Efficient and Effective Array Bound Checking

THI VIET NGA NGUYEN and FRANÇOIS IRIGOIN
Ecole des Mines de Paris

Array bound checking refers to determining whether all array references in a program are within their declared ranges. This checking is critical for software verification and validation because subscripting arrays beyond their declared sizes may produce unexpected results, security holes, or failures. It is available in most commercial compilers but current implementations are not as efficient and effective as one may have hoped: (1) the execution times of array bound checked programs are increased by a factor of up to 5, (2) the compilation times are increased, which is detrimental to development and debugging, (3) the related error messages do not usually carry information to locate the faulty references, and (4) the consistency between actual array sizes and formal array declarations is not often checked.

This article presents two optimization techniques that deal with Points 1, 2, and 3, and a new algorithm to tackle Point 4, which is not addressed by the current literature. The first optimization technique is based on the elimination of redundant tests, to provide very accurate information about faulty references during development and testing phases. The second one is based on the insertion of unavoidable tests to provide the smallest possible slowdown during the production phase. The new algorithm ensures the absence of bound violations in every array access in the called procedure with respect to the array declarations in the calling procedure. Our experiments suggest that the optimization of array bound checking depends on several factors, not only the percentage of removed checks, usually considered as the best improvement measuring metrics. The debugging capability and compile-time and run-time performances of our techniques are better than current implementations. The execution times of SPEC95 CFP benchmarks with range checking added by PIPS, our Fortran research compiler, are slightly longer, less than 20%, than that of unchecked programs. More problems due to functional and data recursion would have to be solved to extend these results from Fortran to other languages such as C, C++, or Java, but the issues addressed in this article are nevertheless relevant.

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: Array bound checking, interprocedural analysis

Authors' addresses: T. V. N. Nguyen (current address): ICPS-LSIIT, Parc d'innovation, Boulevard Sébastien Brant, BP 10413, F-67412 Illkirch Cedex, France; email: nguyen@icps.u-strasbg.fr; F. Irigoin, Centre de Recherche en Informatique, Ecole des Mines de Paris, 35 rue Saint Honoré, 77305 Fontainebleau, France; email: irigoin@cri.ensmp.fr.

1. INTRODUCTION

Array bound checking refers to determining whether all array references in a program are within their declared ranges. 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 produce 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 its defined extent.

Naive implementation of bound checking can be very slow because every array access must be guarded 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, although these two bound checks can be implemented as one unsigned comparison instruction, that is, `if ((unsigned) (ref-low) > up-low) stop`. If the reference is smaller than the lower bound, the unsigned subtraction underflows and creates a very large number, causing the test to fail. However, the overhead still remains, in part because other code transformations or optimizations are prevented by these bound checks [Moreira et al. 2000].

Some compilers solve this problem by providing the user with a compile-time option to enable or disable the checking. The purpose of this option, as mentioned by Muchnick [1997], is to allow users to enable checking in the development and debugging processes, and then, once all the defects are supposedly found and fixed, to turn it off for the production version. However, bound checking is just as important for delivered versions of programs as for development versions because the production ones may have bugs that were not even observed during testing phases. 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.

But to be efficient and effective, checking array references involve several issues beyond standard array bound checking. Stopping the execution at the first violation does not support effective debugging because runs can last very long. Raising a violation when an old idiom, such as the Fortran 66 array pointer-like declaration A(1), is used is not effective. Not raising a violation when an array is accessed through a pointer because the size of the formal array A(*), in Fortran, or A[], in C, is unknown is deceptive. These three issues were revealed by our empirical studies. The first issue is easy to deal with, while the two other issues require program analysis to determine the allocated sizes of arrays pointed to. To that end, we developed techniques named *array resizing* in Ancourt and Nguyen [2001] which enable array bound checking for arrays accessed through pointers.

This article contains contributions about the other four more important issues: execution slowdown, compilation time, lack of information about the error source, and compatibility between call sites and procedure declarations. It presents two approaches for limiting the number of array bound checks by using the analyses provided by PIPS, our Fortran research compiler [Irigoin et al. 1991]. The first technique is based on the elimination of redundant tests, to provide very accurate information about faulty references during development

and testing phases. The second one is based on the insertion of unavoidable tests to provide the smallest possible slowdown during the production phase. Our experiments suggest that the optimization of array bound checking depends on several factors, not only the percentage of removed checks, usually considered as the best improvement measuring metrics. The number of removed checks, compile-time and run-time performances, and debugging capability of our techniques are better than the current commercial implementations and related work.

We also introduce in this article a new kind of error checking that verifies whether the declared size of a formal array is consistent with the size of the corresponding actual parameter, for languages with the same parameter passing rules as Fortran. This analysis is apparently only addressed in some Salford Fortran compilers and the static analyzer *forcheck*, although this conformance test is as important as out-of-bound checking.

In addition, a secondary 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, instead of designing new specific algorithms. 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 array bound violations or to detect them at either compile-time or run-time.

The article 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 and its existing analyses that are used in the following sections is given in Section 3. Section 4 presents our first array 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 still known. Section 6 describes the actual/formal array size checking. Results obtained with our intra- and interprocedural techniques are reported and compared to three commercial compilers in Section 7. Conclusions are given in Section 8.

2. RELATED WORK

The first approach to optimizing array bound checks was developed by Markstein et al. [1982] and then refined several times by Asuru [1992] and Gupta [Gupta 1990, 1993], Spezialetti and Gupta [1995], and Kolte and Wolfe [1995]. They introduced 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 was identical or subsumed by other bound checks was suppressed in a local elimination. In a global elimination, an algorithm first modified bound checks to create additional redundant

checks and then carried out the elimination of redundant checks by using the notions of *available* and *very busy* checks. These two algorithms used backward and forward data flow analyses to solve the problems. For the propagation of checks out of loops, Gupta [1993] identified candidates for propagation which include invariants, loops with increment or decrement of 1, and variables of increasing or decreasing values. Then he used check hoisting to move checks out of loops. However, the published results are not very convincing because of the small size of his examples and because the optimizations were only applied by hand.

Kolte and Wolfe [1995] relied on a check implication graph where nodes were sets of range checks in canonical form and edges denoted implications among these families. As in Gupta [1993], they also computed the *available* and *anticipatable* checks by solving forward and backward data flow problems. To create more redundant checks, there were 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 used 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 led to experimental results that showed the necessity of range check optimization and 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 lacked comparisons between the execution times of codes with and without optimized bound checks to show the impact of removed checks. Furthermore, there were no mentions of bound violations in PerfectClub (*mdg, spc77, trfd*) and Riceps benchmarks (*linpackd*) which are caused by declaring 1 as the last upper bound of the formal array declarations.

Suzuki and Ishihata [1977] implemented a system that inserted logical assertions before array element accesses and then used 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, that is, without goto statements.

The abstract interpretation approach proposed by Cousot and Cousot [1976] and Cousot and Halbwachs [1978] considered array range checking as an example of the automated verification of execution properties of programs. As in Harrison [1977], Welsh [1978], Schwarz et al. [1988], and Rugina and Rinard [2000], they used 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 depended on the different rules they used. Since the algorithms in the abstract interpretation and the program verification approaches did 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 could not be evaluated at compile-time still remained. For instance, Rugina and Rinard [2000] proposed a novel framework for the symbolic bounds analysis that characterized the

regions of memory accessed by statements and procedures. If these regions fall within array bounds, unnecessary array bounds checks are eliminated. However, as said in the article, they did not attempt to eliminate partially redundant checks or move checks to less frequently executed program points. Earlier in the literature, Harrison [1977] and Welsh [1978] used value range propagation to eliminate redundant tests and verify the program correctness. Also present in this approach was the model checking group (Delzanno et al. [2000]), which uses fix point acceleration techniques to help the automated verification of programs.

Pointer, string and array access errors in C were also studied in many articles [Steffen 1992; Hasting and Joyce 1992; Austin et al. 1994; Jones and Kelly 1997; Patil and Fischer 1997; Necula and Lee 1998; Wagner 2000; Dor et al. 2001; Kowshik et al. 2002; Evans and Larochelle 2002]. Lightweight static analyses have been used to detect buffer overflow vulnerabilities, but at the expense of soundness and completeness [Wagner 2000; Evans and Larochelle 2002]. These analyzers will sometimes generate false warnings, or even miss real problems. The overheads of run-time safety checking reported in some research papers were significant, ranging from 130% to 540% by [Austin et al. 1994], and about five times by Purify [Hasting and Joyce 1992]. In addition, Purify cannot detect if we access past the end of an array into the region of the next variable. The certifying compiler of Necula and Lee [1998] does not succeed in eliminating bound checks when the size of the formal array in a called procedure is unknown. Kowshik et al. [2002] introduced a language called *Control-C* with key restrictions to ensure that the memory safety of code can be verified entirely by static checking. However, their assumptions were restrictive, and only suitable for a small class of specific purpose languages, that is, for real-time control systems.

Other articles [Midkiff et al. 1998, Moreira et al. 2000, 2001] described another approach to optimize array reference checking in Java programs. It was based on code replication. Loop iteration spaces are partitioned 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 by Bodik et al. [2000], can reduce by 45% the number of executed bound checks for a representative set of Java programs. Aggarwal and Randall [2001] used related field analysis to remove about 50% array bound checks that have already optimized by other simple optimizations. Qian, Handren, and Verbrugge [2002] described a bound check elimination algorithm which combined three analyses, variable constraint, array field, and rectangular array analysis, to prove the safety of array references. Java virtual machines can use this information to avoid emitting bound checks for these references.

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; 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 [Cousot and Cousot 1976; Suzuki and Ishihata 1977; Harrison 1977; Cousot and Halbwachs 1978; Welsh 1978; Schwarz et al. 1988; Midkiff et al. 1998; Rugina and Rinard 2000].

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.

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 cannot be proven that no access violation will not occur, tests are generated. 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.

The amount of information about the array bound violation was never discussed in the above articles. It was often reduced to *a violation occurred* with no information about the array accessed or 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 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. The amount of information preserved by each approach is also reported. Our two array bound check optimizers as well as the actual/formal array size checking were implemented using preexisting analyses in PIPS. They are described in the next section.

3. PIPS OVERVIEW

PIPS, a source-to-source Fortran compiler, consists of several program analyses passes dealing with call graph computation, data dependences, transformers, preconditions, use-def chains, convex array regions, and of program transformations such as loop transformations, constant folding, and dead code elimination. Each analysis is performed only once on each procedure and produces a summary result that is later used at call sites. The running example in Figure 1 is used to illustrate three analyses for transformers, preconditions, and array regions, which are used by our array bound checkers. This example is an excerpt from an industrial application.

```
PROGRAM MAIN
COMMON /CT/ A(52,21,60)
COMMON /CI/ K,L,M,N
DATA NC /17/
READ *,K,L,N
IF (L.GE.1.AND.N.GE.1) THEN
   NI = 4
   L = L+1
   M = 2*K+1
   K = K+1
   CALL EXTR(NI,NC)
ENDIF
END
```

```
SUBROUTINE EXTR(NI,NC)
COMMON /CT/ A(52,21,60)
COMMON /CI/ K,L,M,N
DO I = L,K
   A(I,1,NC+1) = I*(NC+1)
   J = L+M-I
   A(J,1,NC) = A(I,1,NC)
   A(J,1,NC+1) = A(I,1,NC+1)
ENDDO
END
```

Fig. 1. PIPS running example.

3.1 Transformers

Transformers abstract the effects of statements on program variables. A statement, when executed in a certain state, yields a new state upon termination. In other words, a statement is a state-to-state mapping and the related transformer is a state-to-state relation which includes this function. Transformers are useful to summarize compound statements and module effects and to perform modular analysis.

Semantically, the denotation of a transformer ($\mathcal{T}$) is a function from the set of program states to itself:

$$\mathcal{T} : \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \mathit{State}.$$

In general, the semantic functions for exact transformers are not computable, except for very simple programs, so approximations are needed. In PIPS, convex polyhedra [Schrijver 1986] are used to approximate sets. A *convex polyhedron* over $\mathbb{Z}$ is a set of points of $\mathbb{Z}^n$ defined by $P = \{x \in \mathbb{Z}^n : A.x \leq b\}$ where A is a matrix of $\mathbb{Z}^m \times \mathbb{Z}^n$ and b a vector of $\mathbb{Z}^m$. Each transformer is associated to a list of variables whose values are modified by the statement, and a predicate containing affine equalities and inequalities. For efficiency, it is better to keep track of the few modified variables rather than to add one equation for each of the many unmodified variables. From a mathematic point of view, all implicit equations must be taken into account.

Transformers are computed from elementary statements such as `continue`, `stop`, `read`, `write` and assignments to compound statements such as conditional statements, loops, and sequences of statements. They are also propagated interprocedurally through procedure calls under a no recursion assumption. For array bound checking, only transformers on integer scalar variables are used.

3.1.1 *Elementary Statement.* Assignments of scalar integer variables with affine right-hand side expressions are handled exactly in PIPS. If the same variable appears on both sides, the initial input value and the final output value appear in the constraints. For instance, `T(I){I==I#init+1}` models the effect of the assignment `I=I+1`. The transformer for a `CONTINUE` statement is the identity function `T(){}`, for a `STOP` statement is `T(){0==-1}`, because the set of states reached after a stop is empty, and for an input statement

such as `READ *,N` is `T(N){}`, which means that only the value of variable `N` is modified and that no relationship between its old and its new values is known.

3.1.2 *Conditional Statement.* The test condition is approximated by a convex polyhedron which is combined with the transformer of the true branch. The negation of the test condition also is approximated and added to the transformer of the false branch. The transformer of the test statement is the convex hull of the two above predicates, including the implicit equations for unmodified variables.

3.1.3 *Loop Statement.* To handle loops, different kinds of fixed points are provided to have a flexible choice between efficiency and precision [Irigoin et al. 1991]. For our experiments, we selected a fixed point operator based on discrete derivatives. The loop body transformer is used to find constraints on the derivatives. Invariants, both equations and inequalities, are directly deduced from these constraints using a discrete integration.

3.1.4 *Sequence of Statements.* We assume that each statement in the sequence has been associated to its transformer. To combine transformers, relevant input and output variables are renamed as intermediate values, constraints are merged, and intermediate values are eliminated by projection. Transformers are propagated from bottom to top in the abstract syntax tree of the module.

3.1.5 *Interprocedural Transformers.* Interprocedural propagation of transformers is performed by traversing the call graph in the reverse invocation order, which processes a procedure after its callees. The summary transformer of a procedure is the transformer computed for the procedure body after projecting its local dynamic variables. Information about formal parameters, global variables, and static variables is preserved.

When a call to a procedure is encountered, the transformer of the call statement is computed by translating the summary transformer of the called procedure to the frame of the calling procedure. The translation into the scope of the caller uses global variables information and the bindings between actual and formal parameters. Equations are built to link formal and actual arguments when they are affine. These equations are added to the summary transformer and formal parameters are eliminated.

Transformers of the running example, analyzed interprocedurally, are illustrated in Figure 2. Each statement is preceded by its transformer. We have no transformer information on the statements assigning the elements of array `A` in subroutine `EXTR`, because only scalar variables are handled. With `J=L+M-I`, we have `T(J){I+J==L+M}` which is also the transformer of the sequence of five statements. The transformer of the `DO` loop, `T(I,J){K+1<=I, L<=I}` is always correct, whether the loop is executed or not. Since `K`, `I`, and `L` are local variables of `EXTR`, they are projected from the global transformer of the module and an identity summary transformer is obtained: no integer scalar variable in the caller's scope is modified.

```
      PROGRAM MAIN
C     T(K,L,N) {}
      READ *,K,L,N
C     T(K,L,M,NI) {L+K#init==L#init+K, L#init<=L, L<=L#init+1}
      IF (L.GE.1.AND.N.GE.1) THEN
C     T(K,L,M,NI) {K==K#init+1, 2K==M+1, L==L#init+1, NI==4}
C        BEGIN BLOCK
C     T(NI) {NI==4}
         NI = 4
C     T(L) {L==L#init+1}
         L = L+1
C     T(M) {M==2K+1}
         M = 2*K+1
C     T(K) {K==K#init+1}
         K = K+1
C     T() {}
         CALL EXTR(NI,NC)
C        END BLOCK
      ENDIF
      END

C     T() {}
      SUBROUTINE EXTR(NI,NC)
C     T(I,J) {K+1<=I, L<=I}
      DO I = L,K
C     T(J) {I+J==L+M}
C        BEGIN BLOCK
C     T() {}
         A(I,1,NC) = I*NC
C     T() {}
         A(I,1,NC+1) = I*(NC+1)
C     T(J) {I+J==L+M}
         J = L+M-I
C     T() {}
         A(J,1,NC) = A(I,1,NC)
C     T() {}
         A(J,1,NC+1) = A(I,1,NC+1)
C        END BLOCK
      ENDDO
      END
```

Fig. 2. Transformers of the running example (declarations omitted).

`T(K,L,M,NI){K==K#init+1,2K==M+1,L==L#init+1,NI==4}` is the transformer for the sequence of five statements in the `IF` true branch. The values of variables are not changed by the false branch. The transformer of the `IF` statement is the convex hull of the two branch predicates. Since the values of `K` and `L` can be increased by 1 or not, we have the final transformer of the test statement: `T(K,L,M,NI){L+K#init==L#init+K,L#init<=L,L<=L#init+1}`. Additional information is retained because of interactions between the true and false transformers and the test condition.

3.2 Preconditions

Precondition analyses try to discover constraints holding among the values of scalar variables of a program at a control point and at an entry point. For

any statement, the precondition and the postcondition provide a condition that holds just before the statement execution for the former and just after for the latter. The transformer of a statement is applied to the precondition of the statement to obtain the postcondition. This postcondition becomes the precondition of the following statement in a sequence. We compute the overapproximations ($\overline{\mathcal{P}}$) of preconditions ($\mathcal{P}$) which describe the set of program states reached just before the execution and then can be represented as functions from the set of statements to the power-set of the set of program states:

$$\mathcal{P}, \overline{\mathcal{P}} : \quad \begin{array}{l} \mathit{Statement} \longrightarrow \wp(\mathit{State}). \\ \mathcal{P}(s) \subseteq \overline{\mathcal{P}}(s) \end{array}$$

These overapproximated preconditions are computed for scalar variables and represented as systems of equalities and inequalities like transformers. They are propagated from the module entry point down to the abstract syntax tree leaves.

3.2.1 *Initial Precondition.* The precondition on the entry of a procedure represents what is known about the variables at the start of any execution. 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 the precondition `P(){}` which represents all possible program states.

3.2.2 *Elementary Statement.* The precondition and transformer of a statement are used to compute the postcondition. The variables of the precondition and the initial values of the transformer are renamed to intermediate variables in order to merge the precondition's and the transformer's constraints. The intermediate variables are then projected to obtain the postcondition of the current statement. The postcondition for a `STOP` statement always is `P(){0==-1}`; for a single input statement `READ *,N`, is `P(N){}`.

3.2.3 *Conditional Statement.* As for transformers, the postcondition of a test statement is the convex hull of the two postconditions that have been propagated from the precondition of the statement, along the true and false branches.

3.2.4 *Loop Statement.* The loop body preconditions and the loop postcondition are derived directly using the loop fix point transformer. The accuracy is improved by detecting if the loop is always, never, or sometimes entered, and by using the corresponding semantic equations.

3.2.5 *Sequence of Statements.* The transformer associated to each statement in a sequence, which is computed by a previous phase, is applied to the precondition of the first statement to obtain the postcondition. This postcondition becomes the precondition of the following statement and we repeat the process until we reach the postcondition of the last statement.

3.2.6 *Interprocedural Preconditions.* 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

```
      PROGRAM MAIN
      DATA NC /17/
C     P() {NC==17}
      READ *,K,L,N
C     P(K,L,N) {NC==17}
      IF (L.GE.1.AND.N.GE.1) THEN
C     P(K,L,N) {NC==17, 1<=L, 1<=N}
         NI = 4
C     P(K,L,N,NI) {NC==17, NI==4, 1<=L, 1<=N}
         L = L+1
C     P(K,L,N,NI){NC==17, NI==4, 2<=L, 1<=N}
         M = 2*K+1
C     P(K,L,M,N,NI)  {2K==M-1, NC==17, NI==4, 2<=L, 1<=N}
         K = K+1
C     P(K,L,M,N,NI)  {2K==M+1, NC==17, NI==4, 2<=L, 1<=N}
         CALL EXTR(NI,NC)
      ENDIF
C     P(K,L,M,N,NI) {NC==17}
      END

C     P() {2K==M+1, NC==17, 2<=L}
      SUBROUTINE EXTR(NI,NC)
C     P() {2K==M+1, NC==17, 2<=L}
      DO I = L,K
C     P(I,J) {2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(I,1,NC) = I*NC
C     P(I,J) {2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(I,1,NC+1) = I*(NC+1)
C     P(I,J) {2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         J = L+M-I
C     P(I,J) {I+J==2K+L-1, 2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(J,1,NC) = A(I,1,NC)
C     P(I,J) {I+J==2K+L-1, 2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(J,1,NC+1) = A(I,1,NC+1)
      ENDDO
C     P(I,J) {2K==M+1, NC==17, K+1<=I, L<=I, 2<=L}
      END
```

Fig. 3. Preconditions of the running example (declarations omitted).

current call site is available and is translated into the frame of the called procedure. The process is similar to transformer translation. Equations between actual and formal parameters are added. Local variables of the caller are eliminated which may cause some information loss. Global variables are renamed when possible.

The summary precondition of the called procedure is the convex hull of the translated preconditions of all its call sites. This summary precondition, derived from the calling contexts, becomes the initial precondition of the procedure.

Figure 3 shows the interprocedural preconditions computed for our running example. `{NC==17}` is the initial precondition of `MAIN`. The test condition `L.GT.1.AND.N .GE.1` is polyhedric and is added to the precondition of the true branch. This precondition is propagated to the call to `EXTR` and is translated to its frame. Since `NI` and `N` are not used by `EXTR`, constraints on them are not kept in the summary precondition of `EXTR`. The last precondition in `EXTR` is the

loop postcondition. We have {K+1<=I, L<=I}, the convex hull of two predicates obtained when the loop is entered or not: {I==K+1, L<=K} and {I==L, K+1<=L}.

One main advantage of transformer and precondition analyses is that we can deduce from the program semantics information that is not stated explicitly. A close approach was initially developed by Karr [1976] and Cousot and Halbwachs [1978] that tried to find the affine relationship among variables in a program state. 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 array elements used and defined by elementary and compound statements of programs. The region of an array A of n dimensions at a statement s is a function from the set of states to the power set of $\mathbb{Z}^n$:

$$\begin{aligned} \mathcal{R} : \mathit{Statement} &\longrightarrow \quad \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ s &\longmapsto \quad \lambda\sigma.(\{\phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma)\}). \end{aligned}$$

The semantic function $\mathcal{R}$ associates to each statement s and to each program state σ a set of array elements described by the *region parameters* $\phi = (\phi_1, \ldots, \phi_n)$ vector. The variable ϕ_i represents the ith dimension index. r is the relationship existing between ϕ and the current program state σ.

Since array region analysis was introduced to support dependence analyses on array structures, two kinds of effects on array elements are used: READ if they are used and WRITE if they are defined. Unlike transformers and preconditions, both under- and overapproximations are computed for array regions. A region has the approximation MUST if every element in the region is accessed with certainty, and the approximation MAY if its elements are simply potentially accessed. The approximation of a region is EXACT if the region exactly represents the requested set of array elements.

$$\begin{aligned} \underline{\mathcal{R}}, \overline{\mathcal{R}} : \mathit{Statement} &\longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ \underline{\mathcal{R}}(s)(\sigma) &\subseteq \mathcal{R}(s)(\sigma) \subseteq \overline{\mathcal{R}}(s)(\sigma). \end{aligned}$$

Array regions are approximated by using convex polyhedra. The constraints link the region parameters that represent the array dimensions to the value of the program integer scalar variables. For example, WRITE-EXACT-{PHI1==1,PHI2==I} is the written array region of statement A(1,I) = 5.

Regions are built bottom-up from the deepest nodes to the largest compound statement nodes in the hierarchical control flow graph. 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.

3.3.1 *Elementary Statement.* Each array reference in an assignment is converted to an elementary region. This region is EXACT if and only if the

subscript expressions are affine functions of the program integer scalar variables. The effect depends on the reference position in the statement. To save space, regions of the same array with the same effect are merged by using the convex hull operator. If the resulting region contains array elements that do not belong to the original regions, it becomes a MAY region.

3.3.2 *Conditional Statement.* The array regions of a conditional statement contain the regions of the test condition, plus the unified regions of the true and false branches restrained by the test condition evaluation. EXACT regions may become MAY regions.

3.3.3 *Loop Statement.* The region corresponding to the body of a loop is a function of the loop index value. During the propagation of regions, we need to unify regions corresponding to different, but successive, instances of the loop body in order to get the summary region of the loop. This union operation is equivalent to eliminating the loop index i from the system which consists of the region predicate and the constraint on the iteration space: $lb \leq i \leq ub$. The operation is exact if the lower and upper bounds of the loop index are affine functions and if the elimination of loop index is exact [Ancourt and Irigoin 1991; Pugh 1992].

3.3.4 *Sequence of Statements.* Array regions at different program points correspond to different program states. To obtain array regions of a sequence of statements, we must translate array regions associated to different statements to a reference state, for example, the state preceding the sequence. The translation uses transformers and consists of adding to the predicate of the region the constraints abstracting the effects of the statement between the two states. Then variables of the original state are eliminated. Only variables referring to the reference state are kept in the resulting polyhedron. The projection of variables from a convex polyhedron may introduce integer points that do not belong to the actual projection. In this case, we have a MAY region.

3.3.5 *Interprocedural Array Regions.* The interprocedural propagation of READ and WRITE regions is a reverse invocation order traversal on the call graph. The callees are analyzed first. The summary region of a procedure is computed by eliminating local effects from the region for the procedure body. At each call site, the summary regions of the called subroutine are translated from the callee's name space into the caller's name space, and then used during the intraprocedural analysis of the callers.

Figure 4 illustrates the interprocedural array region computation of the running example. Each statement is preceded by its regions. Accesses to array A occur in the loop body of EXTR. The merged region of READ regions of A is EXACT while that of WRITE regions is MAY because unreferenced elements with PHI1 in maybe nonempty range [I+1,2K+L-I-2] are added in the convex hull {I<=PHI1, PHI1+1+I<=2K+L}. These regions are unified over different iterations, propagated up to have the summary regions of EXTR. After the interprocedural translation, the regions before the call site to EXTR are propagated up, and the READ region becomes MAY because, if the test condition is false, no array elements are used.

```
  PROGRAM MAIN
C <A(PHI1,PHI2,PHI3)-READ-MAY-{PHI2==1, 2<=PHI1, NC<=PHI3, PHI3<=NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1, 2<=PHI1, NC<=PHI3, PHI3<=NC+1}>
  READ *,K,L,N
C <A(PHI1,PHI2,PHI3)-READ-MAY-{PHI2==1,1+L<=PHI1,PHI1<=1+K,NC<=PHI3,PHI3<=NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,1+L<=PHI1,PHI1<=1+2K,NC<=PHI3,PHI3<=NC+1}>
  IF (L.GE.1.AND.N.GE.1) THEN
C <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,1+L<=PHI1,PHI1<=1+K,NC<=PHI3,PHI3<=NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,1+L<=PHI1,PHI1<=1+2K,NC<=PHI3,PHI3<=NC+1}>
C    BEGIN BLOCK
     NI = 4
     L = L+1
     M = 2*K+1
     K = K+1
C <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,L<=PHI1,PHI1<=K,NC<=PHI3,PHI3<=NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,L<=PHI1,PHI1+1<=2K,NC<=PHI3,PHI3<=NC+1}>
     CALL EXTR(NI,NC)
C    END BLOCK
  ENDIF
  END

C <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,L<=PHI1,PHI1<=K,NC<=PHI3,PHI3<=NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,L<=PHI1,PHI1+1<=2K,NC<=PHI3,PHI3<=NC+1}>
  SUBROUTINE EXTR(NI,NC)
C <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,L<=PHI1,PHI1<=K,NC<=PHI3,PHI3<=NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,L<=PHI1,PHI1+1<=2K,NC<=PHI3,PHI3<=NC+1}>
  DO I = L,K
C <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI1==I,PHI2==1, NC<=PHI3,PHI3<=NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,I<=PHI1,PHI1+1+I<=2K+L,NC<=PHI3,PHI3<=NC+1}>
C    BEGIN BLOCK
C <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==I, PHI2==1, PHI3==NC}>
     A(I,1,NC) = I*NC
C <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==I, PHI2==1, PHI3==NC+1}>
     A(I,1,NC+1) = I*(NC+1)
     J = L+M-I
C <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI1==I, PHI2==1, PHI3==NC}>
C <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==J, PHI2==1, PHI3==NC}>
     A(J,1,NC) = A(I,1,NC)
C <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI1==I, PHI2==1, PHI3==NC+1}>
C <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==J, PHI2==1, PHI3==NC+1}>
     A(J,1,NC+1) = A(I,1,NC+1)
C    END BLOCK
  ENDDO
  END
```

Fig. 4. Array regions of the running example (declarations omitted).

Array access analysis is also studied in Allen et al. [1988], Callahan and Kennedy [1988], Feautrier [1991], Maydan et al. [1993], Duesterwald et al. [1993], Tu and Padua [1995], Leservot [1996], and Paek et al. [2002]. There are different approximations of the set of accessed array elements such as convex polyhedra [Triolet et al. 1986, Creusillet and Irigoin 1995, 1996], regular section descriptors and guarded regular section descriptors [Nguyen et al. 1995; Gu and Li 2000], symbolic array section [Gupta et al. 2000], and data access descriptors and linear memory access descriptor [Hoeflinger et al. 2001], etc. The techniques differ in the amount of precision as well as the efficiency on storage and time. They were compared empirically in Hind et al. [1994]. For example,

in Hall et al. [1995], array element sets were represented by lists of polyhedra and there was no exact representation, only under- and overapproximations. In Nguyen et al. [1995] and Gu and Li [2000], array element sets were a list of RSDs (Regular Section Descriptor) with bounds and step, guarded by predicates derived from IF conditions. Lin and Padua [1999] proposed an analysis finding properties of arrays that appear in the subscripts of other arrays (indirection arrays). In Manjunathaiah and Nicole [1997], precision was improved by keeping the complementary array sections when computing the envelope convex union. The abstraction choice was a tradeoff between efficiency and precision. Convex array region analyses are accurate enough to provide useful information for the array region based bound checking that is described in Section 5.

4. ELIMINATION OF REDUNDANT TESTS

As explained un Section 2, the idea of this approach is to insert systematically a set of tests at each array reference, then to compute preconditions for each instrumented statement, and finally to eliminate tests with conditions redundant with respect to their preconditions. Since exact preconditions cannot be computed (otherwise the halting problem could be solved), we deal with overapproximated preconditions. Two theorems necessary to show the correctness of our algorithm are presented first and then our algorithm is detailed.

4.1 Static Safety and Error

Let $\mathcal{VC}$ be the semantic function that represents the array bound violation condition. It associates to each statement a set of program states that cause array bound violations.

$$\mathcal{VC} : Statement \longrightarrow \wp(State).$$

As defined, the precondition of a statement and its overapproximation are also functions from the set of statements to the powerset of the program state set:

$$\mathcal{P}, \overline{\mathcal{P}} : Statement \longrightarrow \wp(State).$$

Let s be a statement, and δ a program state; we have the two following theorems:

THEOREM 4.1.1. *For any statement, if the intersection of the overapproximated precondition and the violation condition is empty then there is no bound violation caused by this statement; the theorem is equivalent to:*

$$(\overline{\mathcal{P}}(s) \cap \mathcal{VC}(s) = \emptyset) \Longrightarrow (\forall \delta \in \mathcal{P}(s) : \delta \notin \mathcal{VC}(s)).$$

PROOF. The hypotheses $\overline{\mathcal{P}}(s) \cap \mathcal{VC}(s) = \emptyset$ imply

$$\forall \delta \in \overline{\mathcal{P}}(s) : \delta \notin \mathcal{VC}(s). \tag{1}$$

On the other hand, following the definition of the overapproximated precondition:

$$\forall \delta \in \mathcal{P}(s) : \delta \in \overline{\mathcal{P}}(s). \tag{2}$$

From (1) and (2), we have

$$\forall \delta \in \mathcal{P}(s) : \delta \notin \mathcal{VC}(s),$$

which means that any program state reaching s does not belong to the set of states causing bound violations at s. So there is no bound violation caused by this statement. □

THEOREM 4.1.2. *For any statement, if the intersection of the overapproximated precondition and the negation of the violation condition is empty then there is surely a bound violation caused by this statement if it is executed; the theorem is equivalent to*

$$(\overline{\mathcal{P}}(s) \cap \neg \mathcal{VC}(s) = \emptyset) \Longrightarrow (\forall \delta \in \mathcal{P}(s) : \delta \in \mathcal{VC}(s)).$$

PROOF. The hypotheses $\overline{\mathcal{P}}(s) \cap \neg \mathcal{VC}(s) = \emptyset$ imply

$$\forall \delta \in \overline{\mathcal{P}}(s) : \delta \in \mathcal{VC}(s) \tag{3}$$

because if $\delta \notin \mathcal{VC}(s)$ then $\delta \in \neg \mathcal{VC}(s)$ and $\delta \in \overline{\mathcal{P}}(s) \cap \neg \mathcal{VC}(s)$, which contradicts the hypotheses. On the other hand:

$$\forall \delta \in \mathcal{P}(s) : \delta \in \overline{\mathcal{P}}(s). \tag{4}$$

From (3) and (4), we have

$$\delta \in \mathcal{P}(s) \Longrightarrow \delta \in \mathcal{VC}(s),$$

which means that every program state reaching s is in the set of states that cause bound violations at s. So there is certainly a bound violation caused by this statement if it is executed. □

4.2 Elimination Algorithm

Our first implementation of a range check optimizer consists of two phases: *bound checks generation* and *redundant code elimination*. Algorithm 4.2.1 describes the elimination of redundant tests, based on Theorem 4.1.1 and Theorem 4.1.2.

ALGORITHM 4.2.1.

```
procedure Elimination_of_Redundant_Tests(p)
  p : current procedure
begin
  for each statement s of p
    for each array reference A(s_1, ..., s_n) in s
      for each dimension i
        l_i = lower_bound_dimension(A, i)
        u_i = upper_bound_dimension(A, i)
        insert "IF (s_i < l_i) STOP message" before s
        insert "IF (s_i > u_i) STOP message" before s
      endfor
    endfor
  endfor
  compute preconditions for each statement of instrumented program p
```

```
for each inserted test statement t with condition e of instrumented p
    P = precondition(t)
    sc1 = P ∩ {e}
    sc2 = P ∩ {¬e}
    switch
        case infeasible(sc1) : // Theorem 4.1.1
            no bound violation, remove t from p
        case infeasible(sc2) : // Theorem 4.1.2
            bound violation found at compile-time
        default : /*undecidable, keep the dynamic check*/
    endswitch
endfor
end
```

In the generation phase, only nontrivial bound checks are generated. Trivial tests which are always false such as `2<1` or `N-1>N` are not taken into account. Each bound check is accompanied with a stop message and, if a bound violation is detected, the message tells the user in which array, on which dimension, on which bound, and in which line the subscript is out of range.

When computing the preconditions for the new code, we have two options: intraprocedural and interprocedural analyses. In the intraprocedural option, the preconditions at the call sites are not taken into account. The conservative precondition `P(){}`, which represents all possible program states, is used instead of the more precise preconditions deduced from the call sites with the interprocedural option. Information provided by these preconditions is used to detect bound violations or eliminate redundant checks. The precondition of a given statement incorporates information propagated from the test condition of the bound checking statements inserted before this statement. For each bound check, we test the feasibility of the system built from it and the precondition. The feasibility test of a system of constraints is implemented in PIPS by using the Simplex and Fourier-Motzkin algorithms [Schrijver 1986]. There are three possibilities:

(1) If the system is infeasible, the bound check is false and is removed from the test condition (Theorem 4.1.1).
(2) If the bound check is true with respect to the precondition, that is, the system built from the precondition and the negation of the bound check is infeasible, a bound violation is detected at compile-time (Theorem 4.1.2).
(3) Otherwise, the bound check is preserved.

Figure 5 shows an excerpt from *swim*, a weather prediction program in the SPEC95 CFP benchmark [Dujmovic and Dujmovic 1998] with procedure calls and array references, which are used to show the effect of interprocedural analyses.

After the bound check generation, the instrumented code with intraprocedural and interprocedural preconditions are represented, respectively, in Figure 6 and Figure 8. The preconditions are in comment lines. Some trivial checks that are never true such as `1.LT.1` and `1.GT.513` are not generated by our bound checker. The codes after the redundancy elimination for the intraprocedural and interprocedural versions are given in Figure 7 and Figure 9,

```
 1    PROGRAM SHALOW
 2    PARAMETER (N1=513, N2=513)
 3    COMMON  U(N1,N2)
 4    COMMON /CONS/ M,N
 5    CALL INITAL
 6    MN = MIN(M,N)
 7    UCHECK = 0.0
 8    DO I = 1, MN
 9       DO J = 1, MN
10          UCHECK = UCHECK + U(I,J)
11       ENDDO
12    ENDDO
13    END
14    SUBROUTINE INITAL
15    PARAMETER (N1=513, N2=513)
16    COMMON  U(N1,N2)
17    COMMON /CONS/ M,N
18    READ (5,*) M, N
19    IF (M.LE.512.AND.N.LE.512) THEN
20       DO I = 1, M
21          DO J = 1, N
22             U(I,J) = I*J
23          ENDDO
24       ENDDO
25       U(M+1,N+1) = U(1,1)
26    ELSE
27       STOP
28    ENDIF
29    END
```

Fig. 5. Array bound checking example (excerpted from *swim*).

respectively. Since there is no difference for module `INITAL`, it is omitted in the interprocedural version (Figure 8 and Figure 9). In both versions, the preconditions `{1<=I, 1<=J}` after entering the loops in `SHALOW` and `INITAL` allow us to remove all lower bound checks. The difference between the intraprocedural and the interprocedural options of preconditions is that, in Figure 6, after the call to `INITAL`, we have an empty postcondition, while in Figure 8 we have `P(M,N) {0<=M, M<=512, 0<=N, N<=512}`. This more precise postcondition helps to eliminate all upper bound checks (`I.GT.N1` and `J.GT.N2`) in the nested loop in `SHALOW` (Figure 9). Here we see the strength of the interprocedural precondition analysis because in the intraprocedural option, bound checks are left inside loops (Figure 7), as long as code hoisting has not been applied. However, for the subroutine `INITAL`, since we have no information about the lower bounds of variables `M` and `N`, lower bound tests remain.

After the 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 to detect out-of-bound errors. Experimental results with the benchmark SPEC95 CFP are given in Section 7.

5. INSERTION OF UNAVOIDABLE TESTS

The basic idea for the second approach is to try to check array overflows on array region (see Section 3.3) at the highest possible compound statement, the procedure statement, and then go down into substatements. In this way, array access checks are hoisted toward the less frequently executed statements. However, the highest the compound statement the less accurate is the associated array region. Before presenting the algorithm, we give two theorems necessary to deal correctly with imprecise information.

5.1 Static Safety and Error

The second array bound checker is based on the convex array region analysis. As defined in Section 3, the region of an array A of n dimensions at a statement

```
      PROGRAM SHALOW
      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 I = 1, MN
C     P(I,J,M,MN,N) {1<=I, I<=MN, N<=M, MN<=N}
        DO 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) STOP "Bound violation: array U, lower dimension 2, line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, MN<=M, MN<=N}
          IF (J.GT.N2) STOP "Bound violation: array U, upper dimension 2, 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) STOP "Bound violation: array U, lower dimension 1, 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.GT.N1) STOP "Bound violation: array U, upper dimension 1, 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 + U(I,J)
        ...
      SUBROUTINE INITAL
      READ (5,*) M, N
C     P(M,N) {}
      IF (M.LE.512.AND.N.LE.512) THEN
C     P(M,N) {M<=512, N<=512}
        DO I = 1, M
C     P(I,J,M,N) {1<=I, I<=M, M<=512, N<=512}
          DO J = 1, N
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=N, M<=512, N<=512}
            IF (J.LT.1) STOP "Bound violation: array U, lower dimension 2, line 22"
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=N, M<=512, N<=512}
            IF (J.GT.N2) STOP "Bound violation: array U, upper dimension 2, line 22"
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=513, J<=N, M<=512, N<=512}
            IF (I.LT.1) STOP "Bound violation: array U, lower dimension 1, line 22"
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=513, J<=N, M<=512, N<=512}
            IF (I.GT.N1) STOP "Bound violation: array U, upper dimension 1, line 22"
C     P(I,J,M,N) {1<=I, I<=513, I<=M, 1<=J, J<=513, J<=N, M<=512, N<=512}
            U(I,J) = I*J
          ...
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, N<=512}
        IF (N+1.LT.1) STOP "Bound violation: array U, lower dimension 2, line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=N}
        IF (N+1.GT.N2) STOP "Bound violation: array U, upper dimension 2, line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=N, N<=512}
        IF (M+1.LT.1) STOP "Bound violation: array U, lower dimension 1, line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=M, 0<=N, N<=512}
        IF (M+1.GT.N1) STOP "Bound violation: array U, upper dimension 1, line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=M, M<=512, 0<=N, N<=512}
        U(M+1,N+1) = U(1,1)
      ELSE
C     P(M,N) {}
        STOP
      ENDIF
```

Fig. 6. Elimination of redundant tests with intraprocedural preconditions: code with generated bound checks and preconditions (declarations omitted).

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
UCHECK = 0.0
DO I = 1, MN
   DO J = 1, MN
      IF (J.GT.N2) STOP "Bound violation: array U, upper dimension 2, line 10"
      IF (I.GT.N1) STOP "Bound violation: array U, upper dimension 1, line 10"
      UCHECK = UCHECK + U(I,J)
   ENDDO
ENDDO
END

SUBROUTINE INITAL
READ (5,*) M, N
IF (M.LE.512.AND.N.LE.512) THEN
   DO I = 1, M
      DO J = 1, N
          U(I,J) = I*J
      ENDDO
   ENDDO
   IF (N+1.LT.1) STOP "Bound violation: array U, lower dimension 2, line 25"
   IF (M+1.LT.1) STOP "Bound violation: array U, lower dimension 1, line 25"
   U(M+1,N+1) = U(1,1)
ELSE
   STOP
ENDIF
END
```

Fig. 7. Elimination of redundant tests with intraprocedural preconditions: code after redundancy elimination (declarations omitted).

s is a function from the set of states to the powerset of $\mathbb{Z}^n$:

$$\begin{aligned} \mathcal{R} : Statement &\longrightarrow State \longrightarrow \wp(\mathbb{Z}^n) \\ s &\longmapsto \lambda\sigma.(\{\phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma)\}). \end{aligned}$$

The violation condition of an array A associates to each statement a set of program states that cause bound violations of this array:

$$\mathcal{VC} : Statement \longrightarrow \wp(State).$$

By using the region vector ϕ, $\mathcal{VC}(s)$ is defined by the following set:

$$\{\sigma : \exists \phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n \; \exists i = 1, n \;\; r(\phi, \sigma) \wedge ((\phi_i < \mathcal{E}(l_i)(\sigma)) \vee (\phi_i > \mathcal{E}(u_i)(\sigma)))\}$$

where r is the region relationship existing between ϕ and the current program state. l_i and u_i, are, respectively, the lower and upper bounds of the dimension i of array A. $\mathcal{E}(e)$ is the function returning the value of an expression e in a program state σ.

Given a procedure, the declaration of an array can be defined as

$$\begin{aligned} \mathcal{DEC} : State &\longrightarrow \wp(\mathbb{Z}^n) \\ \sigma &\longmapsto \{\phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n : \forall i = 1, n \;\; \mathcal{E}(l_i)(\sigma) \leq \phi_i \leq \mathcal{E}(u_i)(\sigma)\}. \end{aligned}$$

We have the two following theorems:

```
      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 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 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) STOP "Bound violation: array U, lower dimension 2, line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, M<=512, MN<=M, MN<=N, N<=512}
            IF (J.GT.N2) STOP "Bound violation: array U, upper dimension 2, line 10"
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) STOP "Bound violation: array U, lower dimension 1, line 10"
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.GT.N1) STOP "Bound violation: array U, upper dimension 1, line 10"
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 + U(I,J)
         ENDDO
      ENDDO
      END
```

Fig. 8. Elimination of redundant tests with interprocedural preconditions: code with generated bound checks and preconditions (declarations omitted).

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
UCHECK = 0.0
DO I = 1, MN
   DO J = 1, MN
      UCHECK = UCHECK + U(I,J)
   ENDDO
ENDDO
END
```

Fig. 9. Elimination of redundant tests with interprocedural preconditions: code after redundancy elimination (declarations omitted).

THEOREM 5.1.1. *For any statement, if the overapproximated region of an array is included in the declared dimensions of the array then there is no bound violation of the array caused by this statement; the theorem is equivalent to*

$$(\forall \delta : \overline{\mathcal{R}}(s)(\sigma) \subseteq \mathcal{DEC}(\sigma)) \Longrightarrow (\mathcal{VC}(s) = \emptyset).$$

PROOF. From the hypotheses, for all δ, we have

$$\overline{\mathcal{R}}(s)(\sigma) \subseteq \mathcal{DEC}(\sigma). \tag{5}$$

On the other hand, following the definition of the overapproximated region:

$$\mathcal{R}(s)(\sigma) \subseteq \overline{\mathcal{R}}(s)(\sigma). \tag{6}$$

From (5) and (6), we have

$$\forall \delta : \mathcal{R}(s)(\sigma) \subseteq \mathcal{DEC}(\sigma),$$

$$\forall \delta \ \forall \phi = (\phi_1, \ldots, \phi_n) : r(\phi, \delta) \Longrightarrow \forall i = 1, n \ \mathcal{E}(l_i)(\sigma) \leq \phi_i \leq \mathcal{E}(u_i)(\sigma),$$

$$\nexists \delta \ \nexists \phi = (\phi_1, \ldots, \phi_n) \ \nexists i = 1, n : r(\phi, \delta) \wedge ((\phi_i < \mathcal{E}(l_i)(\sigma)) \vee (\phi_i > \mathcal{E}(u_i)(\sigma))),$$

which leads to $\mathcal{VC}(s) = \emptyset$. There is no array bound violation caused by s. □

THEOREM 5.1.2. *For any statement, if the underapproximated region of an array contains elements which are outside the declared dimensions of the array then there is certainly a bound violation of the array caused by this statement; the theorem is equivalent to*

$$(\exists \delta \ \exists \phi : \phi \in \underline{\mathcal{R}}(s)(\sigma) \wedge \phi \notin \mathcal{DEC}(\sigma)) \Longrightarrow (\mathcal{VC}(s) \neq \emptyset).$$

PROOF. The hypotheses is

$$\exists \delta \ \exists \phi : \phi \in \underline{\mathcal{R}}(s)(\sigma) \wedge \phi \notin \mathcal{DEC}(\sigma). \tag{7}$$

On the other hand, following the definition of $\underline{\mathcal{R}}$:

$$\underline{\mathcal{R}}(s)(\sigma) \subseteq \mathcal{R}(s)(\sigma). \tag{8}$$

From (7) and (8), we have

$$\exists \delta \ \exists \phi : \phi \in \mathcal{R}(s)(\sigma) \wedge \phi \notin \mathcal{DEC}(\sigma)$$

$$\exists \delta \ \exists \phi = (\phi_1, \ldots, \phi_n) : r(\phi, \sigma) \wedge (\exists i = 1, n : (\phi_i < \mathcal{E}(l_i)(\sigma)) \vee (\phi_i > \mathcal{E}(u_i)(\sigma))),$$

which leads to $\mathcal{VC}(s) \neq \emptyset$ by definition. There is an array bound violation caused by this statement. □

5.2 Insertion Algorithm

Array 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 an answer about the ranges of 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*. Since transformers are used to model the effects of state transitions, we have two options to compute array regions: intraprocedural and interprocedural transformer analyses. The insertion of unavoidable tests is given by Algorithm 5.2.1, based on Theorems 5.1.1 and 5.1.2.

ALGORITHM 5.2.1.
procedure *Insertion_of_Unavoidable_Tests(p)*
p : current procedure, decorated with regions at every statement

```
begin
    s = the compound statement of p
    insertion_of_unavoidable_tests_at_statement(s)
end
procedure insertion_of_unavoidable_tests_at_statement(s)
    for each region R of array A in list of array regions at s
        for each bound check e(φ_i < l_i or φ_i > u_i) at dimension i of array A
            sc_1 = R ∩ {e}
            sc_2 = R ∩ {¬e}
            switch
                case infeasible(sc_1) and (R = MAY or EXACT) : // Theorem 5.1.1
                    no bound violation of A at s
                case infeasible(sc_2) and (R = MUST or EXACT) : // Theorem 5.1.2
                    bound violation of A
                case R = EXACT :
                    sc = project_phi_variables(sc_1)
                    if the projection is exact then
                        insert "IF (sc) STOP message" before s
                    else
                        for each sub-statement ss of s
                            insertion_of_unavoidable_tests_at_statement(ss)
                        endfor
                    endif
                default:
                    for each sub-statement ss of s
                        insertion_of_unavoidable_tests_at_statement(ss)
                    endfor
            endswitch
        endfor
    endfor
end
```

At a compound statement, the read and write regions of each array are used to test the feasibility of the corresponding array bound checks.

(1) If the region is a MAY or EXACT region included in the declared dimensions of the array, no bound check is needed for the compound statement and we stop the process for the array here (Theorem 5.1.1).

(2) If the region is a MUST or EXACT region that contains elements outside the declared dimensions of the array, there is certainly a bound violation. An error is detected at compile-time (Theorem 5.1.2).

(3) If the region is an EXACT region and it is possible to project all pseudovariables ϕ from the system sc_1, we have unavoidable tests to insert before the compound statement. The projection of one variable from a system of constraints is a linear programming operation, performed by using the Fourier-Motzkin method. 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. We have special variables to store information that the lower or upper

```
PROGRAM REGION                          PROGRAM REGION
COMMON ITAB(10),J                       COMMON ITAB(10),J
REAL A(10)                              REAL A(10)
READ *,M                          C     <ITAB(PHI1)-WRITE-MAY-{1<=PHI1}>
J = 11                            C     <A(PHI1)-WRITE-EXACT-{PHI1==11}>
DO I = 1,M                              READ *,M
   ITAB(I) = 1                    C     <ITAB(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=M}>
ENDDO                             C     <A(PHI1)-WRITE-EXACT-{PHI1==11}>
A(J) = 0                                J = 11
END                               C     <ITAB(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=M}>
                                  C     <A(PHI1)-WRITE-EXACT-{PHI1==J}>
                                        DO I = 1,M
                                           ITAB(I) = 1
                                        ENDDO
                                  C     <A(PHI1)-WRITE-EXACT-{PHI1==J}>
                                        A(J) = 0
                                        END
```

Fig. 10. Example with maybe incorrect regions.

bounds of a dimension of an array have already been checked or not. If so, we do not have to check for these bounds when going down to the substatements.

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

However, 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 example in Figure 10 shows how an array bound violation can lead to an unsafe propagation of array `A` regions. If every access to `ITAB` is within its bounds, the propagated regions of array `A` before the `DO` loop, the assignment `J = 11` and `READ *,M` are correct. Otherwise, when $\texttt{M} \geq 11$, the bound violation in `ITAB` modifies the value of `J` and these regions are not correct any more. 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 *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 order of array definitions (*write order*) is used to decide which array region is checked first. We know that the propagated regions for array `A` become false only when an element outside the declared range of `ITAB` is written. So if bound checks for `ITAB` are generated before testing regions of `A`, there is no problem. The code with unavoidable tests is shown in Figure 11.

```
PROGRAM REGION
COMMON ITAB(10),J
REAL A(10)
READ *,M
IF (11.LE.M) STOP "Bound violation: array ITAB, 1st dimension"
STOP "Bound violation: array A, 1st dimension"
J = 11
DO I = 1,M
      ITAB(I) = 1
ENDDO
A(J) = 0
END
```

Fig. 11. Example with instrumented code.

This is very similar to code hoisting; we are only allowed to hoist code up to an earlier point if it is safe to do so. The regions of an array are considered only when every array whose element assignment occurs at least once before the concerned array assignment has been already checked. In case the write order cannot be established for a compound statement, we have to go down to the substatements of the current statement. For instance, with the sequence of statements

```
s1     A(I) = I
s2     B(M,N) = M + N
s3     A(J) = J
```

not every element of A is always written before elements of B, so we have to go down and check the array region for A at s1, array region for B, and then for A at the beginning of sequence s2; s3. Algorithm 5.2.1 is refined by taking into account this write order.

Figures 12 and 13, and Figures 14 and 15, show the running example with the insertion of unavoidable tests approach, respectively, for the intraprocedural and interprocedural transformer options. The result is the same for INITAL, so it is omitted in the interprocedural option (Figures 14 and 15).

For the procedure INITAL, we have two kinds of regions, READ and WRITE for array U, and these regions are treated separately. Lower bound checks remain for the write region U(PHI1,PHI2)-WRITE-MAY-{PHI1<=513, PHI2<=513} and unavoidable tests are inserted before statement U(M+1,N+1) = U(1,1). For the procedure SHALOW in the intraprocedural version, the cumulated region of the nested loop allows us to generate an upper bound check that is outside the loop (Figure 13). Compared to the intraprocedural version of the first approach, this is a point in favor of the second approach because it may lift tests out of loops automatically while the former does not.

In the interprocedural version, the interprocedural transformer {M<=512,N<=512} of INITAL integrated in the cumulated region before the call to INITAL decides 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.

```
      PROGRAM SHALOW
C     <U(PHI1,PHI2)-READ-MAY-{1<=PHI1, 1<=PHI2}>
      CALL INITAL
C     <U(PHI1,PHI2)-READ-MAY-{1<=PHI1, PHI1<=M, PHI1<=N, 1<=PHI2, PHI2<=M, PHI2<=N}>
      MN = MIN(M, N)
C     <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
      UCHECK = 0.0
C     <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
      DO I = 1, MN
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, 1<=PHI2, PHI2<=MN}>
         DO J = 1, MN
C        <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, PHI2==J}>
            UCHECK = UCHECK + U(I,J)
         ENDDO
      ENDDO
      END

      SUBROUTINE INITAL
C     <U(PHI1,PHI2)-READ-MAY-{PHI1==1, PHI2==1}>
C     <U(PHI1,PHI2)-WRITE-MAY-{PHI1<=513, PHI2<=513}>
      READ (5,*) M, N
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==1, PHI2==1, M<=512, N<=512}>
C     <U(PHI1,PHI2)-WRITE-MAY-{PHI1<=M+1, PHI2<=N+1, M<=512, N<=512}>
      IF (M.LE.512.AND.N.LE.512) THEN
C     <U(PHI1,PHI2)-WRITE-EXACT-{1<=PHI1, PHI1<=M, 1<=PHI2, PHI2<=N}>
         DO I = 1, M
C        <U(PHI1,PHI2)-WRITE-EXACT-{PHI1==I, 1<=PHI2, PHI2<=N}>
            DO J = 1, N
C           <U(PHI1,PHI2)-WRITE-EXACT-{PHI1==I, PHI2==J}>
               U(I,J) = I*J
            ENDDO
         ENDDO
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==1, PHI2==1}>
C     <U(PHI1,PHI2)-WRITE-EXACT-{PHI1==M+1, PHI2==N+1}>
         U(M+1,N+1) = U(1,1)
      ELSE
         STOP
      ENDIF
      END
```

Fig. 12. Insertion of unavoidable tests with intraprocedural transformers: code with array regions (declarations omitted).

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 abstractions of array element sets, 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 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 at the expense of information about errors.

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
IF (514.LE.MN) STOP "Bound violation: array U, 2nd dimension"
UCHECK = 0.0
DO I = 1, MN
   DO J = 1, MN
      UCHECK = UCHECK + U(I,J)
   ENDDO
ENDDO
END

SUBROUTINE INITAL
READ (5,*) M, N
IF (M.LE.512.AND.N.LE.512) THEN
   DO I = 1, M
      DO J = 1, N
          U(I,J) = I*J
      ENDDO
   ENDDO
   IF (N+1.LT.1) STOP "Bound violation: array U, 2nd dimension"
   IF (M+1.LT.1) STOP "Bound violation: array U, 1st dimension"
   U(M+1,N+1) = U(1,1)
ELSE
   STOP
ENDIF
END
```

Fig. 13. Insertion of unavoidable tests with intraprocedural transformers: code with unavoidable tests (declarations omitted).

```
   PROGRAM SHALOW
C  <U(PHI1,PHI2)-READ-MAY-{1<=PHI1, PHI1<=512, 1<=PHI2, PHI2<=512}>
   CALL INITAL
C  <U(PHI1,PHI2)-READ-MAY-{1<=PHI1, PHI1<=M, PHI1<=N, 1<=PHI2, PHI2<=M, PHI2<=N}>
   MN = MIN(M, N)
C  <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
   UCHECK = 0.0
C  <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
   DO I = 1, MN
C  <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, 1<=PHI2, PHI2<=MN}>
      DO J = 1, MN
C  <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, PHI2==J}>
         UCHECK = UCHECK + U(I,J)
      ENDDO
   ENDDO
   END
```

Fig. 14. Insertion of unavoidable tests with interprocedural transformers: code with array regions (declarations omitted).

6. ACTUAL/FORMAL ARRAY SIZE CHECKING

Within a program unit, the declaration given for an array provides all the 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

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
UCHECK = 0.0
DO I = 1, MN
   DO J = 1, MN
      UCHECK = UCHECK + U(I,J)
   ENDDO
ENDDO
END
```

Fig. 15. Insertion of unavoidable tests with interprocedural transformers: code with unavoidable tests (declarations omitted).

```
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 I = 1,N
   DO J = 1,M
       X(I,J) = 1.
   ENDDO
ENDDO
END
```

Fig. 16. Actual/formal array size mismatch example.

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 16 illustrates a typical array size violation found in a real application.

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 can be used voluntarily as in Figure 16, but our implementation found an actual/formal array size mismatch in 1 out of the 10 benchmarks from SPEC95 CFP. Furthermore, the fact that bugs related to interprocedural mismatch are much more difficult to track than standard array violation is another reason to provide this actual/formal array size checking. Our algorithm is closely related to Algorithm 4.2.1. Tests of conditions required by the Fortran standard argument association rules are systematically inserted before each call site. Over approximations of preconditions are computed and unnecessary tests are eliminated using these approximations.

6.1 Argument Association Rules

The relationship between the size of the formal and the actual arrays is defined by the association rules of formal and actual arguments in Section 15.9.3.3 of the Fortran 77 standard [ANSI 1983]. A formal array can be associated to an actual array or to 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. 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 application of the above conditions is not straightforward

Table I. Summary of Actual and Formal Arrays

Array name	A	B
Number of dimensions	n	m
Lower bounds	$l_{a1}, \ldots, l_{an}$	$l_{b1}, \ldots, l_{bm}$
Upper bounds	$u_{a1}, \ldots, u_{an}$	$u_{b1}, \ldots, u_{bm}$
Element size	e_a	e_b
Dimension size	$d_{ai} = u_{ai} - l_{ai} + 1$	$d_{bi} = u_{bi} - l_{bi} + 1$
Array size	$\prod_{i=1}^{n} d_{ai}$	$\prod_{i=1}^{m} d_{bi}$
Array element	$A(s_{a1}, \ldots, s_{an})$	$B(s_{b1}, \ldots, s_{bm})$
Subscript value	$1 + \sum_{i=1}^{n} \left((s_{ai} - l_{ai}) \prod_{j=1}^{i-1} d_{aj} \right)$	$1 + \sum_{i=1}^{m} \left((s_{bi} - l_{bi}) \prod_{j=1}^{i-1} d_{bj} \right)$

because of array reshaping (the number and size of dimensions in an actual argument array declaration are different from those in an associated formal argument array declaration) and other dubious practices such as actual and formal arguments of different types. We use the notations in Table I to represent the relationship between an actual array `A` and a formal array `B`. The size of an array is equal to the number of elements in the array. As Fortran language allocates arrays in column-major order, the subscript value of an array reference does not involve the last upper bound. Note that $\prod_{j=1}^{0} d_j = 1$.

Definition 6.1.1. A program respects the association rules of formal and actual arguments if the following conditions are satisfied:

(1) If the whole array A is passed as an actual argument to the formal array argument B, then

$$e_b . \prod_{i=1}^{m} d_{bi} \leq e_a . \prod_{i=1}^{n} d_{ai}. \tag{9}$$

(2) If the array element $A(s_{a1}, \ldots, s_{an})$ is passed as an actual argument to the formal array argument B, then

$$e_b . \prod_{i=1}^{m} d_{bi} \leq e_a . \left(\prod_{i=1}^{n} d_{ai} + 1 - \left(1 + \sum_{i=1}^{n} \left((s_{ai} - l_{ai}) \prod_{j=1}^{i-1} d_{aj} \right) \right) \right). \tag{10}$$

THEOREM 6.1.2. *If there exists* $k \in \mathbb{N}, 1 \leq k \leq min(n, m)$ *such that* $\forall j = 1, k : d_{aj} = d_{bj}$, *Equation (9) is equivalent to*

$$e_b . \prod_{i=k+1}^{m} d_{bi} \leq e_a . \prod_{i=k+1}^{n} d_{ai}.$$

Moreover, if A is passed as an array element whose first k subscripts are equal to their corresponding lower bounds: $\forall j = 1, k : s_{aj} = l_{aj}$, *Equation (10) is equivalent to*

$$e_b . \prod_{i=k+1}^{m} d_{bi} \leq e_a . \left(\prod_{i=k+1}^{n} d_{ai} - \sum_{i=k+1}^{n} \left((s_{ai} - l_{ai}) \prod_{j=k+1}^{i-1} d_{aj} \right) \right).$$

PROOF. The first equivalence is obtained by dividing both sides by a common term. The second one is proven. If $\exists k \in \mathbb{N}, 1 \leq k \leq min(n,m)$ such that $\forall j = 1,k : d_{aj} = d_{bj}$ and $s_{aj} = l_{aj}$, Equation (10) is equivalent to

$$e_b.\prod_{i=1}^{m} d_{bi} \leq e_a.\left(\prod_{i=1}^{n} d_{ai} - \sum_{i=1}^{k}\left((s_{ai}-l_{ai})\prod_{j=1}^{i-1} d_{aj}\right) - \sum_{i=k+1}^{n}\left((s_{ai}-l_{ai})\prod_{j=1}^{i-1} d_{aj}\right)\right),$$

$$e_b.\prod_{i=1}^{m} d_{bi} \leq e_a.\left(\prod_{i=1}^{n} d_{ai} - \sum_{i=k+1}^{n}\left((s_{ai}-l_{ai})\prod_{j=1}^{i-1} d_{aj}\right)\right) (\text{because } s_{ai} = l_{ai}, i = 1,k),$$

$$e_b.\prod_{i=1}^{k} d_{bi}.\prod_{i=k+1}^{m} d_{bi} \leq e_a.\left(\prod_{i=1}^{k} d_{ai}.\prod_{i=k+1}^{n} d_{ai} - \sum_{i=k+1}^{n}\left((s_{ai}-l_{ai})\prod_{j=1}^{k} d_{aj}.\prod_{j=k+1}^{i-1} d_{aj}\right)\right),$$

$$e_b.\prod_{i=k+1}^{m} d_{bi} \leq e_a.\left(\prod_{i=k+1}^{n} d_{ai} - \sum_{i=k+1}^{n}\left((s_{ai}-l_{ai})\prod_{j=k+1}^{i-1} d_{aj}\right)\right) (\text{as } d_{ai} = d_{bi}, i = 1,k).$$

□

6.2 Actual/Formal Array Size Checking Algorithm

Our analysis traverses the call graph in the invocation order. By using Theorem 6.1.2 and the notation in Table I, the actual/formal array size checking is given by Algorithm 6.2.1, which consists of two steps: *array size check generation* and *redundant code elimination*, using preconditions.

ALGORITHM 6.2.1.

```
procedure Actual_Formal_Array_Size_Checking(p)
    p : current procedure
begin
    for each call site c of p
        q = corresponding_callee(c)
        for each actual array argument a of c
            b = corresponding_formal_parameter(a, q)
            if b is an array variable then
                k := number_of_equal_dimensions(a, b, c, p, q)
                s2 := size_of_formal_array(b, k, q)
                s2' := translate_to_caller_frame(s2, p, c, q)
                s1 := size_subscript_value_of_actual_array(a, k, c, q)
                e := ea.s1 < eb.s2'
                insert "IF (e) STOP message" before c
            endif
        endfor
    endfor
    compute preconditions for each statement of instrumented p
    for each inserted test statement t with condition e of instrumented p
        P = precondition(t)
        sc1 = P ∩ {e}
        sc2 = P ∩ {¬e}
        switch
            case infeasible(sc1) :
                no size violation, remove t from p
```

```
            case infeasible(sc2) :
                size violation found at compile-time
            default: /*undecidable, keep the dynamic check*/
        endswitch
    endfor
end
```

For each formal array parameter at each call site, we generate a test to check whether the size of the formal array exceeds that of the actual array. 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 facilitates the computation of array sizes and subscript value expressions. 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 16, by knowing `N==10`, we only have to check `10<M` instead of introducing the nonlinear expression `10*10<N*M`. Each bound check is accompanied with a stop message and, if a bound violation is detected, the message tells the user which call site creates the violation in which array.

Information about global variables and calling contexts, such as the relationship between actual and formal arguments, is used to improve the translation process and simplify the inequality characterizing the bound violation. The size of the formal array can always be translated to the frame of the caller because the dimension bound expressions are integer constant expressions or contain only formal parameters. By using preconditions, a test condition is checked feasible or not. If it is false with respect to the precondition, the test is removed from the program. If it is true, a violation is detected at compile-time. Otherwise, it is preserved in the program.

7. EXPERIMENTAL RESULTS

We used the SPEC95 CFP benchmark, which contains 10 applications written in Fortran 77. 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 number of subscripted references to arrays. The codes are instrumented and then executed using the standard input data to compute the number of dynamic bound checks. Table II summarizes relevant information for each benchmark in SPEC95 CFP.

Note that three of them (*turb3d, apsi*, and *fpppp*) do not meet the Fortran standard for array declaration and reference: they have *pointer-like* formal array declarations `REAL A(1)`, although array references in the corresponding procedures are outside the defined extent of the array. In addition, assumed-size array declarations `REAL A(*)` also exist in these three benchmarks and four other benchmarks. We added proper bounds to the declarations in *turb3d, apsi*, and *fpppp* by applying *array resizing* [Ancourt and Nguyen 2001] to avoid premature aborts due to bound violations. This transformation uses the argument association rules (Section 6.1) to infer new last upper bounds for assumed-size arrays. Except for the spurious violations caused by the pointer-like

Table II. SPEC95 CFP: Numbers of Lines, Subroutines, Static Checks (or Compile-Time Checks) and Dynamic Checks (or Run-Time Checks)

Program	Lines	Subroutines	Static Checks	Dynamic Checks
tomcatv	190	1	304	49330000000
swim	429	6	772	69910000000
su2cor	2332	35	4460	48100000000
hydro2d	4292	42	2016	65300000000
mgrid	484	12	1162	176080000000
applu	3868	16	9562	115620000000
turb3d	2101	23	1852	60810000000
apsi	7361	96	7172	39360000000
fpppp	2784	38	2894	29280000000
wave5	7764	105	10546	34040000000

Table III. SPEC95 CFP: Percentage of Removed Compile-Time and Run-Time Checks

	Elimination of Redundant Tests				Insertion of Unavoidable Tests			
	Intra		Inter		Intra		Inter	
Benchmark	Com.	Run	Com.	Run	Com.	Run	Com.	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

declarations, there are no out-of-bound errors with the standard data input of SPEC95 CFP.

7.1 Array Bound Checking—Removed Checks

Table III shows the percentages of bound checks removed by the two approaches for array bound checking: elimination of redundant tests and insertion of unavoidable tests. For both approaches, 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 do not have much speedup. Run-time checks are more interesting because they have direct effects on execution time. So to compare between the intraprocedural and interprocedural analysis options, and between the elimination of redundant tests and insertion of unavoidable tests approaches, we only used run-time checks results.

With either approach, interprocedural analysis is not an improvement if the intraprocedural one has already done a good job (*tomcatv, mgrid, applu*).

Table IV. Debugging Information Provided by Different Compilers

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

However, with *turb3d*, the percentage went up from 56.18% to 65% with the elimination approach and from 76.78% to 85.57% with the insertion approach. Improvements were also observed for *su2cor*, *swim*, *hydro2d*, and *wave5*. For *tomcatv*, we statically proved that there is no array bound violation, which is an interesting result for verification purposes.

Comparing the two approaches, we see that the insertion one worked uniformly better. We had almost no gain for *tomcatv, mgrid*, and *apsi*, but there were very big gaps between the 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 varied for different benchmarks, approaches, and options. It was not very high for *turb3d* and *wave5*, because they contain many nonlinear expressions and indirections in array references that are not handled accurately enough by PIPS.

7.2 Array Bound Checking—Debugging Information

The amount of information given when a bound violation occurred differed among compilers. This information is shown in Table IV by experiments with three original benchmarks violating the standard for array references: *turb3d*, *apsi*, and *fpppp*. The experiments were 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 the SGI F77 and GNU G77 compilers and we had to leave them out.

The SUN compile-C option provided the lines of code, the arrays, and the dimensions associated with the violations. The SGI-C option did 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 spotted errors in programs with the *Trace/BPT trap(coredump)* message. Meanwhile, our first array bound checker, elimination of redundant tests, provided full information about the location of the violations, and the second one, insertion of unavoidable tests, gave information about the array and the dimension whose bounds were violated. This information is very important for the debugging process. Other experiments with large-scale industrial codes have shown that the additional information such as the module, the lower or upper bound, and the values of variables given by our first approach is very useful for tracing the origin of the error. In addition, to detect as many errors as possible in one pass, the `STOP` messages can be replaced by `PRINT` messages. This way, the repeated compilations and executions are avoided at the

Table V. SPEC95 CFP: Speed (Spd; Lines per Second) and Compilation Time (in Minutes and Seconds); Ultra SPARC IIi 360 MHz; Optimized Code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

	Elimination				Insertion				SUN F77	
Bench	Spd	PIPS	SUN	Total	Spd	PIPS	SUN	Total	w/o C	w. 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	3:03	1:25	6:28	8.9	12:04	1:50	13:54	2:04	6:20

Note: w/o C = without C; *w*. c = with C.

expense of some log postprocessing to eliminate multiple occurrences of one faulty reference.

7.3 Array Bound Checking—Compilation Times

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 SPEC95 CFP are shown in columns 2 and 6 of Table V. The speeds were 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, were 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, with an Ultra SPARC IIi 360 MHz. 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 (in minutes and seconds) for the insertion of unavoidable tests approach were longer, especially for *mgrid, apsi, fpppp*, and *wave5*. That was 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 shown in Table III, the percentage of removed checks of insertion of unavoidable tests is high enough to pay for this tradeoff.

Since PIPS is a 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 the SUN Workshop F77 5.0 compiler for PIPS generated codes. The original codes of SPEC95 CFP were also compiled with and without the array range checking option of SUN. The experimental results showed shorter times for the two implementations of PIPS than for SUN (see the columns Total for elimination of redundant tests and for insertion of unavoidable tests and the column With C of SUN F77).

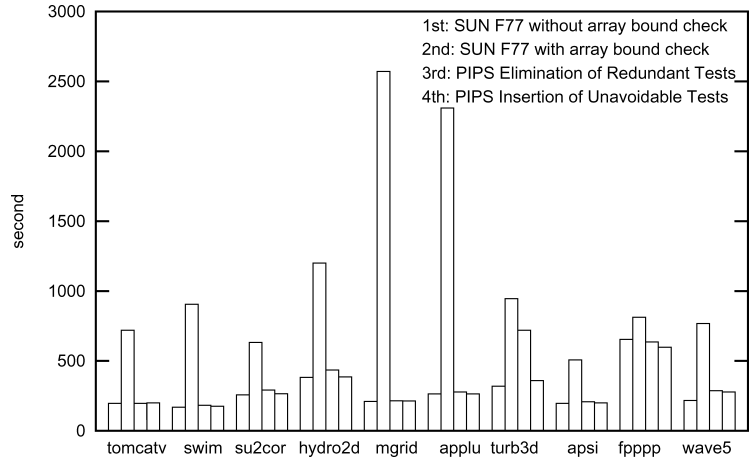

Fig. 17. Execution time: SUN F77 and PIPS; SUN Workshop F77 5.0; Ultra SPARC 360 MHz; optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch).

7.4 Array Bound Checking—Execution Times

The execution times of SPEC95 CFP were measured on different platforms to see the relationship between the percentage of eliminated checks and the slowdown. This set of experiments is reported with the optimizing options turned on, using the SPEC95 CFP measurement guidelines. The code generated by PIPS with its own range checking using the interprocedural option for transformers and preconditions was compiled by other compilers (SUN, SGI, and IBM) to generate executable files. For IBM, because an internal compiler error occurred when compiling the Fortran code with options -O5 and -C together, we used -O3. In addition, an input/output error occurred 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 Figures 17, 18, and 19.

We can see the overheads of range checking in *mgrid* and *applu* for SUN (Figure 17), *mgrid* and *swim* for SGI (Figure 18), and *tomcatv* and *turb3d* for IBM (Figure 19). 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 II, column 5. As the range checking of the IBM compiler was already optimized, the PIPS versions worked better than IBM in general but worse for the *turb3d* benchmark. The reason is that analyses of nonlinear expressions, which occur frequently in this benchmark, are not implemented yet in PIPS.

Comparing the execution time of the PIPS codes with that of other bound checked codes, on average, the PIPS elimination of redundant tests was 3.94 times faster than SUN, 1.88 times faster than SGI, but only 1.03 times faster than IBM, which does not provide error information. The PIPS insertion of unavoidable tests was 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 were slightly longer than that of the unsafe programs

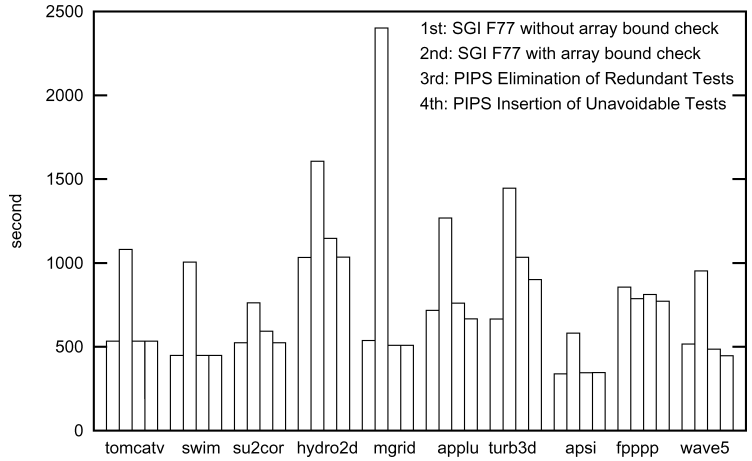

Fig. 18. Execution time: SGI F90 and PIPS; SGI MIPSpro F90 7.3; O2 R5000 195 MHz, IRIX 6.3; Optimized code (f90 -Ofast=ip32_5k).

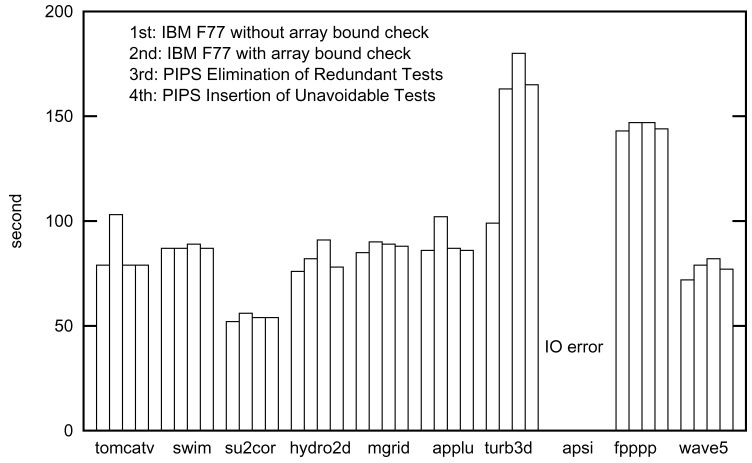

Fig. 19. Execution time: IBM F77 and PIPS; IBM XL F77 7.1; RS/6000 44P-270 375 MHz 4 CPU, AIX 4.3; optimized code (f77 -O3 -lmass).

without bound checks. On average, these times for the PIPS elimination of redundant tests were about 19.29% longer for SUN, 7.05% longer for SGI, and 16.59% longer for IBM. For the PIPS insertion of unavoidable tests, they were about 5.33% longer for SUN, 0.61% longer for SGI, and 10.62% longer for IBM.

7.5 Actual/Formal Array Size Checking

The numbers of compile-time and run-time checks added, as well as the total compilation time (in minutes and seconds) and the slowdown caused by the actual/formal array size checking for the SPEC95 CFP benchmarks are shown

Table VI. SPEC95 CFP: Number of Added Compile-Time and Run-Time Checks, Total Compilation Time (in Minutes and Seconds) and Execution Slowdown

Benchmark	Compile Checks	Run Checks	Compilation	Slowdown
tomcatv	0	0	0:01	0.00%
swim	0	0	0:03	0.00%
su2cor	0	0	0:59	0.00%
hydro2d	0	0	0:21	0.00%
mgrid	24	40151	0:05	0.15%
applu	0	0	0:17	0.00%
turb3d	29	281417	0:42	2.12%
apsi	6	240127	1:25	1.67%
fpppp	2	727917	2:29	0.79%
wave5	34	Bound violation	2:36	Bound violation

in Table VI. Although assumed-size arrays are supposed not to exceed the size of the corresponding actual arrays, they still raised problems for the actual/formal array size checking. If the formal array is declared correctly but the actual one has an assumed-size, we cannot compare them. So we have to apply array resizing to seven benchmarks in SPEC95 CFP.

By using static analyses, our checking has proved that there was no array size violation in 5 out of the 10 benchmarks. Other bound checks were added before some procedure calls in the five remaining benchmarks. We cannot compare the effectiveness of our approach to some compilers (Salford compilers) that do this checking, but the cost here was small enough. The maximum slowdown was only 2.12% for *turb3d*.

A bound violation was detected in *wave5*, an electromagnetic particle simulation program. Figure 20 contains the piece of code that caused a bound violation for array `TMP` when passing it as an 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(NX2,78885/NX2)` in subroutine `SOLV2Y`. This array was declared as `TMP(NX2,*)` in the original code, before the array resizing phase. With the argument association rules, we inferred its new declaration, based on the size of the actual array `BX(NC1)` in subroutine `FIELD`. By using binding information between formal and actual arguments and preconditions, we had `SLV2XY:NXD==SOLV2Y:NX2==FIELD:NX2==FIELD:NX+2` and `SLV2XY:NY==SOLV2Y:NY2-2==FIELD:NY2-2==FIELD:NY`. So the size of array `TMP` in `SLV2XY` was translated into the frame of `SOLV2Y`, from `NXD*NY*2` to `NX2*(NY2-2)*2`. The array size check in the frame of `SOLV2Y` was `78885.LT.NX2*(NY2 -2)*2`. When executing the instrumented code with its standard input data where the grid size `NX==1250` and `NY==60`, we had that `78885<1252*60*2` was true, so there was a bound violation here.

Because `TMP`, associated to `BX`, is accessed outside its 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 array size violations are not checked. So if actual/formal array size checking is omitted, this kind of violation makes other analyses such as alias analysis impossible and code maintenance difficult.

```
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),TMP(NX2,78885/NX2),BY(4)
IF (78885.LT.NX2*(NY2-2)*2) STOP "Size violation: array TMP"
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 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. 20. Actual/formal array size mismatch (excerpt from *wave5*).

8. CONCLUSION

To improve the efficiency and effectiveness of array bound checking, we designed and experimented two algorithms for array bound checking and one algorithm for actual/formal array size checking. The number of removed/added bound checks, the information about violations, and the compilation and the execution times were measured for the SPEC95 CFP benchmarks with three different compilers and with our experimental implementations. The results show that our techniques improved significantly speed and/or information.

The study of Richardson and Ganapathi [1989] suggested 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. On average, the slowdown is divided by 2. Once again, it is still a question of a tradeoff between speed and accuracy, but in the domain of verification, proving or checking the correctness of code is the most important criterion.

8.1 Comparison Between Two Array Bound Checking Approaches

The experimental results show the effectiveness and the limited optimization cost of our two 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 provides very accurate information for the debugging. 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 removed dynamic 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 removed dynamic bound checks was 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.

8.2 Comparison with Commercial Compilers

Within the SUN environment, we measured shorter compilation times using our two source-to-source array bound checkers followed by F77 than using F77 alone with its -C option. The average compilation time speedups were 3.03 for elimination of redundant tests and 1.1 for insertion of unavoidable tests. The PIPS elimination of redundant tests had shorter compilation and execution times than the SUN compiler while preserving the same diagnostic capabilities.

At run-time, the slowdowns measured for SUN and SGI compilers were large enough to make improvement easy. Unexpectedly, we obtained mostly speedups by adding array bound checks of the insertion of unavoidable tests version (see Figure 18, Columns 1 and 4 for each benchmark). This was not the case with IBM XLF compiler 7.1.0.0, which nevertheless was not uniformly efficient and which broke down with an internal error when combining options -C and -O5. However, our execution times were in the same range as IBM's, seven times out of nine slightly better. Furthermore, as the SUN compiler does, 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. As in code hoisting methods, the 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. This shows that a simple time comparison 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 inferred from Figures 17 and 19 that the SUN compiler is not as efficient as IBM's.

8.3 Comparison with Previous Research Experiments

Among related work, the article "Elimination of Redundant Array Subscript Range Checks" [Kolte and Wolfe 1995] contains the most experimental results. They implemented in their Nascent research compiler different techniques for range checking optimizations based on different check placement schemes. Their experimental results showed that simple optimizations such as preheader insertion with loop-limit-substitution of linear checks greatly reduce the number of checks removed. The set of experiments was 10 Fortran programs from the Perfect, Riceps, and Mendez benchmarks. However, we could not fully compare our results with theirs because the authors did 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 depend on architectures and compilers. For instance, in our experiments 0.01% of dynamic checks slowed down the execution of *swim* by 7.65% (see Table III and Figure 17). So, to evaluate array bound checking optimization, it would be necessary to compare the execution times of generated codes. However, that information was missing in Kolte and Wolfe [1995]. Furthermore, no information about the origin of violation was preserved in their approach. Problems related to pointer-like and assumed-size arrays in Perfect Club and Riceps benchmarks were not mentioned. Out-of-bound errors occurred in four benchmarks: *mdg, spc77, trfd* from PerfectClub, and *linpackd* from Riceps. These errors were caused by violating standard declaration of arrays in Fortran and there was no possibility to obtain comparable results for these programs. Kolte and Wolfe's best figures for removed checks were in the very same range as ours, and it is very interesting to see that specific techniques did not work better than generic techniques.

8.4 Code Quality

Result analysis showed the importance of code quality. Proper array declarations are needed to avoid out-of-bound errors caused by standard violation. Assumed-size array declarations prevent a complete range checking, so we used array resizing to overcome this problem. Furthermore, some benchmarks benefit from more sophisticated techniques or modifications such as cloning, parameter checking, scalarization for dealing with indirections, scalarization for loop bounds, nonunit loop increments, etc. For example, we improved the percentage of removed checks in the elimination of redundant tests from 94.14% (see Table III) to 99.50% for *hydro2d* by cloning the subroutine `ADLEN`, which has two totally different behaviors for two parameter values: "half" and "full" steps. The execution time on SUN was 10% shorter. The elimination percentage went up to 100% from 97.09% for *applu* by adding one `STOP` statement after the parameter checking that was 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 was then measured on SUN. Poor code quality can make static analysis insufficient; run-time checks remain and run-time failures cannot be eliminated because unintended behaviors must be taken into account.

Another conclusion is retrieved from experience with industrial codes that have out-of-bound violations. Programmers sometimes initialize the whole common block by using only the first array, which causes upper bound violations. To cope with this style of programming, we developed *area bound checking* [Nguyen 2002], which compares the array reference to the size of the corresponding common block, not to the upper bound in the array declaration. So in fact, a lower bound violation is more likely to be a real bug than an upper one. It should be noted that our implementations scale up to 200,000 lines of code.

8.5 Improvements and Perspectives

Some analyses for nonlinear expressions or indirection arrays that have a direct impact on array range checking were not implemented in PIPS, so we did not

obtain satisfying 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 Midkiff et al. [1998] and 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.

Actual/formal array size checking is as important as normal array bound checking 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.

Our implementations 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. Fewer than 1700 additional lines of C code are sufficient to implement both kinds of 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 C, Ada, Java, etc., but they should be extended to cope with other language characters such as recursive calls and data structures, pointer arithmetics, dynamic memory allocation, and heritage. Advanced pointer analysis must be taken into account in order to make the array region precise, and then the insertion of unavoidable test approach efficient. We believe that encouraging results can be achieved at least with the elimination of redundant tests approach for well-behaved C programs, since information can be deduced to remove unnecessary tests. Additional analysis such as array resizing can be used to infer array descriptors. All these problems are being currently studied. The PIPS software and documentation as well as the array bound checking implementations are available online at `http://www.cri.ensmp.fr/pips`.

ACKNOWLEDGMENTS

We would like to thank B. Creusillet and F. Coelho for implementing some key algorithms in PIPS, as well as C. Ancourt, P. Jouvelot, and R. Keryell for their collaborations. We also wish to give special thanks to S. Algarotti, C. Mongenet, and R. David for letting us use their IBM and SGI machines. Last, we would especially like to thank the anonymous reviewers whose comments have been very helpful in clarifying our contributions and to making the article more self-contained.

REFERENCES

AGGARWAL, A. AND RANDALL, K. H. 2001. Related field analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 214–220.

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

ALLEN, F., BURKE, M., CHARLES, P., CYTRON, R., AND FERRANTE, J. 1988. An overview of the PTRAN analysis system for multiprocessing. *J. Parall. Distrib. Comput. 5*, 617–640.

AMI, T. L., REPS, T., SAGIV, M., AND WILHELM, R. 2000. Putting static analysis to work for verification: A case study. In *Proceedings of the International Symposium on Software Testing and Analysis*. 26–38.

ANCOURT, C. AND IRIGOIN, F. 1991. Scanning polyhedra with DO loops. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 39–50.

ANCOURT, C. AND NGUYEN, T. V. N. 2001. Array resizing for code debugging, maintenance and reuse. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 32–37.

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

ASURU, J. M. 1992. Optimization of array subscript range checks. *ACM Lett. Programm. Lang. Syst. 1*, 2 (June), 109–118.

AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 290–301.

BODIK, R., GUPTA, R., AND SARKAR, V. 2000. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 321–333.

CALLAHAN, D. AND KENNEDY, K. 1988. Analysis of interprocedural side effects in a parallel programming environment. *J. Parall. Distrib. Comput. 5*, 517–550.

CHIN, W.-N. AND GOH, E.-K. 1995. A reexamination of 'Optimization of array subscript range checks.' *ACM Trans. Programm. Lang. Syst. 17*, 2 (March), 217–227.

COUSOT, P. AND COUSOT, R. 1976. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. 106–130.

COUSOT, P. AND COUSOT, R. 1977. Static determination of dynamic properties of recursive programs. In *Proceedings of the IFIP Conference on Formal Description of Programming Concepts*. 237–277.

COUSOT, P. AND HALBWACHS, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 84–96.

CREUSILLET, B. AND IRIGOIN, F. 1995. Interprocedural array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1033. Springer-Verlag, Berlin, Germany, 46–60.

CREUSILLET, B. AND IRIGOIN, F. 1996. Exact vs. approximate array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1239. Springer-Verlag, Berlin, Germany, 86–100.

DELZANNO, G., JUNG, G., AND PODELSKI, A. 2000. Static analysis of array bounds as infinite-state model checking. Unpublished article.

DOR, N., RODEH, M., AND SAGIV, M. 2001. Cleanness checking of string manipulation in C programs via integer analysis. In *the Static Analysis*. Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, Berlin, Germany, 194–212.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1993. A practical data flow framework for array reference analysis and its application in optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 68–77.

DUJMOVIC, J. J. AND DUJMOVIC, I. 1998. Evolution and evaluation of SPEC benchmarks. *ACM SIGMETRICS 26*, 3, 2–9.

EVANS, D. AND LAROCHELLE, D. 2002. Improving security using extensible lightweight static analysis. *IEEE Softw. 19*, 1 (Jan./Feb.), 42–51.

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Int. J. Parall. Programm. 20*, 1, 23–53.

GU, J. AND LI, Z. 2000. Efficient interprocedural array data-flow analysis for automatic program parallelization. *IEEE Trans. Softw. Eng. 26*, 3 (March), 244–261.

GUPTA, M., MUKHOPADHYAY, S., AND SINHA, N. 2000. Automatic parallelization of recursive procedures. *Int. J. Parall. Programm. 28*, 6, 537–562.

GUPTA, R. 1990. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 272–282.

GUPTA, R. 1993. Optimizing array bound checks using flow analysis. *ACM Lett. Programm. Lang. Syst. 2*, 1–4 (March–Dec.), 135–150.

HALL, M., MURPHY, B., AMARASINGHE, S., LIAO, S.-W., AND LAM, M. 1995. Interprocedural analysis for parallelization. In *International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1033. Springer-Verlag, Berlin, Germany, 61–80.

HARRISON, W. H. 1977. Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng. SE-3*, 3 (May), 243–250.

HASTING, R. AND JOYCE, B. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*. 125–136.

HIND, M., BURKE, M., CARINI, P., AND MIDKIFF, S. 1994. An empirical study of precise interprocedural array analysis. *Sci. Programm. 3*, 3, 255–271.

HOEFLINGER, J. P., PAEK, Y., AND YI, K. 2001. Unified interprocedural parallelism detection. *Int. J. Parall. Programm. 29*, 2, 185–215.

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

JONES, R. W. M. AND KELLY, P. H. J. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging*.

KARR, M. 1976. Affine relationship among variables of a program. *Acta Informatica*, *6*, 133–151.

KOLTE, P. AND WOLFE, M. 1995. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 270–278.

KOWSHIK, S., DHURJATI, D., AND ADVE, V. 2002. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*.

LESERVOT, A. 1996. Analyses interprocédurales du flot des données. Ph.D. dissertation. Université Paris VI, Paris, France.

LIN, Y. AND PADUA, D. 1999. Demand-driven interprocedural array property analysis. In *International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1863. Springer-Verlag, Berlin, Germany, 303–317.

MANJUNATHAIAH, M. AND NICOLE, D. A. 1997. Precise analysis of array usage in scientific programs. *Sci. Programm. 6*, 229–242.

MARKSTEIN, V., COCKE, J., AND MARKSTEIN, P. 1982. Optimization of range checking. *ACM SIGPLAN Not. 17*, 6 (June), 114–119.

MAYDAN, D. E., AMARASINGHE, S. P., AND LAM, M. S. 1993. Array data-flow analysis and its use in array privatization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 2–15.

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

MOREIRA, J. E., MIDKIFF, S. P., AND GUPTA, M. 2000. From flop to megaflops: JAVA for technical computing. *ACM Trans. Programm. Lang. Syst. 22*, 2 (March), 265–295.

MOREIRA, J. E., MIDKIFF, S. P., GUPTA, M., ARTIGAS, P. V., WU, P., AND ALMASI, G. 2001. The NINJA project. *Commun. ACM 44*, 10 (Oct.), 102–109.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA.

NECULA, G. C. AND LEE, P. 1998. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 333–344.

NGUYEN, T. N., GU, J., AND LI, Z. 1995. An interprocedural parallelizing compiler and its support for memory hierarchy research. In *International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1033. Springer-Verlag, Berlin, Germany, 96–110.

NGUYEN, T. V. N. 2002. Efficient and effective software verifications for scientific applications using static analyses and code instrumentation. Ph.D. dissertation. Ecole des Mines de Paris, Paris, France.

PAEK, Y., HOEFLINGER, J., AND PADUA, D. 2002. Efficient and precise array access analysis. *ACM Trans. Programm. Lang. Syst. 24*, 1, 65–109.

PATIL, H. AND FISCHER, C. N. 1997. Low-cost, concurrent checking of pointer and array accesses in C programs. *Soft.—Pract. Exper. 27*, 1, 87–110.

PUGH, W. 1992. A practical algorithm for exact array dependence analysis. *Commun. ACM 35*, 8 (Aug.), 102–114.

QIAN, F., HANDREN, L., AND VERBRUGGE, C. 2002. A comprehensive approach to array bounds check elimination for Java. In *Compiler Construction*. Lecture Notes in Computer Science, vol. 2304. Springer-Verlag, Berlin, Germany, 325–341.

RICHARDSON, S. AND GANAPATHI, M. 1989. Interprocedural optimization: Experimental results. *Soft.—Pract. Exper. 19*, 2 (Feb.), 149–169.

RUGINA, R. AND RINARD, M. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 182–195.

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

SCHWARZ, B., KIRCHGASSNER, W., AND LANDWEHR, R. 1988. An optimizer for Ada—design, experiences and results. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 175–184.

SPEZIALETTI, M. AND GUPTA, R. 1995. Loop monotonic statements. *IEEE Trans. Soft. Eng. 21*, 6 (June), 497–505.

STEFFEN, J. L. 1992. Adding run-time checking to the portable C compiler. *Soft.—Pract. Exper. 22*, 4 (April), 305–316.

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

TRIOLET, R., FEAUTRIER, P., AND IRIGOIN, F. 1986. Automatic parallelization of Fortran programs in the presence of procedure calls. In *Proceedings of the European Symposium on Programming*.

TU, P. AND PADUA, D. A. 1995. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the International Conference on Supercomputing*. 414–423.

WAGNER, D. A. 2000. Static analysis and computer security: New techniques for software assurance. Ph.D. dissertation. Computer Science, University of California, Berkeley, Berkeley, CA.

WELSH, J. 1978. Economic range checks in Pascal. *Soft.—Pract. Exper. 8*, 85–97.

Received May 2003; revised October 2003; accepted February 2004