Efficient Intraprocedural Array Bound Checking

Nga T. V. Nguyen F. Irigoin C. Ancourt R. Keryell

CRI - ENSMP
77305 Fontainebleau, France
nguyen,irigoin,ancourt@cri.ensmp.fr

LIT - ENST Bretagne
29285 Brest, France
keryell@enst-bretagne.fr

ABSTRACT

Array bound checking is critical for code safety and debugging but users are not ready to trade much execution time for it. A considerable research work has been carried out during the past 25 years but experimental results are scarce. Commercial implementations are limited to intraprocedural array bound checking and are not really fulfilling user expectations for compilation and execution times.

Instead of designing a new specific algorithm, we implemented two algorithms representative of the main published approaches by re-using static analysis techniques available in PIPS, an interprocedural parallelizer. The first algorithm is based on redundant bound checking elimination. The second one is based on insertion of unavoidable tests.

Results with the SPEC CFP95 benchmarks show that commercial products could easily be improved using automatic analysis techniques and that user expectations can be fulfilled for most benchmarks. However, additional techniques should be used to obtain excellent results for all benchmarks.

Our approach to optimize bound checking can also be applied to other imperative languages such as Fortran, Pascal, Java when used for scientific applications.

Keywords

array reference, array bound checking, range checking, subscript out of range, bound violation, program verification.

1 INTRODUCTION

Array bound checking refers to determining whether all array references are within their declared range in all of their uses in a program. These array bound checks may be analyzed intraprocedurally or interprocedurally, depending on the need for accuracy. 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, security holes or failures. For the safety of execution, some languages such as Java require that a program only be allowed to access elements of an array that are part of the defined extent of the array.

But on the other hand, bound checking can be very expensive because every array access must be preceded by two bound checks per dimension to assert the legality of the access. This increases the size of the executable file, the compilation time and the execution time.

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

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

The objective of this project is to see if it would be possible to do efficient range checking by reusing optimization and automatic analysis techniques already implemented in commercial compilers. Array bound checking in this paper is intraprocedural but other interprocedural analyses are used optionally to gather more precise information.

The paper is organized as follows. Section 2 discusses the related work on optimizations of array range checking and explains what the two main 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 unavoidable tests. Experimental results for both approaches and analysis are reported in Section 5. The conclusion is given in the last section.

2 RELATED WORK

In 1977, Suzuki and Ishihata [27] implemented a system that inserts logical assertions before array element accesses and then uses theorem proving techniques to verify the absence of array range violations. Such techniques are often expensive and are restricted to programs written in a structured manner, i.e without goto statements.

Another approach was developed by V. Markstein, Cocke and P. Markstein [18], Asuru [19], R. Gupta [12, 13], Spezialetti and R. Gupta [25], Kolte and Wolfe [17]. 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 [13], 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 carries out the elimination of redundant checks by using the notions of *available* and *very busy* checks. These two algorithms use backward and forward data flow analyses to solve the problems. For the propagation of checks out of loops, R. Gupta identifies candidates for propagation which include invariants, loops with increment or decrement of one and variables of increasing or decreasing values. Then he uses check hoisting to move checks out of loops. However, the results are not very convincing because of the small size of his examples and because the optimizations have been applied by hand only.

In their article, Kolte and Wolfe [17] rely on a check implication graph where nodes are sets of range checks in canonical form and edges denote implications between these families. Like [13], 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 after determining the most efficient places to move bound checks to.

Their implementation in the Fortran compiler Nascent [17] is a combination and development of other techniques in range check optimization with experimental results that show the necessity of range check optimization, the effectiveness and cost of optimizations. However, high percentages, even 99.99%, of eliminated tests do not always mean faster execution times. This article lacks a comparison between the execution times of codes with and without optimized bound checks to show the impact of removed checks. Furthermore, there are no mentions of bound violations in PerfectClub (*mdg, spc77, trfd*) and Riceps benchmarks (*linpackd*) which are caused by formal array dimension declared 1.

The abstract interpretation approach proposed by Cousot and others [4] consider array range checking as an example for the automated verification of execution properties of programs. Like Schwarz et al. [24], 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 the different rules they use. 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 checks, they could only take advantage of completely redundant checks. So the run time overhead of partial redundant checks that cannot be evaluated at compile time still remains. Also in this approach is the model checking group [7] who uses fix point acceleration techniques to help automated verification of program.

The implementation of the algorithm Eliminating Array Bound Checks on Demand in [2], which is based on an extended Static Single-Assignment graph representation, can remove about 45% of dynamic bound checks of a representative set of Java programs.

Other articles by Midkiff, Moreira, Snir and M. Gupta [22, 20] describe another approach to 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 because of its precise exception semantics.

Although there are many different techniques for array bound checking optimization, we can partition them into two main possible approaches. The first approach puts array bound checks at every array reference and removes a check if it is redundant [18, 12, 19, 13, 17]. In the second approach, one only puts array bound checks at places where one doesn't know how to prove that they are useless [27, 22, 24].

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

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

So both approaches have advantages and drawbacks when comparing the number of transformations and the number of analyses. A 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 without hoisting, and the second one is based on optimized test insertion without code replication. Both were implemented in PIPS, *Paralléliseur Interprocédural de Programmes Scientifiques* [16, 15], a research project developed at Ecole des Mines de Paris. It consists of several analysis phases dealing with dependences, transformers, preconditions, array regions and of program transformations such as loop transformations, constant folding, dead code elimination,...

3 ELIMINATION OF REDUNDANT TESTS

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

In essence, a *partial redundancy* [21] is a computation that is done more than once on some path through a flow graph, i.e., some path through the flow graph contains a point at which the given computation has already been computed and will be computed again. To eliminate redundant bound checks, we use information 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. They are propagated from bottom to top in the abstract syntax tree.

Preconditions in PIPS are affine predicates over scalar integer variable values à la Cousot/Halbwachs [4]. These predicates hold just before the execution of the corresponding statement. 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 preconditions of the following statements. Unstructured programs [16] are also handled in PIPS.

For each module in program, the algorithm Elimination of redundant tests works as follow:

1. Generate non-trivial bound checks for every statement that has array references. The form of the test is: `IF` (ref.`LT`.lower .`OR`. ref.`GT`.upper) `STOP`

2. Compute transformers and preconditions for the new code with added bound checks.

3. For each bound check, test the feasibility of the system that is builded from the precondition of the current statement and the bound check.

- If the system is infeasible, the bound check is `FALSE` and is removed.
- If the bound check is `TRUE` with respect to the precondition, a bound violation is detected at compile time.
- If the two above cases are not satisfied, the bound check remains in the program.

```
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 1: Example of Elimination of redundant test

Figure 1 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`. The check `I+1.GT.15` in the first test makes the other computation of upper bound check after it become redundant. So in the final code, we have only one bound check to compute but it is left inside the loop because no hoisting is used. However, if at the entry point of the module we have other information such as `K<=5`, the derived precondition `I<=10` will permit us to eliminate all bound checks in this code. This kind of information comes from interprocedural analysis where the preconditions of the call sites are translated to the callee's name space, using the relations between actual and formal parameters, and between the declarations of global variables. The initial precondition of the callee is the convex hull of all translated preconditions.

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

After the partial redundancy elimination transformation, the number of generated tests is greatly reduced. If a bound violation is detected, a stop message indicates that the subscript is out-of-range in a given array, on which dimension and in which subroutine. PIPS translates Fortran programs into instrumented Fortran codes with source bound checks which are then compiled and executed using their standard input data sets. The experimental results with this approach are given in Section 5.

4 INSERTION OF UNAVOIDABLE TESTS

The second implementation is based on an array region analysis phase. An **array region**, as defined in [6, 15], 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 relied upon or not. The approximation of a region is `EXACT` if the region exactly represents the requested set of array elements. `MAY` and `MUST`, respectively is an over- and under-approximation of `EXACT`: `MUST` $\subseteq$ `EXACT` $\subseteq$ `MAY`. For instance, the region $< \mathtt{A}(\phi_1,\phi_2) - \mathtt{EXACT} - \{\phi_1 = \mathtt{I}, \phi_1 = \phi_2\} >$ corresponds to an array reference `A(I,I)`, where the region parameters ϕ_1 and ϕ_2 respectively represent the first and second dimensions of `A`.

Regions are built bottom-up, intraprocedurally, from the deepest nodes to the largest compound statement nodes in the *hierarchical control flow graph* [16]. 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 conservative. Intraprocedural region analysis has to deal with elementary statements like assignments, basic blocks, tests, loops and unstructured statements.

Our second array bound checker inserts unavoidable tests in a top-down analysis: from the largest compound statement node down to elementary statements. At each statement, we use its array regions to test the feasibility of bound checks. Our algorithm uses the two following 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 `EXACT` region, if the exact bound checks can be generated, they are inserted before the block of code. If not, we continue to go down to the children nodes in the hierarchical control flow graph. The algorithm completes because we can always generate bound checks directly for array references of elementary statements 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 2: Example of Insertion of unavoidable tests

Figure 2 shows the example in Section 3 with the Insertion of unavoidable tests approach. The exact region of array `U` (line 4) gives the only one bound check in the final code which is outside the loop. This is a point in favor of the second approach because it may lift test out of loops automatically while the first one does not.

Our purpose is to generate a minimum number of bound checks with the available information from array regions. Bound checks are inserted outside loops and at the beginning of the program. The other advantage of this algorithm is that it detects the sure bound violations or indicates that there is certainly no bound violation as early as possible, thanks to the context given by top-down analyses of insertion of tests. That is the goal of the second approach group as explained in section 2. Our region-based algorithm can be parameterized with respect to different notions of array regions, not only convex polyhedra region. Guarded regions, list of regions [10] or dimension per dimension regions could be used to improve the computation time of convex regions.

5 EXPERIMENTAL RESULTS

We used the SPEC CFP95 benchmark [9], which contains 10 applications written in Fortran. These are scientific benchmarks with floating point arithmetic and many of them have

			Range checks		Elimination		Insertion	
Program	**Lines**	**Subs**	**Static**	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**
tomcatv	190	1	304	4933x10^7	100.00%	100.00%	100.00%	100.00%
swim	429	6	772	6991x10^7	98.45%	99.99%	84.72%	99.99%
su2cor	2332	35	4460	4810x10^7	96.59%	97.54%	94.48%	97.66%
hydro2d	4292	42	2016	6530x10^7	96.13%	94.14%	93.85%	99.46%
mgrid	484	12	1162	17608x10^7	94.92%	99.60%	97.93%	99.66%
applu	3868	16	9562	11562x10^7	99.62%	97.09%	96.41%	99.87%
turb3d	2101	23	1852	6081x10^7	93.09%	99.99%	94.33%	99.99%
mod_turb3d					97.62%	59.85%	98.97%	85.57%
apsi	7361	96	7172	3936x10^7	95.87%	99.99%	96.42%	99.99%
mod_apsi					98.02%	99.90%	99.79%	99.99%
fpppp	2784	38	2894	2928x10^7	92.81%	99.99%	92.81%	99.99%
mod_fpppp					94.61%	97.02%	95.82%	97.40%
wave5	7764	105	10546	3404x10^7	94.66%	86.86%	94.01%	91.29%

Table 1: SPEC CFP95: numbers of lines, subroutines, static and dynamic range checks, percentages of static and dynamic checks eliminated by Elimination of redundant tests and Insertion of unavoidable tests

been derived from publicly available application programs. Each benchmark contains a large amount of subscripted references to arrays. The codes are instrumented to compute the number of dynamic range checks. Table 1 summarizes relevant information for each benchmark in SPEC CFP95. Note that three of them do not meet the Fortran standard for array declaration and reference. We added proper bounds [1] to the declarations in *turb3d, apsi* and *fpppp* to avoid premature aborts due to bound violations.

Eliminated Bound Checks

The last four columns in Table 1 show the percentages of the numbers of bound checks eliminated by the two approaches: 96.34% at compile time and 95.40% at run time for Elimination of redundant tests; 95.35% at compile time and 97.08% at run time for Insertion of unavoidable tests. The number of dynamic checks is more important because it decides the execution time. We can see that the second approach is uniformly better, because tests can be lifted out of loops as shown in the running example (Section 4).

Compilation Times

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

The compile time for the second version is longer than that of the first one, especially for *mgrid, mod_apsi, mod_fpppp* and *wave5*. It is due to the feasibility testing in PIPS to compute array regions. This could be improved by a more sophisticated implementation of array regions as mentioned above. As shown in Table 1, the percentage of removed checks of Insertion of unavoidable tests is high enough to pay for this tradeoff.

As PIPS is a Fortran *source-to-source* compiler, the code generated by PIPS with its own range checking is then compiled by other compilers. We measured the compilation times taken by PIPS as a preprocessor and by SUN Workshop F77 5.0 compiler for PIPS generated codes. The original codes of SPEC CFP95 are also compiled with and without the array range checking option of SUN. Table 1 shows shorter times for the two versions of PIPS than for SUN (see columns Total of Elimination of redundant tests and of Insertion of unavoidable tests and column With C of SUN F77).

Execution Times

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

The code generated by PIPS with its own range checking is compiled by other compilers to generate executable files. Experiments are 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. Their -C options are for intraprocedural array bound checking only. There is no range checking option for SGI F77 and GNU G77 compilers and we had to leave it out. For IBM, because an internal compiler error occurred when compiling the Fortran code with options -O5 and -C together, we used *-O3*. In addition, there is an IO error for *apsi* so we do not have results for this benchmark on the IBM machine.The execution times of

[1] We did not use Fortran assumed-size array [1] with an asterisk because it prevents array bound checking.

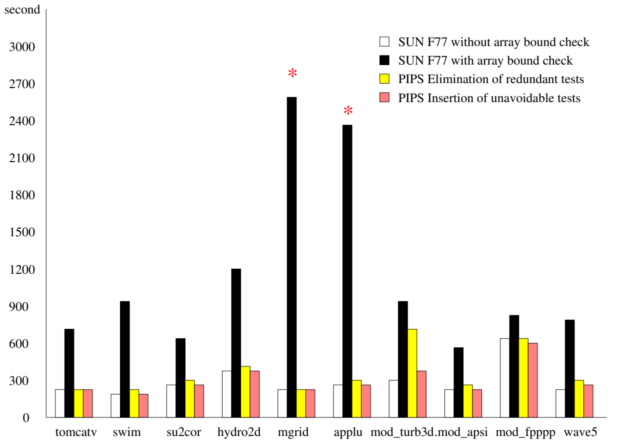

Figure 3: Execution time: SUN F77 and PIPS - Ultra SPARC 360MHz - Optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

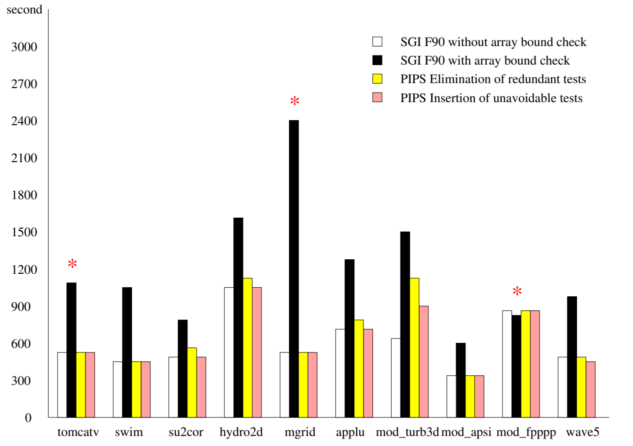

Figure 4: Execution time: SGI F90 and PIPS - SGI MIPSpro F90 7.3, O2 R5000 195MHz, IRIX 6.3 - Optimized code (f90 -Ofast=ip32_5k)

codes obtained with and without the bound checking option of these compilers and with the PIPS versions are provided in Figure 3, Figure 4 and Figure 5.

We can see the overheads of range checking in *mgrid* and *applu* for SUN (Fig 3), *mgrid* and *swim* for SGI (Fig 4) and *tomcatv* and *mod_turb3d* for IBM (Fig 5). PIPS optimizing array bound checkers work very well for *tomcatv, swim, mgrid* and *applu*. These benchmarks have more dynamic bound checks than others as shown in Table 1.

As the range checking of the IBM compiler is already optimized, the PIPS versions work better than IBM in general but worse for *mod_turb3d* benchmark. The reason is that analyses of non-linear expressions haven't been implemented yet in PIPS.

Comparing the execution time of PIPS codes with that of other bound checked codes, on average, PIPS Elimination of redundant tests is about 3.94 times faster than SUN, 1.88 times faster than SGI and 1.03 times faster than IBM. PIPS Insertion of unavoidable tests is about 4.37 times faster than SUN, 2.02 times faster than SGI and 1.07 times faster than IBM. The execution times of programs with range checking added by PIPS are slightly longer than that of the unsafe programs without bound checks. On average, these times for PIPS Elimination of redundant tests are about 19.29% longer for SUN, 7.05% longer for SGI and 16.59% longer for IBM. For PIPS Insertion of unavoidable tests, they are about 5.33% longer for SUN, 0.61% longer for SGI and 10.62% longer for IBM.

Also, our array bound checking optimizers can detect bound violations earlier than other compilers in three original benchmarks violating the standard for array references: *turb3d*, *apsi* and *fpppp*. Especially with SGI, since the programs did not stop on the system we used when they reached an out-of-bound trap.

Discussion and Improvements

The experimental results show the effectiveness and the optimization cost of our two implementations: 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 seems simple and the number of eliminated tests depends on the strength of data flow analyses, such as predicates over scalar integer variable values, used to perform the elimination.

The second implementation inserts useful checks directly by using array region analyses. It produces better results with a higher number of dynamic removed checks and faster execution times. For a small program like *tomcatv*, the differences between the two approaches are limited, but for large programs with more than 2000 lines of code, there are clear differences. The maximum improvement in dynamic removed bound checks is 5.32% for *hydro2d*. The main advantage of this top-down analyses approach is that it detects the sure bound violations or indicates that there is certainly no bound violation as soon as possible.

Result analysis shows the importance of code quality. Proper array declarations are needed to avoid out-of-bound errors caused by standard violation. Furthermore, some benchmarks require more sophisticated techniques or modifications such as cloning, parameter checking, scalarization for dealing with indirections, scalarization for loop bounds, loop increments different than one,... For example, we can improve the percentage of removed checks in Elimination of redundant tests from 94.14% to 99.50% for *hydro2d* by cloning the subroutine `ADLEN` which has two totally different behaviors for two parameter values: "half" and "full" step. The execution time on SUN is reduced by 10%. The elimination percentage goes up to 100% from 97.09% for *applu* by

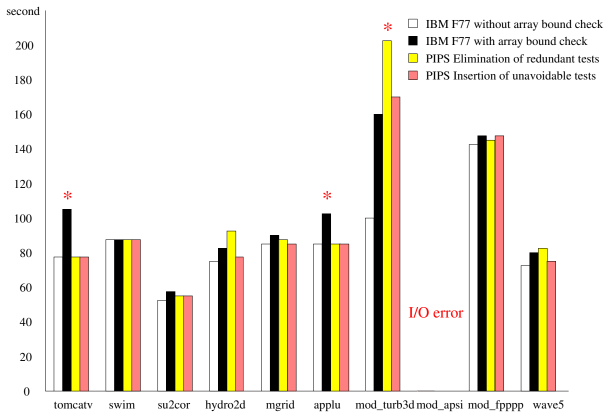

Figure 5: Execution time: IBM F77 and PIPS - IBM XL F77 7.1, RS/6000 44P-270 375MHz 4 CPU, AIX 4.3 - Optimized code (f77 -O3 -lmass)

adding one `STOP` statement after the parameter checking that is done for lower bound test but not for upper bound test of read variables (`NX,NY,NZ` in the main program `APPLU`). A 5.4% of decrease of the execution time is then measured on SUN.

Some analysis for non-linear expressions that have direct impact on array range checking have not been implemented in PIPS, so we do not have very good results for *mod_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 [22, 20] when MAY regions give necessary but not sufficient conditions for a bound violation to occur. However, the code size increase may raise problems that go beyond a simple reuse of existing techniques.

Finally we observe that the percentage of removed checks is not an accurate predictor of slowdown. For example, the execution time of code without bound checks for *swim* is only 92.35% of that of code with 99.99% of dynamic checks removed (see Table 1 and Figure 3). So after array bound checking optimization, it is necessary to also compare the execution times of generated codes which is missing in [17].

6 CONCLUSION

We have measured compilation and execution times for the standard SPEC CFP95 benchmarks with three different compilers and with two experimental algorithms. Within the SUN environment, we measured shorter compilation times using our two experimental source-to-source array bound checkers followed by F77 than using F77 alone with its -C option. The compilation time speedups are 3.03 for Elimination of redundant tests and 1.1 for Insertion of unavoidable tests.

At run time, the slowdowns measured for SUN and SGI compilers are large enough to make improvement easy. We mostly obtained speedups by adding array bound checks of the Insertion of unavoidable tests version, in comparison with the execution time with no bound checks of the SGI compiler. This is not the case with IBM XLF compiler 7.1.0.0 which nevertheless is not uniformly efficient and which breaks down with an internal error when combining options -C and -O5. However, our execution times are in the same range as IBM's, 7 times out of 9 slightly better.

We could not directly compare our results with those presented in [17] about the Perfect Club and Riceps benchmarks because the authors do not include execution times and because percentage of eliminated tests is not always a good predictor of execution time. However, their best figures are in the very same range as ours and it is very interesting to see that specific techniques do not work better than re-used techniques.

The implementations in this paper suggest that commercial products with automatic analyses could easily be improved to perform efficient array bound checking. Less than 3000 additional lines of C code are sufficient to implement both approaches 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 approach to optimizing bound checking could also be applied to other imperative languages for scientific applications that require software verification such as Ada, Java,...

Although interprocedural analyses may be used to improve the result, the range checking studied in this paper is purely intraprocedural. We are now planning to study interprocedural array bound checking as a whole program transformation because related bugs are much more difficult to track than intraprocedural ones.

REFERENCES

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

[2] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Programming Language Design and Implementation 2000*, pages 321–333, Vancouver, British Columbia, Canada, 2000.

[3] 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.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5^{th} ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

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

[6] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Lecture Notes in Computer Science - Languages and*

	Elimination				Insertion				SUN F77	
Program	**Speed**	**PIPS**	**SUN**	**Total**	**Speed**	**PIPS**	**SUN**	**Total**	**Without C**	**With C**
tomcatv	55.50	0:02	0:02	0:04	22.20	0:05	0:02	0:07	0:03	0:13
swim	70.75	0:04	0:02	0:06	23.58	0:12	0:05	0:17	0:04	0:20
su2cor	39.40	0:40	0:25	1:05	10.56	2:28	0:33	3:01	0:33	2:23
hydro2d	107.06	0:16	0:19	0:35	39.84	0:43	0:32	1:15	0:32	1:03
mgrid	85.25	0:04	0:07	0:11	8.97	0:38	0:06	0:44	0:08	0:37
applu	74.97	0:33	0:25	0:58	20.28	2:02	0:54	2:56	0:57	11:12
mod_turb3d	86.07	0:15	0:16	0:31	19.86	1:05	0:17	1:22	0:17	0:44
mod_apsi	55.80	1:16	1:02	2:18	9.18	7:42	1:03	8:45	1:16	3:15
mod_fpppp	50.69	0:42	0:45	1:27	8.83	4:01	0:56	4:57	0:54	1:22
wave5	35.24	5:03	1:25	6:28	8.96	12:04	1:50	13:54	2:04	6:20

Table 2: Compilation times: Elimination of redundant tests (Speed, PIPS, SUN and Total), Insertion of unavoidable tests(Speed, PIPS, SUN and Total), SUN F77 without and with array bound checking (Ultra SPARC 360MHz, Solaris 7) - Optimizing options (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch -c)

Compilers for Parallel Computing, pages 46–60. Springer-Verlag, August 1995.

[7] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Software Tools for Technology Transfer*, 2000.

[8] 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.

[9] J. J. Dujmovic and I. Dujmoviv. Evolution and evaluation of SPEC benchmarks. In *ACM:SIGMETRICS*, pages 2–9, 1998.

[10] 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.

[11] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

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

[20] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: JAVA for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, March 2000.

[21] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

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

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

[24] B. Schwarz, W. Kirchgassner, and R. Landwehr. An optimizer for Ada - design, experiences and results. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 175–184, Atlanta, Georgia, USA, June 1988.

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

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

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

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

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

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