Optimization of array bound checking

Nguyen Thi Viet Nga
CRI - ENSMP
77305 Fontainebleau Cedex, France
nguyen@cri.ensmp.fr

RAPPORT INTERNE E/235/CRI
(rapport de première année)
19 Octobre 2000

Abstract

Optimization of array bound checking at compile-time is performed in order to reduce the overhead of the run-time checking. There are many researches in this field but the results seem not very highly appreciated, even with the commercial compilers. This paper describes two techniques for array bound checking implemented in our research compiler PIPS. It also gives experimental results together with the comparison of different approaches.

Keywords: array reference, array bound checking, range checking, subscript out of range, bound violation.

1 Introduction

Array bound checking refers to determining whether all array references are within their declared range in all of its uses in a program. These array bound checks may be analysed intraprocedurally or interprocedurally, depending on the accuracy need. Such checking is desirable for any program, regardless of the language it is written in, since bound violations are among the most common programming errors. Subscripting arrays beyond their declared sizes may result in unexpected results, including segmentation faults.

For the safety of execution, some language specifications such as Java require that a program only be allowed to access elements of an array that are part of the defined extent of the array.

But on the other hand, bound checking can be very expensive because every array access must be accompanied by two bound checks per dimension to determine the validity of the access. It increases the size of the executable file and degrades execution performance.

Many other languages solve this problem, for example Pascal and Fortran, by providing the user with a compile-time option to enable or disable the checking. The purpose of this option is to allow the user to enable the checking for the developement and debugging runs of the program, and then, once all the defects have been found and fixed, to turn it off for the production version.

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

Our experiments are done with the SPEC CFP95 benchmark which contains 10 applications written in FORTRAN. A brief description of each benchmark follows.

101.tomcatv is a vectorized mesh generation program with 190 lines of code.

102.swim solves the system of shallow water equations with 429 lines of codes and 6 subroutines.

103.su2cor computes the masses of elementary particles in the framework of the Quark-Gluon theory, from the area of quantum physics with 2332 lines of code and 35 subroutines.

104.hydro2d solves the Hydrodynamical Navier Stokes equations to compute galactical jets, from the area of astrophysics with 4292 lines of code and 42 subroutines.

107.mgrid is a multi-grid solver in 3D potential field with 484 lines of code and 12 subroutines.

110.applu solves the parabolic/elliptic partial differential equations with 3868 lines of code and 16 subroutines.

125.turb3d simulates isotropic, homogeneous turbulence in a cube with 2101 lines of code and 23 subroutines.

141.apsi solves problems regarding temperature, wind, velocity and distribution of pollutants with 7361 lines of code and 96 subroutines.

145.fpppp measures performance on one style of computation which occurs in the Gaussian series of programs, from quantum chemistry with 2784 lines of code and 38 subroutines.

146.wave5 solves Maxwell equations and particle equations of motion on a cartesian mesh with 7764 lines of code and 105 subroutines.

Program	Lines	Subs	Static	Dynamic
101.tomcatv	190	1	304	$4933\text{x}10^7$
102.swim	429	6	772	$6991\text{x}10^7$
103.su2cor	2332	35	4460	$4810\text{x}10^7$
104.hydro2d	4292	42	2016	$6530\text{x}10^7$
107.mgrid	484	12	1162	$17608\text{x}10^7$
110.applu	3868	16	9562	$11562\text{x}10^7$
125.turb3d	2101	23	1852	$6081\text{x}10^7$
141.apsi	7361	96	7172	$3936\text{x}10^7$
145.fpppp	2784	38	2894	$2928\text{x}10^7$
146.wave5	7764	105	10546	$3404\text{x}10^7$

Table 1: Statistics of SPEC CFP95: number of lines, subroutines and number of estimated static and dynamic range checks for the SPEC input data

Table 1 summarizes the statistics of each benchmark in SPEC CFP95. For further informations on SPEC, the reader is referred to *www.spec.org*.

SPEC CFP95 are FORTRAN scientific benchmarks with floating point arithmetic and many of them have been derived from publicly available application programs. Each benchmark contains a large amount of subscripted references to arrays. We use the compiler SUN Workshop f77 version 5.0 with and without array bound checking option. The CPU execution time of each program on an Ultra SPARC 360MHz is presented in Figure 1 and Figure 2. These measurements are realized with no additional compile-time flags (Figure 1) and also with optimization flags (Figure 2). These figures show the large execution overhead of commercial range checkings (-C option) for both unoptimized and optimized codes. On average, they run about three times slower and have much larger generated code and much longer compilation time. Beside this, there are three benchmarks with subscript out of range errors that cause terminating exception. It is due to standard violating declarations of arrays in some subroutines, such as $A(N, 1)$ where the second dimension is replaced by 1. That explains why we tried to improve array bound checking in both developement and production phases.

The rest of the paper is organized as follows. The next section discusses the related work on optimizations of array range checking and explain what the two approaches are, with their advantages and drawbacks. Section 3 presents our first implementation based on elimination of redundant tests. Section 4 presents the second implementation based on insertion of inavoid-

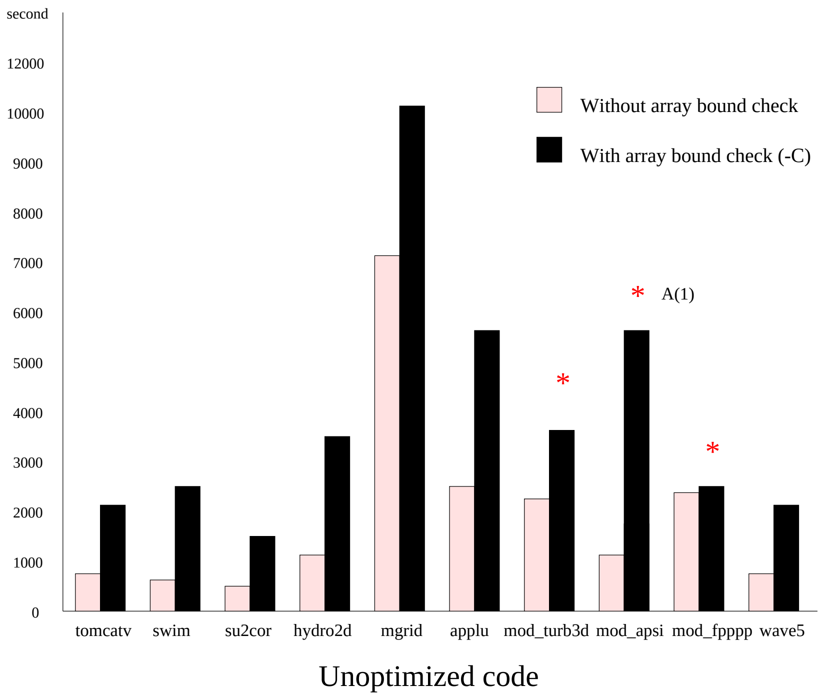

Figure 1: Execution time with and without array bound checking (SUN f77 5.0 Ultra SPARC 360MHz) - Unoptimized code

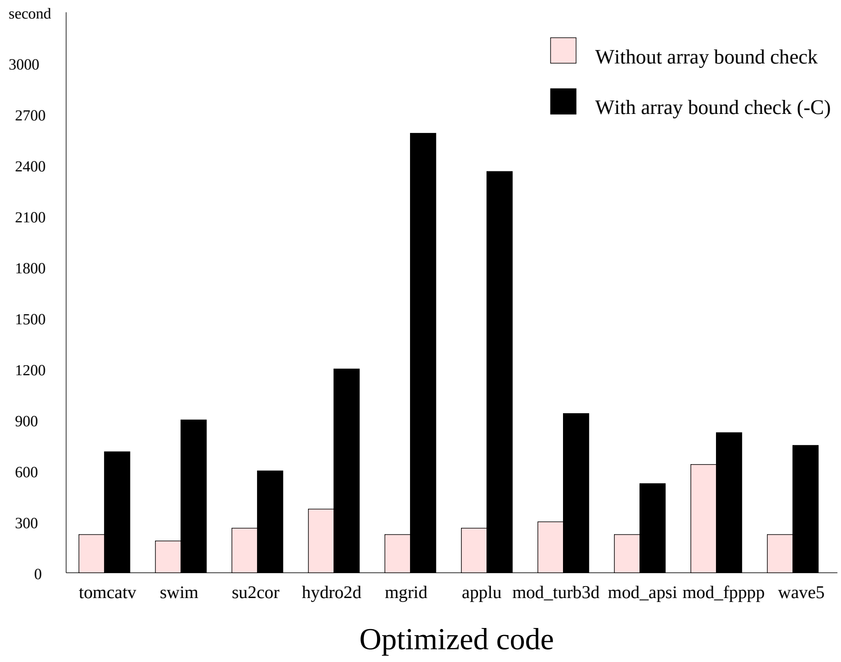

Figure 2: Execution time with and without array bound checking (SUN f77 5.0 Ultra SPARC 360MHz) - Optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

able tests. Experimental results are reported in Section 5. A comparison and a combination of the two approaches and the conclusion for the paper are given in the last section.

2 Related work

A great deal of work has been done in the area of optimizing range checks for arrays.

In 1977, Suzuki and Ishihata [24] implemented a system that inserts logical assertions for the validity of program before array element accesses and then uses theorem proving techniques to verify the absence of array range violations. Such techniques are often expensive and they are restricted to programs written in a structured manner, i.e without goto statement.

Another approach was developped by V. Markstein, Cocke and P. Markstein [15], Asuru [16], Gupta [9, 10], Spezialetti and Gupta [22], Kolte and Wolfe [14]. They propose algorithms to reduce the execution overhead of range checks through 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 [10], a bound check that is identical or subsumed by other bound checks will be suppressed in a local elimination. In a global elimination, an algorithm first modifies bound checks to create additional redundant checks and then carry out the elimination of redundant checks by using the notions of *available* and *very busy* checks. These two algorithms use backward and forward data flow analyses to solve the problems. For the propagation of checks out of loops, Gupta identifies candidates for propagation which include invariants, loops with increment or decrement of one and variables of increasing or decreasing values. Then he uses check hoisting to move checks out of loops. However, the results are not very convincing because of the small size of his examples and because the optimizations have been applied by hand only.

In their article, Kolte and Wolfe [14] rely on a check implication graph where nodes are families of set of range checks in canonical form and edges denote implications between these families. Like [10], they compute also the *available* and *anticipatable* checks by solving the forward and backward data flow problems. To create more redundant checks, there are five schemes to insert checks at safe and profitable program points: *no-insertion, safe-earliest, latest-not-isolated placement, check-strengthening and preheader insertion.* They use partial redundancy elimination to determine the most efficient places to move bound checks to.

Their implementation in the Fortran compiler Nascent [14] is a combination and developement of other techniques in range check optimization with interesting experimental results that show the necessity of range check optimization, the effectiveness and cost of optimization. However, high percentages, even 99.99%, of eliminated tests do not always mean faster execution times. This article lacks a comparision between the execution times of code with optimized bound checks and other codes to show the impact of removed checks. Beside this, there are no mentions of bound violations in PerfectClub and Riceps benchmarks (*mdg, spc77, trfd, linpackd*) which are caused by standard violating declaration of array dimension in Fortran.

The abstract interpretation approach proposed by Cousot and others [3] consider array range checking as an example for the automatical verification of execution properties of programs. They use static data flow analysis information to prove at compile time that an array bound violation cannot occur at runtime and the test for the violation is unnecessary. Different algorithms for propagation and combination of assertions depend on different rules they used. Since the algorithms in the abstract interpretation approach and the program verification approach do not perform any insertion of checks in the program to create more redundant check, they could only take advantage of completely redundant checks. So the run time overhead of partial redundant checks which cannot be evaluated at compile-time still remains.

The implementation of the algorithm Eliminating Array Bound Checks on Demand in article [1] which is based on extended Static Single-Assignment graph representation can remove about 45% of dynamic bound checks for Java programs.

Other articles by Midkiff, Moreira, Snir and Gupta [20, 18] describe another approach optimize array reference checking in Java programs based on code replication. All of the optimizations work by partitioning a loop nest, an iteration space into regions with different access violation characteristics. In unsafe regions, run-time tests are performed, whereas in other regions they are not necessary because all array indices are guaranteed to be within bounds. The optimizations differ on their level of refinement and practicality. These techniques are less complicated than the abstract interpretation approach but still effective. However, they did not use any control-flow analysis to reduce some code replication and the optimizations here are mainly for Java applications.

Although there are many different techniques for array bound checking optimization, we can partition them into main possible approaches. The first approach puts array bound checks at every array reference and removes a check if it is redundant [15, 9, 16, 10, 14]. In the second approach, we only put array bound checks at places where we don't know how to prove that they are useless [24, 6, 20].

The first attempts to reduce the dynamic and static number 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 and then it can be eliminated. Hoisting range checks out of loops is also applied when it is possible. The analyses are simple or sophisticated depending on each technique. The question is: is it worth performing range check optimizations complicated when the hoisting of array bound checks from the most inside loop out may be sufficient?

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

So both approaches have avantages and drawbacks when comparing the number of transformations and the number of analyses. We might even end up with a combination of bottom-up and top-down analyses in order to have a better result. The main goal of our work here is to compare the effectiveness and the optimization cost of two different algorithms described in the following sections. The first one is based on test elimination, and the second one is based on optimized test insertion.

3 Elimination of redundant tests

PIPS, *Paralléliseur Interprocédural de Programmes Scientifiques* [13, 12], is a research project developed at Ecole des Mines de Paris. It consists of several analyses phases such as dependence analyses, transformer, precondition, complexity, array region or program transformations such as loop transformations, constant folding, unreachable code elimination.

Our first implementation of a range check optimizer is based on two-step algorithm that is described as follows:

1. Generate bound checks for every array reference

However if one bound check is verified always true, for example $N-1 <= N$, or exists already in the list of bound checks for the same statement, it doesn't need to be added. As Fortran permits assumed-size array declarators with an unbounded upper bound of the last dimension, a range check is not generated for this case either. Checks are inserted before the current statement with error messages to display if bound violation occurs.

2. Apply Partial Redundancy Elimination

In essence, a *partial redundancy* [19] is a computation that is done more than once on some path through a flow graph. To eliminate partial redundancy for logical expressions such as array bound checks, we can use informations given by two auxiliary analyses in PIPS: *transformers* and *preconditions*.

Transformers abstract the effects of instructions upon the values of integer scalar variables by giving an affine approximation of the relations that exist between their values before and after the execution of a statement or procedure call. They are propagated from bottom to top in the call tree.

Preconditions in PIPS are affine predicates over scalar integer variable values à la Cousot/Halbwachs [3]. These predicates hold just before the execution of the corresponding statement. They do not have to be exact, or even optimal in some sense, but must hold for any possible execution of the program. Preconditions are propagated from the module entry point down to the abstract syntax tree leaves. Transformers, which are computed during a preliminary phase, are applied to preconditions to obtain postconditions, which usually are the precondition of the next statement.

So if some path through the flow graph contains a point at which a bound check has already been computed and it will be computed again at a following point, by using the postcondition of the former point or the precondition of the latter point, we can decide that the bound check is redundant and then remove it.

Beside this, precondition of a statement can determine if a bound test is always true or is always false. If it is alway true, there is certainly bound violation that can be detected statically at compile time. If it is always false, the test should be deleted.

Figure 3 shows a motivating example to illustrate this implementation. The precondition $1 <= I$ after entering the loop permits us to eliminate all lower bound checks for array U and V. The check $I+1.GT.15$ in the first test makes all computations of upper bound checks after it become redundant. So in the final code, we have only one bound check to compute but it is left inside the loop (no hoisting).

In fact, the accuracy of preconditions depends on the analysis sophistication. We can choose different fix-point operators to handle loops, intra- or inter-procedural analysis to have more precise information.

```
C  P() {}
   SUBROUTINE DOLOOP(K)
   REAL U(15)
C  P() {}
   N = 2*K
C  P(N) {2K==N}
   DO 10 I = 1, N, 2
C  P(I,N) {2K==N, 1<=I, I<=2K}
      IF (I+1.LT.1.OR.I+1.GT.15) STOP
C  P(I,N) {2K==N, 1<=I, I<=14, I<=2K}
      U(I+1) = I
C  P(I,N) {2K==N, 1<=I, I<=14, I<=2K}
      IF (I.LT.1.OR.I.GT.15) STOP
C  P(I,N) {2K==N, 1<=I, I<=14, I<=2K}
      U(I) = I+1
C  P(I,N) {2K==N, 1<=I, I<=14, I<=2K}
10 CONTINUE
   END

   SUBROUTINE DOLOOP(K)
   REAL U(15)
   N = 2*K
   DO 10 I = 1, N, 2
      IF (I+1.GT.15) STOP
      U(I+1) = I
      U(I) = I+1
10 CONTINUE
   END
```

Figure 3: Example of Elimination of redundant test

After the partial redundancy elimination transformation, the number of generated tests is greatly reduced. If a bound violation is detected, a stop message will tell that the subscript is out of range in which array, on which dimension and in which subroutine. PIPS translates Fortran programs into instrumented Fortran codes with real bound checks which are then compiled and executed using their standard input data sets. Table 3 shows the percentages of the numbers of static and dynamic bound checks eliminated for SPEC CFP95 benchmarks by this implementation. The average percent-

Program	Static checks	Dynamic checks
101.tomcatv	97.70%	99.99%
102.swim	98.40%	99.99%
103.su2cor	96.60%	97.59%
104.hydro2d	96.01%	99.50%
107.mgrid	94.90%	99.60%
110.applu	99.60%	97.09%
125.turb3d	93.10%	99.99%
141.apsi	95.90%	99.99%
145.fpppp	92.80%	99.99%
146.wave5	94.60%	86.86%

Table 2: Percentage of static and dynamic checks eliminated: Elimination of redundant tests

age of removed static bound checks is 95.96% and that of removed dynamic checks is 98.05%.

We discuss the execution time of the instrumented code and the required compilation time of PIPS to achieve these results in the Experimental results section.

4 Insertion of unavoidable tests

This second implementation is based on an array region analyses phase. An array region, as defined in [5, 12], is a set of array elements described by a convex polyhedron containing affine equalities and inequalities. These constraints link the region parameters that represent the array dimensions to the value of the program integer scalar variables.

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

For instance, the region: $< A(\phi_1, \phi_2) - EXACT - \{\phi_1 == I, \phi_1 == \phi_2\} >$ where the region parameters ϕ_1 and ϕ_2 respectively represent the first and second dimensions of A, corresponds to an array reference $A(I, I)$.

Regions are built bottom-up, intraprocedurally, 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 pieces from different control branches are merged with a meet operator. The approximation of regions is conservatively reserved. Intraprocedural region analysis has to deal with elementary statements like assignments, basic blocks, tests, loops and unstructured statements.

The interprocedural propagation of regions is a bottom-up (or backward) analysis in the program call graph. 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, using the relations between actual and formal parameters, and between the declarations of global variables in both routines.

Implementation in PIPS tackles several non-exclusive cases caused by Fortran language features, such as array reshaping due to different dimension declarations, offset between the first elements of the source and target arrays due to parameter passing, offset due to different COMMON declarations in the caller and the callee, different type between target and source variables.

Our second array bound checker inserts unavoidable tests in a top-down analysis that requires a hierarchical control flow graph, from the largest compound statement node down to elementary statements. At each statement, we consider its array regions in order to create bound checks.

Based on the information calculated by the preliminary phase of array region analyses in PIPS, our algorithm uses some interesting proven properties:

1. If a MAY region of a node in the control flow graph is included in the declared dimensions of the array, no bound check is needed for the corresponding block of code.

2. If a MUST region of a node in the control flow graph contains elements which are outside the declared dimensions of the array, there is certainly a bound violation for the corresponding block of code. An error can be detected at compile time.

If neither of these two properties is satisfied, we consider the approximation of the region. In case of a EXACT region, if the exact bound checks can be generated, they will be inserted before the block of code. If not, like other cases of approximation, we continue to go down to the children nodes in the control flow graph.

```
      SUBROUTINE DOLOOP(K)
      REAL U(15)
      N = 2*K
C     <U(PHI1)-W-EXACT-{1<=PHI1, PHI1<=2K, N==2K}>
      DO 10 I = 1, N, 2
C     <U(PHI1)-W-EXACT-{PHI1==I+1,N==2K,1<=I,I<=2K}>
         U(I+1) = I
C     <U(PHI1)-W-EXACT-{PHI1==I,N==2K,1<=I,I<=2K}>
         U(I) = I+1
10    CONTINUE
      END

      SUBROUTINE DOLOOP(K)
      REAL U(15)
      IF (8.LE.K) STOP
      N = 2*K
      DO 10 I = 1, N, 2
         U(I+1) = I
         U(I) = I+1
10    CONTINUE
      END
```

Figure 4: Example of Insertion of unavoidable tests

Program	Static checks	Dynamic checks
101.tomcatv	97.70%	99.99%
102.swim	82.10%	99.99%
103.su2cor	94.50%	97.65%
104.hydro2d	93.80%	99.60%
107.mgrid	96.90%	99.54%
110.applu	96.40%	99.87%
125.turb3d	94.30%	99.99%
141.apsi	96.40%	99.99%
145.fpppp	92.80%	99.99%
146.wave5	94.00%	91.23%

Table 3: Percentage of static and dynamic checks eliminated: Insertion of unavoidable tests

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

Figure 4 shows an example of insertion of unavoidable tests based on array regions analyses. As the array regions for U and V are inside the declared dimensions of each array, though the approximation of V is MAY, no bound check is necessary for this program.

Our purpose is to generate a minimum number of bound checks with the available information from array regions. The main advantage of this algorithm is that it permits to detect the sure bound violations or to tell that there is certainly no bound violation as soon as possible, thanks to the context given by top-down analyses of insertion of tests and by preconditions. That is the goal of the second approach group as explained in section 2.

However, in the current implementation, some generated tests are still redundant, due to the same tests for different arrays. We could build a more sophisticated algorithm but with the supplementary phase *partial redundancy elimination* in PIPS, we have also achieved interesting results, as shown in Table 3.

The average percentage of eliminated static bound checks is 93.89% and that of eliminated dynamic bound checks is 98.78%. The numbers of removed dynamic checks for *103.su2cor* ans *146.wave5* are not very good because of the incrementation different than one in some loops in *103.su2cor*, i.e $DO10I = 1, LVEC, 2$, and because of the unstructured control flow graph in *146.wave5*. As soon as PIPS could deal well with these features, we could have better results for these cases.

5 Experimental results

We run our set of experiments with two set of compilation options: unoptimized and optimized. The optimized options are in the baseline of the submitted result from SUN for SPEC CFP95 (*f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch*). All the measurements were performed on an Ultra SPARC 360 Mhz workstation with the SUN Workshop f77 5.0 compiler.

As PIPS is a Fortran *source-to-source* compiler, the code generated by PIPS with its own range checking is compiled by SUN f77 and then executed. The execution time of the SUN versions with and without bound checking and the PIPS versions are compared in the Figure 5 and Figure 6. The CPU time here is given in seconds. Figure 5 and Figure 6, respectively, represent execution times for unoptimized and optimized compile option.

In both figures, for each program, the four bars show the execution time without range checking of SUN F77, with range checking of SUN F77, with PIPS Elimination of redundant tests and with PIPS Insertion of unavoidable tests.

We can see the overhead of range checking for *107.mgrid* and *110.applu* in Figure 6 which is caused by the big number of dynamic bound checks shown in Table 1. PIPS optimizing array bound checkers can reduce this overhead.

Comparing the execution time of PIPS code in the third and the last bar with that of SUN f77 bound checked code in the second bar, it is about 2.75 time faster for the unoptimized code and about 5.45 time faster for the optimized code. Specially, with *107.mgrid* in Figure 6, it takes 43m2s to run for the code compiled with option -C of SUN f77, but only 3m37s with our array bound checked code.

The execution times of programs with range checking added by PIPS are little bit longer, about 3%, than that of the unsafe program without bound checks.

On the other hand, our array bound checking optimizers can detect bound violations earlier than SUN f77 compiler with three benchmarks: *125.turb3d*, *141.apsi* and *145.fpppp*.

The times required by SUN and two version of PIPS for range checking are shown in Table 4. We compiled the 10 benchmarks with optimized

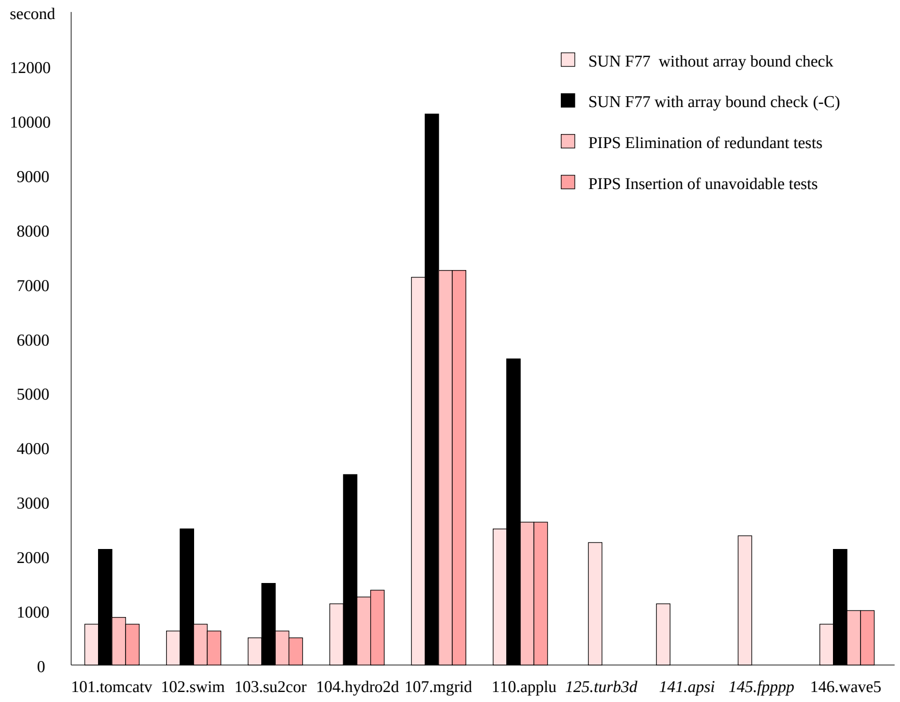

Figure 5: Execution time without optimization: SUN F77 without array bound checking, SUN F77 with array bound checking, PIPS Elimination of redundant tests, PIPS Insertion of inavoidable tests (Ultra SPARC 360MHz)

options only, because the production version is more important than the developement one. F77 is used with option -c for compilation only. The compilation times are in minutes and seconds. The second column presents the compilation time of SUN f77 with its array bound checking as well as other optimized options (*f77 -c -C -fast -xarch=v8plusa -fsimple=2 -xprefetch*). The two final columns show the sum of time required by PIPS to parse, analyse, optimize and generate Fortran code with its own range checking and time to compile this code by SUN f77 (*f77 -c -fast -xarch=v8plusa -fsimple=2 -xprefetch*).

In general, PIPS optimized array bound checkers take less time than SUN f77 5.0's one. The Elimination of redundant tests makes code run much faster and always has smaller cost. Except some long compile-time

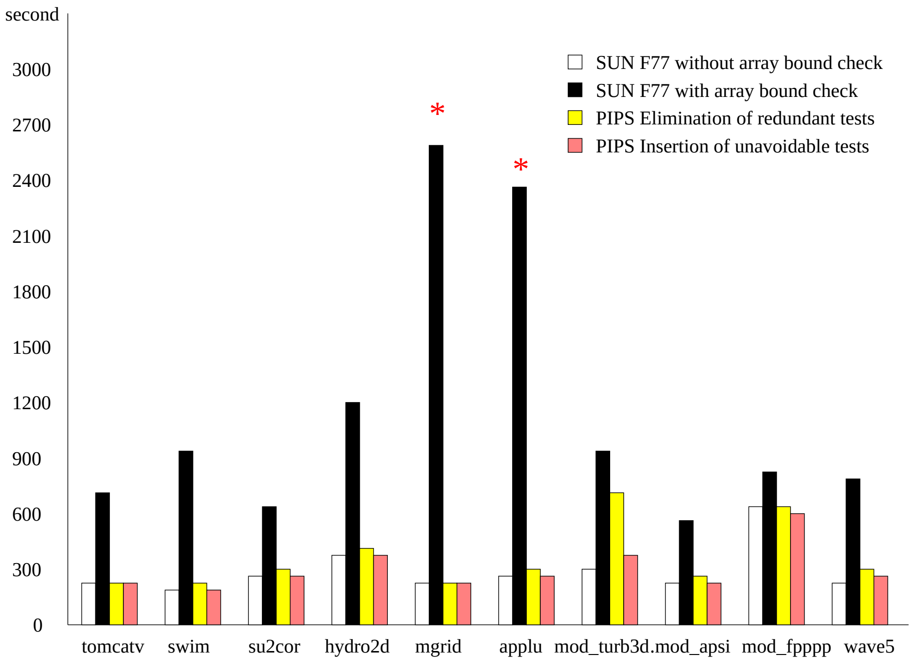

Figure 6: Execution time with optimization (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch): SUN F77 without array bound checking, SUN F77 with array bound checking, PIPS Elimination of redundant tests, PIPS Insertion of inavoidable tests (Ultra SPARC 360MHz)

cases such as (*103.su2cor, 141.apsi, 145.fpppp and 146.wave5*), the Insertion of unavoidable tests version does better with more removed dynamic checks and shorter run time. One explanation for the long compilation time in these 4 cases is due to the calculation time taken by feasibility test in PIPS.

With 31605 lines of code and 374 subroutines, it takes about 8m55s for the first version and 29m38s for the second version of PIPS array bound checkers. The real range check optimization phase takes a fraction of this compilation time but we have not attempted to calculate the exact time for it because the total time is acceptable with respect to SUN F77 compilation time.

Program	SUN -C	PIPS Elim.	PIPS Ins.
101.tomcatv	0:12	0:04	0:07
102.swim	0:19	0:06	0:17
103.su2cor	2:19	1:05	3:01
104.hydro2d	1:01	0:35	1:15
107.mgrid	0:36	0:11	0:44
110.applu	11:01	0:58	2:56
125.turb3d	0:44	0:31	1:22
141.apsi	3:11	2:18	8:45
145.fpppp	1:22	1:27	4:57
146.wave5	6:20	6:28	13:54

Table 4: Compilation time for array bound checking with optimization: SUN F77, PIPS Elimination of redundant tests, PIPS Insertion of unavoidable tests (Ultra SPARC 360MHz)

6 Conclusion

Our two optimizing array bound checkers show interesting results with high percentages of removed range checks and much faster execution times than the commercial SUN f77 compiler. The set of experiments is selected from a standard benchmark, the SPEC CFP95. The required optimization cost is small enough to use the safe versions of programs for production activities. We can use these array bound checking phases to debug programs by finding and fixing all bound violation errors.

We have to compare the two implemented approaches: Elimination of redundant tests and Insertion of unavoidable tests. The first implementation puts array bound checks everywhere and then removes the redundant checks. This approach seems simple and the number of eliminated tests depends on the strength of the data flow analyses, such as predicates over scalar integer variable values. The second implementation inserts useful checks directly by using array region analyses. The experimental results show that the second implementation produces better results in general. For the optimized code, it takes on average 90.57% of the execution time compared to the first implementation. For a small program like *101.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 bound checks is 4.37% for *146.wave5*.

Other transformations could be used by both implementations in order to remove more redundant checks but by comparing the cost and the results (Table 4, Figure 5 and Figure 6), we can conclude that it's not worth doing that. Our approach to optimize bound checking can also be applied to other imperative languages such as Pascal, C and Java. Array bound checkers should perform range checking interprocedurally, which means to check if there are bound violations between procedure calls. It is suprising that none of the SUN Workshop f77 version 5.0 nor IBM AIX XL Fortran compiler checks these array references interprocedurally. Thanks to the powerful interprocedural analyses designed and implemented in PIPS, we can exploit it to complete our array bound checkers.

References

[1] R. Bodik, R. Gupta, and V. Sarkar. Abcd : Eliminating array bounds checks on demand. In *PLDI 2000*, pages 321–333, Vancouver, British Columbia, Canada, 2000.

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

[3] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[4] B. Creusillet. *Analyses de régions de tableaux et applications*. PhD thesis, Centre de Recherche en Informatique, Ecole des Mines de Paris, December 1996.

[5] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Lecture Notes in Computer Science - Languages and Compilers for Parallel Computing*, pages 46–60. Springer-Verlag, August 1995.

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

[7] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its application in optimization. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 68–77, 1993.

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

[9] R. Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.

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

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

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

[13] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[14] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGLAN Notices - International Conference on Programming Language Design and Implementation*, pages 270–278, La Jolla, CA, USA, June 1995.

[15] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. *ACM SIGPLAN Symposium Compiler Contruction*, pages 114–119, June 1982.

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

[17] V. Menon and K. Pingali. A case for source-level transformations in matlab. In *ACM SIGPLAN 2nd Conference on Domain-Specific Languages*, volume 35, pages 53–65, Austin, Texas, USA, October 1999.

[18] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops : Java for technical computing. *ACM Transactions on Programming Languages and Systems*, pages 1–29, 1999.

[19] S. S. Muchnick. *Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[20] S. P.Midkiff, J. E.Moreira, and M.Snir. Optimizing array reference checking in java programs. *IBM Systems journal*, 37(3):409–453, 1998.

[21] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI 2000*, pages 182–195, Vancouver, British Columbia, Canada, 2000.

[22] M. Spezialetti and R. Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, June 1995.

[23] J. L. Steffen. Adding run-time checking to the portable c compiler. *Software-Practice and Experience*, 22(4):305–316, April 1992.

[24] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Symposium Principles of Programming Languages*, pages 132–143, 1977.

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

[26] J. Welsh. Economic range checks in pascal. *Software-Practice and Experience*, 8:85–97, 1978.

[27] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Notices*, volume 33, pages 249–257, 1998.