Efficient Intraprocedural Array Bound Checking

Thi Viet Nga NGUYEN
François IRIGOIN
Corinne ANCOURT
CRI - ENSMP
77305 Fontainebleau France
{nguyen,irigoin,ancourt}@cri.ensmp.fr

Ronan KERYELL
LIT - ENST Bretagne
29285 Brest France
keryell@cri.ensmp.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 fot compilation and execution times.

Instead of designing a new specific algorithm, we implemented two algorithms representative of the main published approaches by re-using automatic parallelization 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 parallelization techniques and that user expectations can be fulfilled for most benchmarks. However, additional techniques would have 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 analysed 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 developement 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 developement 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 parallelization techniques already implemented in commercial compilers.

The paper is organized as follows. Section 2 contains experimental results obtained with three commercial compilers for the SPEC CFP95 benchmark. Section 3 discusses the related work on optimizations of array range checking and explains what the two main approaches are, with their advantages and drawbacks. Section 4 presents our first implementation based on elimination of redundant tests. Section 5 presents the second implementation based on insertion of

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

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

unavoidable tests. Experimental results for both approaches and analysis are reported in Section 6. The conclusion is given in the last section.

2. COMMERCIAL COMPILERS

Our first set of experiments was 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 GNU G77 compilers and we had to leave it out.

We used the SPEC CFP95 benchmark (*www.spec.org*), which contains 10 applications written in Fortran. These are scientific benchmarks with floating point arithmetic and many of them have been derived from publicly available application programs. Each benchmark contains a large amount of subscripted references to arrays. 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.

We measured the execution times of codes with and without the range checking option using the SPEC measurement guidelines. Figure 1 and Figure 2 represent CPU times for SPEC CFP95 programs with their reference input data, compiled by SUN F77 5.0, executed on an Ultra SPARC 5 360MHz workstation under Solaris 7. These measurements were undertaken with no additional compile time flags (Figure 1) and with optimization flags (Figure 2). These options are in the base line of the submitted result for SPEC CFP95, *-fast -xarch=v8plusa -fsimple=2 -xprefetch*.

The execution times measured on SGI's and IBM's architectures are shown respectively in Figure 3 and Figure 4. As SGI MIPSpro F77 compiler does not have an array bound checking option, we used the F90 compiler. The results were obtained on an O2 R5000 195MHz machine, IRIX 6.3 with compile time option *-Ofast=ip32_5k*. For IBM, because an internal compiler error occured when compiling the Fortran code with options -O5 and -C together, we used *-O3 -lmass*.

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

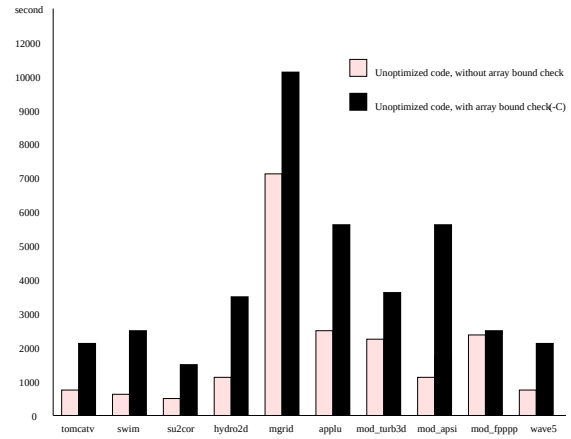

Figure 1: Execution times with and without array bound checking (SUN F77 5.0, Ultra SPARC 360MHz, Solaris 7) - Unoptimized code for development

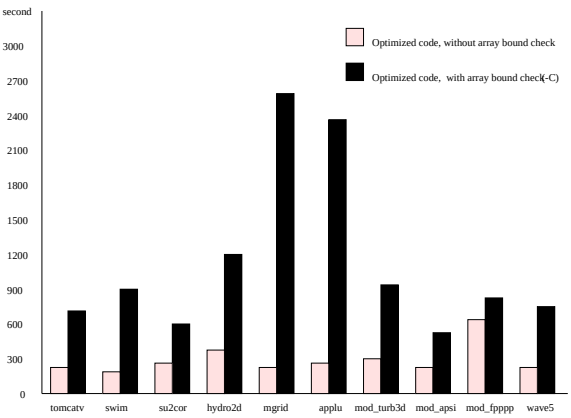

Figure 2: Execution times with and without array bound checking (SUN F77 5.0, Ultra SPARC 360MHz, Solaris 7) - Optimized code for production (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

In addition, there is an IO error for *apsi* so we do not have results for this benchmark on the IBM machine. All benchmarks were compiled and run on a RS/6000 44P-270 model, 4 CPUs 375 MHz, with operating system AIX 4.3.

The four figures generally show the large execution overheads for commercial range checking (-C option) with both unoptimized and optimized codes. On average, IBM XL Fortran 77 is only 1.13 times slower but it is 1.65 times slower for *mod_turb3d* and *mod_apsi* fails to execute. SGI MIPSpro F90 7.3 is 1.74 times slower. SUN compiler is 3.13 times slower for optimized code and 2 times slower for unoptimized code.

Array bound checking generates much larger codes and takes a much longer compilation time for all cases. Table 2 is an example of the sizes of executable files and compilation time for the SUN F77 5.0 optimized version. This slowdown does not meet user needs and explains why a considerable research work has already been devoted to the problem over

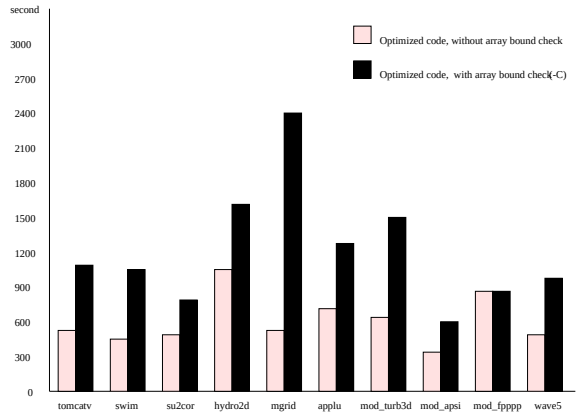

Figure 3: Execution times with and without array bound checking (SGI MIPSpro F90 7.3, O2 R5000 195MHz, IRIX 6.3) - Optimized code (f90 -Ofats=ip32_5k)

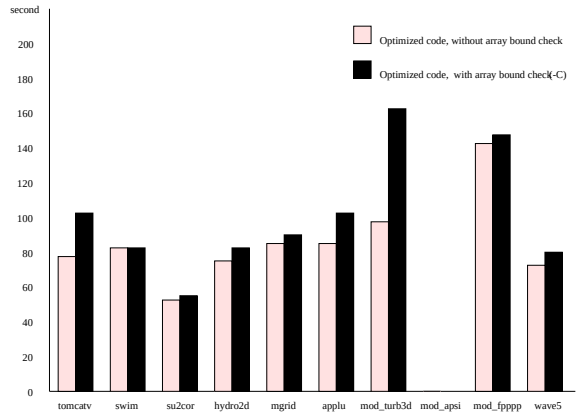

Figure 4: Execution times with and without array bound checking (IBM XLF F77 7.1, RS/6000 44P-270 375MHz 4CPU, AIX 4.3) - Optimized code (f77 -O3 -lmass)

the past 25 years.

3. RELATED WORK

In 1977, Suzuki and Ishihata [25] 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 developped by V. Markstein, Cocke and P. Markstein [15], Asuru [16], R. Gupta [9, 10], Spezialetti and R. Gupta [23], 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 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.

	File size		Compilation time	
Program	**Wo -C**	**With -C**	**Wo -C**	**With -C**
tomcatv	100336	128744	0:03	0:13
swim	112004	174792	0:04	0:20
su2cor	221752	536656	0:33	2:23
hydro2d	229076	386840	0:32	1:03
mgrid	114852	225324	0:08	0:37
applu	212912	999092	0:57	11:12
mod_turb3d	149364	260712	0:17	0:44
mod_apsi	361508	791028	1:16	3:15
mod_fpppp	201640	255820	0:54	1:22
wave5	552984	1459740	2:04	6:20

Table 2: Executable file size (in bytes) and compilation time (in minutes and seconds) with and without array bound checking (SUN F77 5.0, Ultra SPARC 360MHz, Solaris 7) - Optimized code for production (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

In their article, Kolte and Wolfe [14] rely on a check implication graph where nodes are sets 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 after determining 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 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 comparision 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 array dimension declared 1. However, experimental comparison between their work and the techniques presented in this paper is provided in Section 6.

The abstract interpretation approach proposed by Cousot and others [4] consider array range checking as an example for the automatical verification of execution properties of

programs. Like Schwarz et al. [22], 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.

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 [20, 18] 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 [15, 9, 16, 10, 14]. In the second approach, one only puts array bound checks at places where one doesn't know how to prove that they are useless [25, 20, 22].

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, *Parallèliseur Interprocédural de Programmes Scientifiques* [13, 12], 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, unreachable code elimination,...

4. ELIMINATION OF REDUNDANT TESTS

Our first implementation of a range check optimizer is based on the two-step algorithm described below:

1. Generate bound checks for every array reference

However if one bound check is always true, for example $N - 1 <= N$, or already exists in the list of bound checks for the same statement, it doesn't need to be added. Checks are inserted before the current statement with error messages to describe the bound violation.

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 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 or procedure call. They are propagated from bottom to top in the call tree and in the abtract 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. 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 preconditions of the following statements.

If some path through the flow graph contains a point at which a bound check has already been computed and 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.

Furthermore, the precondition of a statement can determine

```
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 5: Example of Elimination of redundant test

Program	Static checks	Dynamic checks
tomcatv	100.00%	100.00%
swim	98.45%	99.99%
su2cor	96.59%	97.54%
hydro2d	96.13%	94.14%
mgrid	94.92%	99.60%
applu	99.62%	97.09%
turb3d	93.09%	99.99%
mod_turb3d	97.62%	59.85%
apsi	95.87%	99.99%
mod_apsi	98.02%	99.90%
fpppp	92.81%	99.99%
mod_fpppp	94.61%	97.02%
wave5	94.66%	86.86%

Table 3: Percentage of static and dynamic checks eliminated: Elimination of Redundant Tests

if a bound test is always true or false. If it is alway true, there is a bound violation that can be detected at compile time. If it is always false, the test should be removed.

Figure 5 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 arrays 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 because no hoisting is used.

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 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. Table 3 shows the percentages of the numbers of bound checks eliminated at compile time and run time for SPEC CFP95 benchmarks by this implementation. The average percentages of removed bound checks are 96.34% at compile time and 95.40% at run time. We discuss the execution times of the instrumented code and the PIPS processing time necessay to achieve these results in the Experimental Results section.

5. INSERTION OF UNAVOIDABLE TESTS

This second implementation is based on an array region analysis phase. An array region, as defined in [6, 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.

```
   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 6: Example of Insertion of unavoidable tests

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. In fact, MAY and MUST, respectively is an over- and under-approximation of EXACT: MUST $\subseteq$ EXACT $\subseteq$ MAY. For instance, the region $< \mathtt{A}(\phi_1, \phi_2) - \mathtt{EXACT} - \{\phi_1 == \mathtt{I}, \phi_1 == \phi_2\} >$ 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. 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 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.

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 or different types 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 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 will be 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.

Figure 6 shows an example of Insertion of Unavoidable Tests based on array regions analyses. As the array regions for `U` and `V` are within 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 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 and by preconditions. That is the goal of the second approach group as explained in section 2. In addition, there is no need for explicit check hoisting in this version.

Program	Static checks	Dynamic checks
tomcatv	100.00%	100.00%
swim	84.72%	99.99%
su2cor	94.48%	97.66%
hydro2d	93.85%	99.46%
mgrid	97.93%	99.66%
applu	96.41%	99.87%
turb3d	94.33%	99.99%
mod_turb3d	98.97%	85.57%
apsi	96.42%	99.99%
mod_apsi	99.79%	99.99%
fpppp	92.81%	99.99%
mod_fpppp	95.82%	97.40%
wave5	94.01%	91.29%

Table 4: Percentage of static and dynamic checks eliminated: Insertion of Unavoidable Tests

The average percentages of eliminated bound checks are 95.35% at compile time and 97.08% at run time (Table 4).

6. EXPERIMENTAL RESULTS

In this section, we give experimental results about compilation and execution times for SPEC CFP95 benchmarks, a comparison with the related work done by Kolte and Wolfe in [14] in PLDI'95 for the Perfect Club and Riceps benchmarks. Some additional techniques needed to obtain excellent results for recognized benchmarks as well as the comparison between two implementations are also reported in this section.

6.1 Compilation times

The compilation speeds obtained with PIPS to parse, analyze, optimize and generate Fortran code with its own range checking for SPEC CFP95 are shown in Table 5. 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 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 calculation time taken by feasibility testing in PIPS for array regions. This could be improved by a more sophisticated implementation of array regions, such as guarded regions, list of regions as in [8] or dimension per dimension regions. As shown in Table 4, 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 for PIPS

	Elimination of Redundant Tests			Insertion of Unavoidable Tests			SUN F77	
Program	**PIPS**	**SUN**	**Total**	**PIPS**	**SUN**	**Total**	**Without C**	**With C**
tomcatv	0:02	0:02	0:04	0:05	0:02	0:07	0:03	0:13
swim	0:04	0:02	0:06	0:12	0:05	0:17	0:04	0:20
su2cor	0:40	0:25	1:05	2:28	0:33	3:01	0:33	2:23
hydro2d	0:16	0:19	0:35	0:43	0:32	1:15	0:32	1:03
mgrid	0:04	0:07	0:11	0:38	0:06	0:44	0:08	0:37
applu	0:33	0:25	0:58	2:02	0:54	2:56	0:57	11:12
mod_turb3d	0:15	0:16	0:31	1:05	0:17	1:22	0:17	0:44
mod_apsi	1:16	1:02	2:18	7:42	1:03	8:45	1:16	3:15
mod_fpppp	0:42	0:45	1:27	4:01	0:56	4:57	0:54	1:22
wave5	5:03	1:25	6:28	12:04	1:50	13:54	2:04	6:20

Table 6: Compilation times in minutes and seconds: Elimination of Redundant Tests (PIPS, SUN and Total), Insertion of Unavoidable Tests (PIPS, SUN and Total), SUN F77 without array bound checking, SUN F77 with array bound checking, (Ultra SPARC 360MHz, Solaris 7) - Optimizing options (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch -c)

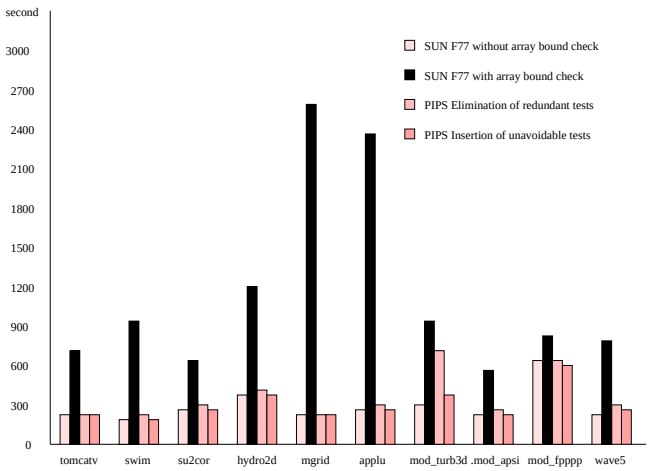

Figure 7: Execution time: SUN F77 without array bound checking, SUN F77 with array bound checking, PIPS Elimination of Redundant Tests, PIPS Insertion of Unavoidable Tests (SUN Workshop F77 5.0, Ultra SPARC 360MHz, Solaris 7) - Optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

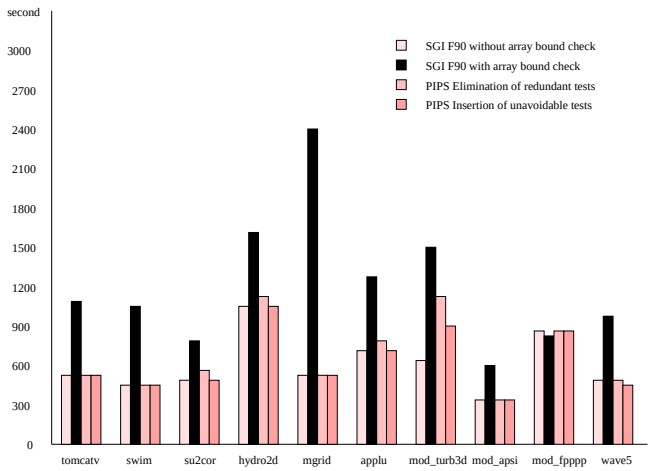

Figure 8: Execution time: SGI F90 without array bound checking, SGI F90 with array bound checking, PIPS Elimination of Redundant Tests, PIPS Insertion of Unavoidable Tests (SGI MIPSpro F90 7.3, O2 R5000 195MHz, IRIX 6.3) - Optimized code (f90 -Ofast=ip32_5k)

Program	PIPS Elimination	PIPS Insertion
tomcatv	55.50	22.20
swim	70.75	23.58
su2cor	39.40	10.56
hydro2d	107.06	39.84
mgrid	85.25	8.97
applu	74.97	20.28
mod_turb3d	86.07	19.86
mod_apsi	55.80	9.18
mod_fpppp	50.69	8.83
wave5	35.24	8.96

Table 5: Compilation speed (lines of code/seconds) of PIPS array bound checkers: Elimination of Redundant Tests and Insertion of Unavoidable Tests (Ultra SPARC 360MHz, Solaris 7)

generated codes and original SPEC CFP95 benchmarks on our SUN platform (Table 6).

This table shows shorter times for the two versions of PIPS than for SUN F77 (see column Total of Elimination of Redundant Tests and of Insertion of Unavoidable Tests and column With C of SUN F77).

6.2 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.

The code generated by PIPS with its own range checking is compiled by other compilers to generate executable files. The execution times of codes obtained with and without the bound checking option of three commercial compilers

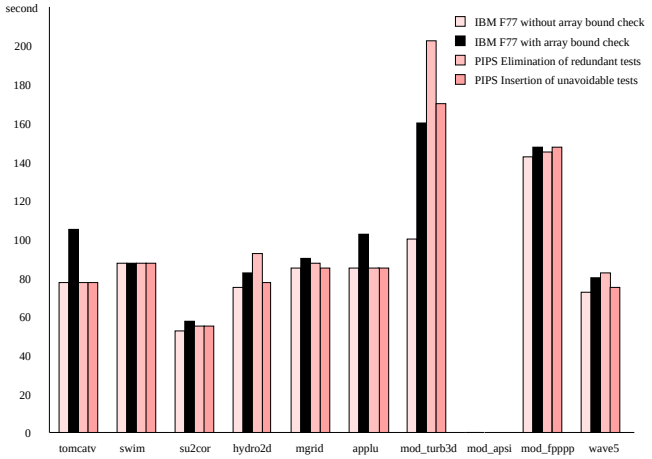

Figure 9: Execution time: IBM F77 without array bound checking, IBM F77 with array bound checking, PIPS Elimination of Redundant Tests, PIPS Insertion of Unavoidable Tests (IBM XL F77 7.1, RS/6000 44P-270 375MHz 4 CPU, AIX 4.3) - Optimized code (f77 -O3 -lmass)

and with the PIPS versions are provided in Figure 7, Figure 8 and Figure 9. The CPU times are given in seconds.

We can see the overheads of range checking in *mgrid* and *applu* for SUN (Fig. 7), *mgrid* and *swim* for SGI (Fig. 8) and *tomcatv* and *mod_turb3d* for IBM (Fig.9). 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.

	PIPS		NASCENT
Program	Elimination	Insertion	Best result (ALL)
arc2d	97.66%	97.69%	99.96%
bdna	95.90%	97.75%	98.44%
dyfesm	97.56%	99.51%	98.73%
qcd	96.12%	99.50%	97.06%
simple	99.96%	99.98%	99.97%

Table 7: Percentage of dynamic checks eliminated on PerfectClub and Riceps benchmarks

6.3 Comparison with Kolte and Wolfe

Among the research cited in the Related Work section, the article *Elimination of Redundant Array Subscript Range Checks* by Kolte and Wolfe contains 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 show that simple optimizations such as preheader insertion with loop-limit-substitution of linear checks greatly reduce the number of dynamic checks removed. Their experiments are based on 10 Fortran programs from the Perfect, Riceps and Mendez benchmarks. However, the article lacks a comparison between the execution times of code with and without optimized bound checks to show the impact of check removal.

Out-of-bound errors occur in 4 benchmarks: *mdg, spc77, trfd* from PerfectClub and *linpackd* from Riceps. These errors are caused by violating standard declaration of array in Fortran and there is no possibility to obtain comparable results for these programs.

Compilation times of Nascent for the ten benchmarks are much shorter than that of PIPS. Table 7 shows the percentage of dynamic checks removed for the five usable benchmarks. Our two techniques perform as well as the Nascent specific techniques.

6.4 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.

Based on result analysis, we can see 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* (see Table 3) 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% (see Figure 7). The elimination percentage goes up to 100% from 97.09% for *applu* (see Table 3) by adding one STOP statement after the parameter checking that is done for lower bound test but seems forgotten 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 measured on SUN (see Figure 7).

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 [20, 18] 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 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 3 and Figure 7). So after array bound checking optimization, it is necessary to also compare the execution times of generated codes which is missing in [14].

7. CONCLUSION

We have measured execution and compilation times for the standard SPEC CFP95 benchmarks with three different compilers and with two experimental algorithms based on two intra- and inter-procedural automatic parallelization analyses. On average, we obtained slowdowns smaller than 19.29% for Elimination of Redundant Tests and 5.33% for Insertion of Unavoidable Tests on SUN with respect to the execution times with no array bound checks. These numbers are 7.05% and 0.61% on SGI, 16.59% and 10.62% on IBM. With the SGI compiler we mostly obtained speedups by adding array bound checks for the Insertion of Unavoidable Tests version.

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 adn SGI compilers are large enough to make improvement easy. 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 [14] 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 parallelization features 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,...

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.

8. ACKNOWLEDGEMENTS

We would like to thank Béatrice Creusillet for implementing the MAY-MUST array region analysis, Fabien Coelho for speeding up some key algorithms in PIPS as well as Pierre Jouvelot and Georges-André Silber for their helpful comments. We also wish to give special thanks to Serge Algarotti and Catherine Mongenet for letting us use their IBM and SGI machines.

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

[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 on Compiler Construction*, 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*, 22(2):265–295, March 2000.

[19] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 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 *Programming Language Design and Implementation 2000*, pages 182–195, Vancouver, British Columbia, Canada, 2000.

[22] 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.

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

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

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

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

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

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