALL INITAL
   MN = MIN(M, N)
   UCHECK = 0.0
   DO 35 I = 1, MN
      DO 35 J = 1, MN
         UCHECK = UCHECK + ABS(UNEW(I,J))
35 CONTINUE
   END
```

c) Interprocedural option for SHALOW

Fig. 4. Running example with Insertion of unavoidable tests (declarations omitted - see Fig. 1)

```
PROGRAM VIOLATION
COMMON /FOO/ Z1(10,10), Z2(10,10)
CALL ZERO(Z1,10,20)
PRINT *, Z1
PRINT *, Z2
END
```

```
SUBROUTINE ZERO(X,N,M)
REAL X(N,M)
DO 100 I = 1,N
   DO 100 J = 1,M
      X(I,J) = 1.
100 CONTINUE
END
```

Fig. 5. Interprocedural bound violation example

element in the array.

Within a program unit, the array declaration given for an array provides all range information needed for the array in an execution of the program unit. But in the whole program, when a formal array argument is associated with an actual array argument, we also have to ensure that there is no bound violation in every array access in the called procedure with respect to the array declarations in the calling procedure. If not, we cannot know what happens when accessing the memory beyond the allocated regions. The example in Figure 5 illustrates a simple interprocedural bound violation.

Although exceeding the size of an actual argument array is strictly forbidden in the Fortran standard [ANSI 1983], commercial compilers such as SUN Workshop F77 version 5.0, SGI MIPSpro F90 version 7.3 and IBM XLF F77 version 7.1.0.0 do not check it. One can argue that this kind of violation is rare in practice or conversely, that is used voluntarily, but our implementation found an interprocedural bound violation in one out of the ten benchmarks from SPEC95 CFP. Furthermore, the fact that bugs related to interprocedural checking are much more difficult to track than to intraprocedural one is not a good reason to omit the interprocedural array bound check.

Our analysis traverses the call graph of programs in the invocation order. For each procedure, each call site in this procedure and each actual array argument in the parameter list, the following steps are performed:

(1) Find the corresponding formal array argument;
(2) Compute the size of the formal array;
(3) Translate this size expression to the frame of the caller;
(4) Compute the size of the actual array;
(5) If the actual argument is passed as an array element, then subtract the subscript value of this element from the actual array size and add one;
(6) Test the feasibility of the system built from the precondition of the call site and from the inequality expressing an array bound violation: size of the actual array $<$ size of the formal array. Then:
    —If the system is not feasible, there is no bound violation;
    —If the inequality is true with respect to the precondition, there is a bound violation;
    —Otherwise, we have a test to put before the current call site;

This is a brief description of the algorithm, which, in fact, is a bit more complicated. We use preconditions and information about global variables and calling

Program	Lines	Subroutines	Compile-time checks	Run-time checks
tomcatv	190	1	304	4933x10^7
swim	429	6	772	6991x10^7
su2cor	2332	35	4460	4810x10^7
hydro2d	4292	42	2016	6530x10^7
mgrid	484	12	1162	17608x10^7
applu	3868	16	9562	11562x10^7
turb3d	2101	23	1852	6081x10^7
apsi	7361	96	7172	3936x10^7
fpppp	2784	38	2894	2928x10^7
wave5	7764	105	10546	3404x10^7

Table 1. SPEC95 CFP: numbers of lines, subroutines, compile-time and run-time checks

contexts, such as the relation between actual and formal arguments, to improve the translation process and simplify the inequality characterizing the bound violation.

To compare the size of the actual and formal array arguments, we try to treat dimensions independently by computing k, the number of equal values among the first dimensions of the actual and formal arrays. When the actual argument is an array element, k is also the number of first subscripts that are equal to their corresponding lower bounds. This step simplifies the computation of array sizes and subscript value expressions: the sizes of the actual and formal arrays are computed using all dimensions but the first k ones. Similarly, if the actual argument is an array element, its subscript value expression is evaluated only from the subscript $k+1$. By doing this, the inequality between the sizes of the actual and formal arrays can be simplified, and thus the feasibility test can also be simplified. For instance, in the example in Figure 5, by knowing `N == 10`, we only have to check `10 < M` instead of introducing the non linear expression `10*20 < N*M`. However, some problems stemming from language features such as *array reshaping* (the number and size of dimensions in an actual argument array declaration may be different from those in an associated formal argument array declaration) can prevent this simplification. Furthermore, assumed-size array declarations in Fortran can make this analysis impossible so we use a preliminary phase in PIPS called *array resizing* [Ancourt and Nguyen 2001] to deal with this problem. Our full-fledged algorithm handles all these cases.

7. EXPERIMENTAL RESULTS

We used the SPEC CFP95 benchmark [Dujmovic and Dujmovic 1998], which contains 10 applications written in Fortran. These are scientific benchmarks with floating point arithmetic, and many of them have been derived from publicly available application programs. Each benchmark contains a large amount of subscripted references to arrays. The codes are instrumented and then executed using the standard input data to compute the number of dynamic range checks. Table 1 summarizes relevant information for each benchmark in SPEC CFP95. Note that three of them do not meet the Fortran standard for array declaration and reference: they have pointer-type declarations `REAL A(1)`, although array references in the corresponding procedures are outside the defined extent of the array. We added proper bounds to the declarations in *turb3d, apsi* and *fpppp* by applying array resizing [Ancourt

	Elimination of redundant tests				Insertion of unavoidable tests			
	Intra		Inter		Intra		Inter	
Prog.	**Comp.**	**Run**	**Comp.**	**Run**	**Comp.**	**Run**	**Comp.**	**Run**
tomcatv	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
swim	97.00%	97.00%	98.45%	99.99%	84.60%	99.99%	84.72%	99.99%
su2cor	94.52%	95.20%	96.59%	97.54%	92.12%	96.60%	94.48%	97.66%
hydro2d	95.13%	93.50%	96.13%	94.14%	90.02%	97.70%	93.85%	99.46%
mgrid	91.32%	99.60%	94.92%	99.60%	96.61%	99.50%	97.93%	99.66%
applu	98.54%	96.75%	99.62%	97.09%	96.38%	99.80%	96.41%	99.87%
turb3d	92.23%	56.18%	97.62%	65.00%	87.03%	76.78%	98.97%	85.57%
apsi	97.08%	99.20%	98.02%	99.90%	98,70%	99.31%	99.79%	99.99%
fpppp	94.12%	96.48%	94.61%	97.02%	92.18%	97.23%	95.82%	97.40%
wave5	94.52%	86.26%	94.66%	86.86%	91.12%	89.83%	94.01%	91.29%

Table 2. SPEC95 CFP Intraprocedural Array Bound Check: Percentages of removed compile-time and run-time checks

and Nguyen 2001] to avoid premature aborts due to bound violations.

7.1 Intraprocedural Array Bound Check - Removed Checks

Table 2 shows the percentages of bound checks removed by the two approaches for intraprocedural array bound checking: Elimination of redundant tests and Insertion of unavoidable tests. For each approach, we used the intraprocedural option for transformers and preconditions analyses, which is faster but less accurate and the interprocedural option, which is slower but improves the accuracy. For each combination of approach and option, we measured the percentages of compile-time and run-time checks removed. The number of eliminated compile-time checks may be high, but if remaining checks are inside some frequently executed blocks of code, we will not have much speed-up. Run-time checks are more interesting because they have direct effects on execution time. So to compare between intraprocedural and interprocedural analysis options, between the Elimination of redundant tests and Insertion of unavoidable tests approaches, we use only run-time checks results.

In both approaches, we do not have much gain with interprocedural analysis if the intraprocedural one has already done a good job (*tomcatv, mgrid, applu*). There is more room left to improve the number of removed tests with *turb3d* (8.82%, 8.75%), *su2cor* (2.34%, 1.06%), *swim* (2.99% with Elimination approach), *hydro2d* and *wave5* (respectively, 1.76% and 1.36% with Insertion approach) because of the smaller percentages of removed tests. For *tomcatv*, we can statically prove that there is no intraprocedural bound violation by using either intra- or inter-procedural analyses, which is an interesting result for verification purposes.

Comparing the two approaches, we see that the second works uniformly better. We have almost no gain for *tomcatv, mgrid* and *apsi*, but there are very big gaps between Insertion of unavoidable tests and Elimination of redundant tests for *turb3d* (about 20.0% with both options), *hydro2d* (4.20% with intraprocedural option, 5.32% with interprocedural option) and *wave5* (3.57% with intraprocedural option, 4.43% with interprocedural option).

The percentage of removed tests varies for different benchmarks, approaches and options. It is not very high for *turb3d* and *wave5*, since there are a lot of non-linear expressions in array bound checks that has not yet been handled by PIPS.

	Elimination				Insertion				SUN F77	
Prog.	**Spd**	**PIPS**	**SUN**	**Tot.**	**Spd**	**PIPS**	**SUN**	**Tot.**	**Wo C**	**+ C**
tomcatv	55.5	0:02	0:02	0:04	22.2	0:05	0:02	0:07	0:03	0:13
swim	70.7	0:04	0:02	0:06	23.5	0:12	0:05	0:17	0:04	0:20
su2cor	39.4	0:40	0:25	1:05	10.5	2:28	0:33	3:01	0:33	2:23
hydro2d	107.0	0:16	0:19	0:35	39.8	0:43	0:32	1:15	0:32	1:03
mgrid	85.2	0:04	0:07	0:11	8.9	0:38	0:06	0:44	0:08	0:37
applu	74.9	0:33	0:25	0:58	20.2	2:02	0:54	2:56	0:57	11:12
turb3d	86.0	0:15	0:16	0:31	19.8	1:05	0:17	1:22	0:17	0:44
apsi	55.8	1:16	1:02	2:18	9.1	7:42	1:03	8:45	1:16	3:15
fpppp	50.6	0:42	0:45	1:27	8.8	4:01	0:56	4:57	0:54	1:22
wave5	35.2	5:03	1:25	6:28	8.9	12:04	1:50	13:54	2:04	6:20

Table 3. SPEC95 CFP Intraprocedural Array Bound Check - Compilation times Ultra SPARC 360MHz - Optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

7.2 Intraprocedural Array Bound Check - Compilation Time

The compilation speeds, expressed in source lines per second, obtained with PIPS to parse, analyze (transformers, preconditions, array regions), optimize (array bound check) and generate Fortran code with its own range checking for SPEC CFP95 are shown in Column 2 and 6, Table 3. The speeds are measured with interprocedural analyses for transformers and preconditions, which are slower than the intraprocedural ones. Comment lines are not taken into account. The 10 benchmarks, with 20644 lines of code and 374 subroutines, are processed at an average speed of 66.07 lines per second for the Elimination of redundant tests and 17.24 lines per second for the Insertion of unavoidable tests. The range check optimization phase only takes a very small fraction of this compilation time but we have not attempted to measure it because only the total time matters to the user.

The compilation times for the second approach are longer, especially for *mgrid, apsi, fpppp* and *wave5*. It is due to the satisfiability test used in PIPS to compute array regions. This could be improved by a more sophisticated implementation of array regions as mentioned in Section 5. As shown in Table 2, the percentage of removed checks of Insertion of unavoidable tests is high enough to pay for this tradeoff.

Since PIPS is a Fortran *source-to-source* compiler, the code generated by PIPS with its own range checking is then compiled by other compilers. We measured the compilation times taken by PIPS as a preprocessor and by SUN Workshop F77 5.0 compiler for PIPS generated codes. The original codes of SPEC CFP95 are also compiled with and without the array range checking option of SUN. Experimental results show shorter times for the two implementations of PIPS than for SUN (see columns Total of Elimination of redundant tests and of Insertion of unavoidable tests and column With C of SUN F77).

7.3 Intraprocedural Array Bound Check - Execution Time

The execution times of SPEC CFP95 are measured on different platforms to see the relation between the percentage of eliminated checks and the slowdown. This set of experiments is reported with the optimizing options turned on, using the SPEC measurement guidelines. The code generated by PIPS with its own range checking

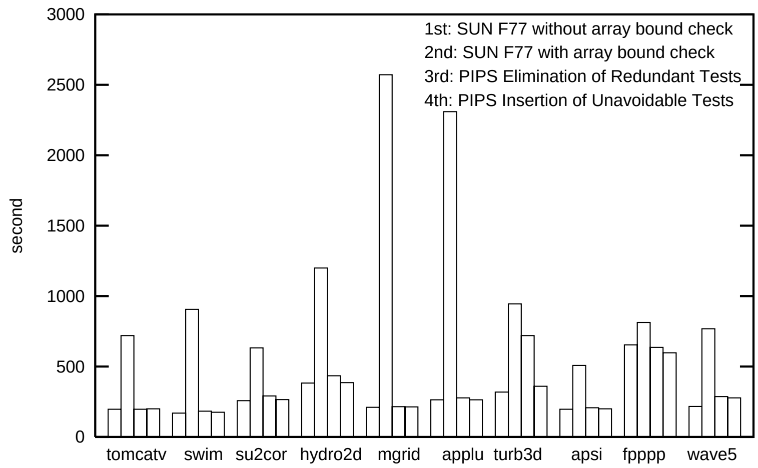

Fig. 6. Execution time: SUN F77 and PIPS - Ultra SPARC 360MHz - Optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

using the interprocedural option for transformers and preconditions is compiled by other compilers to generate executable files. Experiments have been performed with three commercial compilers: SUN Workshop F77 version 5.0, SGI MIPSpro F90 version 7.3 and IBM XLF F77 version 7.1.0.0. There is no range checking option for SGI F77 and GNU G77 compilers and we had to leave them out. For IBM, because an internal compiler error occurred when compiling the Fortran code with options -O5 and -C together, we used -O3. In addition, there is an IO error for *apsi* so we do not have results for this benchmark on the IBM machine. The execution times of codes obtained with and without the bound checking option of these compilers and with the PIPS versions are provided in Figure 6, Figure 7 and Figure 8.

We can see the overheads of range checking in *mgrid* and *applu* for SUN (Figure 6), *mgrid* and *swim* for SGI (Figure 7) and *tomcatv* and *turb3d* for IBM (Figure 8). PIPS optimizing array bound checkers work very well for *tomcatv, swim, mgrid* and *applu*. These benchmarks have more dynamic bound checks than others, as shown in Table 1.

As the range checking of the IBM compiler is already optimized, the PIPS versions work better than IBM in general but worse for *turb3d* benchmark. The reason is that analyses of non-linear expressions are not implemented yet in PIPS. Comparing the execution time of PIPS codes with that of other bound checked codes, on average, PIPS Elimination of redundant tests is about 3.94 times faster than SUN, 1.88 times faster than SGI and 1.03 times faster than IBM. PIPS Insertion of unavoidable tests is about 4.37 times faster than SUN, 2.02 times faster than SGI and 1.07 times faster than IBM. The execution times of programs with range checking added by PIPS are slightly longer than that of the unsafe programs without bound checks. On average, these times for PIPS Elimination of redundant tests are about 19.29% longer for SUN, 7.05% longer for SGI and 16.59% longer for IBM. For PIPS Insertion of unavoidable tests, they are about 5.33% longer for SUN, 0.61% longer for SGI and

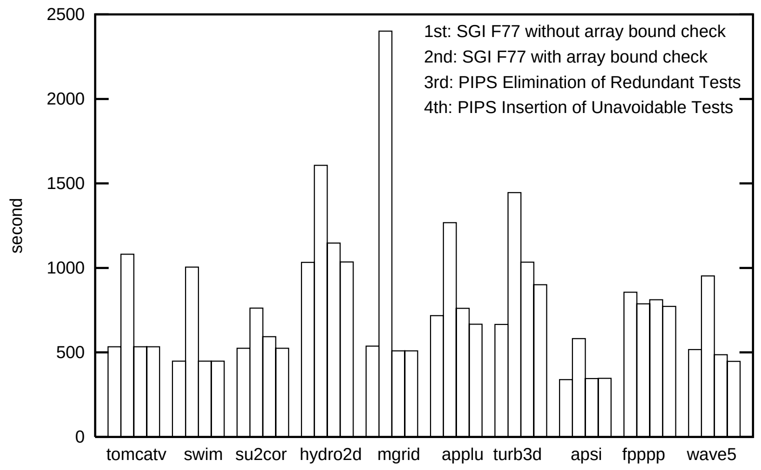

Fig. 7. Execution time: SGI F90 and PIPS - SGI MIPSpro F90 7.3, O2 R5000 195MHz, IRIX 6.3 - Optimized code (f90 -Ofast=ip32_5k)

	SUN	**SGI**	**IBM**	**PIPS Elimination**	**PIPS Insertion**
Line	x			x	
Array	x			x	x
Dimension	x			x	x

Table 4. Debugging information given by different compilers

10.62% longer for IBM.

7.4 Intraprocedural Array Bound Check - Debugging Information

The amounts of information given when a bound violation occured differ between compilers. This information is shown in Table 4 by experiments with three original benchmarks violating the standard for array references: *turb3d*, *apsi* and *fpppp*. The SUN compiler -C option provides the lines of code, the arrays and the dimensions associated to the violations. The SGI -C option does not provide any information, which is particularly clear since the programs did not stop on the system we used when they reached an out-of-bound trap. The IBM -C option spots errors in programs with the "Trace/BPT trap(coredump)" message. Meanwhile, our first array bound checker provides full information about the location of the violations and the second one gives information about the array and the dimension whose bounds are violated. The PIPS Elimination of redundant tests has shorter compilation time and execution time than the SUN compiler while preserving the same diagnostic capabilities. As in code hoisting methods, PIPS Insertion of unavoidable tests propagates bound checks outside loops and into the beginning of the program so the precise location of the violation is lost. But compared to IBM's compiler, which is in the same performance range, the array and the dimension improperly accessed are still known.

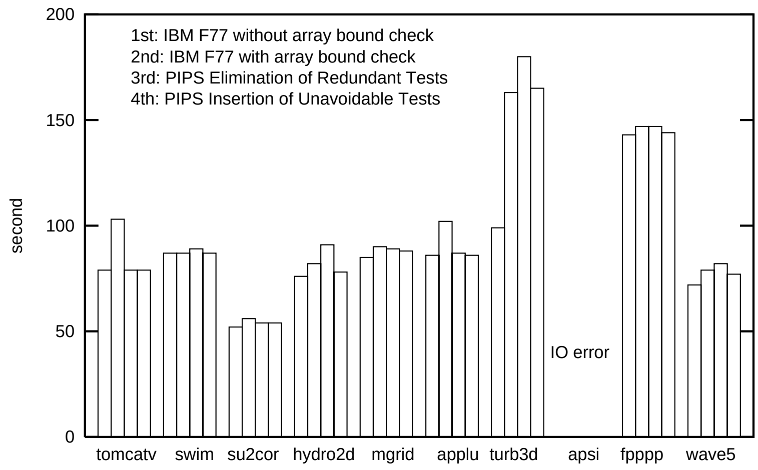

Fig. 8. Execution time: IBM F77 and PIPS - IBM XL F77 7.1, RS/6000 44P-270 375MHz 4 CPU, AIX 4.3 - Optimized code (f77 -O3 -lmass)

	Compilation		**Execution**	
Program	**Checks**	**Time (second)**	**Checks**	**Slowdown**
tomcatv	0	1	0	0.00%
swim	0	3	0	0.00%
su2cor	0	59	0	0.00%
hydro2d	0	21	0	0.00%
mgrid	24	5	40151	0.15%
applu	0	17	0	0.00%
turb3d	29	42	281417	2.12%
apsi	6	85	240127	1.67%
fpppp	2	149	727917	0.79%
wave5	34	156	Bound violation	Bound violation

Table 5. SPEC95 CFP Interprocedural Array Bound Check: Number of added compile-time and run-time checks, compilation time and execution slowdown

7.5 Interprocedural Array Bound Check

Table 5 shows the number of compile-time and run-time checks added, as well as the compilation time and the slowdown caused by the interprocedural array bound checking for the SPEC95 CFP benchmarks. By using static analyses, our checking has proved that there is no interprocedural bound violation in 5 out of 10 benchmarks. Other bound checks are added before some procedure calls in the 5 remaining benchmarks. Since there is no other compiler that does this checking, we cannot compare the effectiveness of our approach but the cost here is small enough. A bound violation is detected in *wave5*, an electromagnetic particle simulation program. Figure 9 contains the piece of code that causes a bound violation for array `BX(NC1)` when passing it as argument in procedure calls. The size of array `TMP(NXD, NY,2)` in subroutine `SLV2XY` must be less than or equal to the size of array `TMP(NX2, *)` in subroutine `SOLV2Y` and so less than or equal to the size of array `BX(NC1)` in subroutine `FIELD`. By using binding information between formal and actual arguments

```
SUBROUTINE FIELD
PARAMETER (NC1 = 78885)
COMMON/EFIELD/EX(NC1),EY(NC1),EZ(NC1),BX(NC1),BY(NC1),BZ(NC1)
NX2 = NX + 2
NY2 = NY + 2
CALL SOLV2Y(NX2,NY2,HX,HY,0.0D0,V,BC,BDY1,BX)
END

SUBROUTINE SOLV2Y(NX2,NY2,HX,HY,DD,Q,BX,BY,TMP)
DIMENSION BX(4),Q(NX2,*),TMP(NX2,*),BY(4)
CALL SLV2XY(NX2-2,NY2-2,NX2,HX,HY,DUMMY,DD,Q,BX,BY,TMP,0)
END

SUBROUTINE SLV2XY(NX,NY,NXD,HX,HY,GX,DD,Q,BX,BY,TMP,IGXSW)
DIMENSION Q(NXD,*),TMP(NXD,NY,2),BX(4),GX(NXD),BY(4)
CALL VSLV1P(NX,NY,NXD,HX,GX,DIAG,Q(1,2),TMP,TMP(1,1,2),IGXSW,ISING)
END
```

Fig. 9. Interprocedural bound violation in *wave5*, SPEC95 CPU

and preconditions, we have `SLV2XY:NXD = SOLV2Y:NX2 = FIELD:NX2 = FIELD:NX` +2 and `SLV2XY:NY = SOLV2Y:NY2-2 = FIELD:NY2-2 = FIELD:NY`. So we have the following check about array sizes: `FIELD:NC1 < SLV2XY:NXD*SLV2XY:NY*2` that is translated into the frame of `FIELD` by `NC1 < (NX+2)*NY*2`. When executing the instrumented code with its standard input data where the grid size `NX = 1250` and `NY = 60`, we have `78885 < 1252*60*2` is true, so there is a bound violation here.

Because `BX` is accessed outside it declared range by procedure calls and `BY` is allocated just after `BX` in memory, the two arrays `TMP` and `BY` share some memory locations if bound violations are not checked. So if interprocedural array bound checking is omitted, bound violation makes other analyses such as alias analysis become much more difficult.

8. CONCLUSION

We designed and experimented two intraprocedural and one interprocedural algorithms for array bound checking. The number of removed/added bound checks, the compilation and the execution times, the information about violations were measured for the SPEC95 CFP benchmarks with three different compilers and with our experimental implementations.

Some other studies [Richardson and Ganapathi 1989] suggest that interprocedural analyses give little benefit in optimization and are too expensive to be worthwhile. However, our implementations show that with powerful and efficient interprocedural analysis techniques, more redundant checks are removed and we can even prove the absence of bound violations in some programs. Once again, it is still a question of tradeoff between speed and accuracy, but in the domain of verification, proving the correctness of code is the most important criterion.

The experimental results show the effectiveness and the limited optimization cost of our two intraprocedural array bound check approaches: Elimination of redundant tests and Insertion of unavoidable tests.

The first one puts array bound checks everywhere and then removes the redundant ones. This approach is simple and the number of eliminated tests depends on the strength of data flow analyses, such as predicates over scalar integer variable values, used to perform the elimination.

The second implementation inserts useful checks directly by using array region analyses. It produces better results with a higher number of dynamic removed checks and faster execution times. For a small program like *tomcatv*, the differences between the two approaches are limited, but for large programs with more than 2000 lines of code, there are clear differences. The maximum improvement in dynamic removed bound checks is 20.57% for *turb3d*. The main advantage of this top-down analysis approach is that it detects the sure bound violations or indicates that there is certainly no bound violation as soon as possible.

Within the SUN environment, we measured shorter compilation times using our two intraprocedural source-to-source array bound checkers followed by F77 than using F77 alone with its -C option. The compilation time speedups are 3.03 for Elimination of redundant tests and 1.1 for Insertion of unavoidable tests.

At run-time, the slowdowns measured for SUN and SGI compilers are large enough to make improvement easy. We mostly obtained speedups by adding array bound checks of the Insertion of unavoidable tests version, in comparison with the execution time with no bound checks of the SGI compiler. This is not the case with IBM XLF compiler 7.1.0.0, which nevertheless is not uniformly efficient and which breaks down with an internal error when combining options -C and -O5. However, our execution times are in the same range as IBM's, 7 times out of 9 slightly better. Furthermore, as does the SUN compiler, our array bound checkers provide programmers useful and precise information about the dimension and name of the array experiencing a bound violation, even the line of code with the first approach, while the IBM compiler only indicates that an overflow occurred somewhere. This shows that a simple comparison of techniques is not possible since the amount of information produced differs. Without good diagnostic capability, many hours can be wasted in debugging the cause of an array bound violation message. It should not be infer from Figure 6 and 8 that the SUN compiler is not as good as the IBM's one.

We could not directly compare our results with those presented in [Kolte and Wolfe 1995] about the Perfect Club and Riceps benchmarks because the authors do not include execution times. We observe that the percentage of removed checks is not an accurate predictor of slowdown. These two numbers are not proportional because they may depend on architectures and compilers. For example, the execution time of code without bound checks for *swim* is only 92.35% although 99.99% of dynamic checks removed (see Table 2 and Figure 6). So to evaluate array bound checking optimization, it is necessary to compare the execution times of generated codes which is missing in [Kolte and Wolfe 1995]. However, their best figures are in the very same range as ours and it is very interesting to see that specific techniques do not work better than re-used techniques. Furthermore, no information about the origin of violation is preserved.

Interprocedural array bound checking is as important as the intraprocedural one because it allows us to detect complicated bugs. We developed an efficient algorithm that guarantees the safety of code. This can be a complementary phase when doing array range checking for whole programs.

Result analysis shows the importance of code quality. Proper array declarations are needed to avoid out-of-bound errors caused by standard violation. Furthermore, some benchmarks require more sophisticated techniques or modifications such as cloning, parameter checking, scalarization for dealing with indirections, scalarization for loop bounds, loop increments different than one,... For example, we can improve the percentage of removed checks in Elimination of redundant tests from 94.14% to 99.50% for *hydro2d* by cloning the subroutine `ADLEN` which has two totally different behaviors for two parameter values: "half" and "full" step. The execution time on SUN is reduced by 10%. The elimination percentage goes up to 100% from 97.09% for *applu* by adding one `STOP` statement after the parameter checking that is performed for lower bound tests but not for upper bound test of read variables (`NX,NY,NZ` in the main program `APPLU`). A 5.4% decrease of the execution time is then measured on SUN. A poor code quality can make static analysis insufficient; run-time checks remain and run-time failures cannot be eliminated.

Some analyses for non-linear expressions that have direct impact on array range checking have not been implemented in PIPS, so we do not have very good results for *turb3d* and *wave5* in Elimination of redundant tests.

For the array region-based version, the number of bound checks could be reduced by replicating code as in [P.Midkiff et al. 1998; Moreira et al. 2000] when `MAY` regions give necessary but not sufficient conditions for a bound violation to occur. However, the code size increase may raise problems that go beyond a simple reuse of existing techniques.

The implementations in this paper suggest that commercial products with automatic analyses could easily be improved to perform efficient array bound checking without sacrificing information about the location of the violation. Less than 1600 additional lines of C code (comments included) are sufficient to implement both intraprocedural and interprocedural checking in PIPS. The execution overhead is small enough to consider the use of safe versions of programs for production activities. These array bound checkers could possibly be a source-to-source preprocessor for GNU g77, since it does not have a range checking option.

Our approaches to optimizing bound checking could also be applied to other imperative languages for scientific applications that require software verification such as Ada, Java,... The PIPS software and documentation as well as the array bound checking implementations are available on *http://www.cri.ensmp.fr/pips*.

ACKNOWLEDGMENTS

We would like to thank Béatrice Creusillet and Fabien Coelho for implementing some key algorithms in PIPS as well as Corinne Ancourt, Pierre Jouvelot and Ronan Keryell for their helpful comments. We also wish to give special thanks to Serge Algarotti, Catherine Mongenet and Romaric David for letting us use their IBM and SGI machines.

REFERENCES

Aggarwal, A. and Randall, K. H. 2001. Related field analysis. In *International Conference on Programming Language Design and Implementation* (Snowbird,Utah,USA, June 2001), pp. 214–220.

Ami, T. L., Reps, T., Sagiv, M., and Wilhelm, R. 2000. Putting static analysis to work for verification: A case study. In *ISSTA'00* (Portland, Oregon, 2000), pp. 26–38.

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

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

Austin, T. M., Breach, S. E., and Sohi, G. S. 1994. Efficient detection of all pointer and array access errors. *ACM SIGPLAN Notices 29*, 6, 290–301.

Bodik, R., Gupta, R., and Sarkar, V. 2000. ABCD: Eliminating Array Bounds Checks on Demand. In *International Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada, 2000), pp. 321–333.

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

Chamberlain, B. L., Lewis, E. C., Lin, C., and Snyder, L. 1999. Regions: An abstraction for expressing array computation. In *APL'99* (Scranton, PA, USA, 1999), pp. 41–49. ACM.

Chin, W.-N. and Goh, E.-K. 1995. A reexamination of "Optimization of array subscript range checks". *ACM Transactions on Programming Languages and Systems 17*, 2 (March), 217–227.

Cousot, P. 1990. Methods and logics for proving programs. In J. van Leeuwen Ed., *Formal Models and Semantics*, Volume B of *Handbook of Theoretical Computer Science*, Chapter 15, pp. 843–993. Elsevier Science Publishers B.V., Amsterdam, The Netherlands.

Cousot, P. and Cousot, R. 1976. Static determination of dynamic properties of programs. In *Second International Symposium on Programming* (1976), pp. 106–130. Dunod, Paris, France.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages* (Los Angeles, California, January 1977), pp. 238–252.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages* (January 1978), pp. 84–96.

Creusillet, B. and Irigoin, F. 1995. Interprocedural array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*, Volume 1033 of *Lecture Notes in Computer Science* (1995), pp. 46–60. Springer-Verlag.

Creusillet, B. and Irigoin, F. 1996. Exact vs. approximate array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*, Volume 1239 of *Lecture Notes in Computer Science* (1996), pp. 86–100. Springer-Verlag.

Delzanno, G., Jung, G., and Podelski, A. 2000. Static analysis of array bounds as infinite-state model checking. Extended abstract.

Dor, N., Rodeh, M., and Sagiv, M. 2001. Cleaness checking of string manipulation in c programs via integer analysis. In *Static Analysis*, Volume 2126 of *Lecture Notes in Computer Science* (2001), pp. 194–212. Springer-Verlag.

Duesterwald, E., Gupta, R., and Soffa, M. L. 1993. A practical data flow framework for array reference analysis and its application in optimization. In *International Conference on Programming Language Design and Implementation* (Albuquerque, N.M, June 1993), pp. 68–77.

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

Feautrier, P. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20*, 1, 23–53.

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

GUPTA, R. 1990. A fresh look at optimizing array bound checking. In *International Conference on Programming Language Design and Implementation* (White Plains, New York, June 1990), pp. 272–282.

GUPTA, R. 1993. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems 2*, 1-4 (March-December), 135–150.

HARRISON, W. H. 1977. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering SE-3*, 3 (May), 243–250.

IRIGOIN, F. 1993. Interprocedural analyses for programming environments. In *Environments and Tools for Parallel Scientific Computing* (1993), pp. 333–350. Elsevier.

IRIGOIN, F., JOUVELOT, P., AND TRIOLET, R. 1991. Semantical interprocedural parallelization: an overview of the PIPS project. In *International Conference on Supercomputing* (June 1991), pp. 144–151.

KOLTE, P. AND WOLFE, M. 1995. Elimination of redundant array subscript range checks. In *International Conference on Programming Language Design and Implementation* (La Jolla, CA, USA, June 1995), pp. 270–278.

LIN, Y. AND PADUA, D. 1999. Demand-driven interprocedural array property analysis. In *International Workshop on Languages and Compilers for Parallel Computing* (1999).

MARKSTEIN, V., COCKE, J., AND MARKSTEIN, P. 1982. Optimization of range checking. *ACM SIGPLAN Symposium on Compiler Construction*, 114–119.

M.ASURU, J. 1992. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems 1*, 2 (June), 109–118.

MAYDAN, D. E. 1992. *Accurate Analysis of Array References*. Ph. D. thesis, Computer Science Department, Stanford University.

MAYDAN, D. E., AMARASINGHE, S. P., AND LAM, M. S. 1992. Data dependence and data-flow analysis of arrays. In *International Workshop on Languages and Compilers for Parallel Computing* (1992).

MENON, V. AND PINGALI, K. 1999. A case for source-level transformations in MATLAB. In *ACM SIGPLAN 2nd Conference on Domain-Specific Languages*, Volume 35 (Austin, Texas, USA, October 1999), pp. 53–65.

MOREIRA, J. E., MIDKIFF, S. P., AND GUPTA, M. 2000. From flop to megaflops: JAVA for technical computing. *ACM Transactions on Programming Languages and Systems 22*, 2 (March), 265–295.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

MUCHNICK, S. S. AND JONES, N. D. 1981. *Program Flow Analysis: Theory and Applications*. Prentice Hall.

NGUYEN, T. V. N. AND IRIGOIN, F. 2001. Interprocedural program analyses for efficient array bound checking. Technical report, CRI-Ecole des Mines de Paris, France.

NGUYEN, T. V. N., IRIGOIN, F., ANCOURT, C., AND KERYELL, R. 2001. Efficient intraprocedural array bound checking. In *Second International Workshop on Automated Program Analysis, Testing and Verification* (Toronto, Canada, May 2001).

PAEK, Y., HOEFLINGER, J., AND PADUA, D. 2001. Efficient and precise array access analysis.

P.MIDKIFF, S., E.MOREIRA, J., AND M.SNIR. 1998. Optimizing array reference checking in JAVA programs. *IBM Systems Journal 37*, 3, 409–453.

RICHARDSON, S. AND GANAPATHI, M. 1989. Interprocedural optimization: Experimental results. *Software - Practice and Experience 19*, 2 (February), 149–169.

RUGINA, R. AND RINARD, M. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *International Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada, 2000), pp. 182–195.

SCHRIJVER, A. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons.

SCHWARZ, B., KIRCHGASSNER, W., AND LANDWEHR, R. 1988. An optimizer for Ada - design, experiences and results. In *International Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA, June 1988), pp. 175–184.

SPEZIALETTI, M. AND GUPTA, R. 1995. Loop monotonic statements. *IEEE Transactions on Software Engineering 21*, 6 (June), 497–505.

STEFFEN, J. L. 1992. Adding run-time checking to the portable C compiler. *Software - Practice and Experience 22*, 4 (April), 305–316.

SUZUKI, N. AND ISHIHATA, K. 1977. Implementation of an array bound checker. In *Symposium on Principles of Programming Languages* (1977), pp. 132–143.

TRIOLET, R., FEAUTRIER, P., AND IRIGOIN, F. 1986. Automatic parallelization of Fortran programs in the presence of procedure calls. Technical Report 120 (March), Universite P. et M. Curie, Laboratoire MASI, Paris-France.

V.AHO, A., SETHI, R., AND D.ULLMAN, J. 1986. *Compilers Principles, Techniques, and Tools.* Addison-Wesley.

WELSH, J. 1978. Economic range checks in Pascal. *Software - Practice and Experience 8*, 85–97.

WONNACOTT, D. 2000. Extending scalar optimizations for arrays. In *International Workshop on Languages and Compilers for Parallel Computing* (New York, 2000), pp. 93–107.

XI, H. AND PFENNING, F. 1998. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices 33*, 5, 249–257.