Elimination of Redundant Array Subscript Range Checks *

Priyadarshan Kolte and Michael Wolfe

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
{pkolte, mwolfe}@cse.ogi.edu

Abstract

This paper presents a compiler optimization algorithm to reduce the run time overhead of array subscript range checks in programs without compromising safety. The algorithm is based on partial redundancy elimination and it incorporates previously developed algorithms for range check optimization. We implemented the algorithm in our research compiler, Nascent, and conducted experiments on a suite of 10 benchmark programs to obtain four results: (1) the execution overhead of naive range checking is high enough to merit optimization, (2) there are substantial differences between various optimizations, (3) loop-based optimizations that hoist checks out of loops are effective in eliminating about 98% of the range checks, and (4) more sophisticated analysis and optimization algorithms produce very marginal benefits.

1 Introduction

Program statements that access elements of an array outside the declared array ranges introduce errors which can be difficult to detect. Since compile-time checking of whether all array accesses in a program are within the declared ranges is not possible in general, many compilers offer the option of inserting run-time range checks into the compiled program so that errors due to array range violations are detected during execution. A range check compares the array subscript expression with the array bounds and then traps (or raises an exception) if the subscript is not within the declared array range. In spite of the advantage of compiler-inserted range checks in improving the reliability of programs, programmers often prefer to compile their programs without range checking because the execution overhead of the range checks is too high. Hence, compiler optimizations that reduce the execution overhead of range checks without compromising safety are useful. Range check optimization is especially beneficial in safety-oriented languages such as Ada, where range checking is not just a compiler option, but is required by the language definition.

Although range checks are subject to traditional compiler optimizations such as constant propagation, common subexpression elimination, and invariant code motion, range checks possess an interesting property that we study in this project: *a range check C_i may "imply" a check C_j, and therefore performing C_i makes performing C_j unnecessary.* For example, Figure 1(a) shows a program with two statements, S_1 and S_2, and four range checks, $C_1 \ldots C_4$. Check C_2 implies check C_4 because of the mathematical fact $(2 * N \leq 10) \Rightarrow (2*N-1 \leq 10)$. Thus, check C_4 is redundant because a "stronger" check, C_2, has "already been performed." The redundant check is eliminated and the optimized program has only three range checks as shown in Figure 1(b). Further optimization exploits the fact $(2 * N - 1 \geq 5) \Rightarrow (2 * N \geq 5)$ At the program point where check C_1 is performed, the compiler deduces that a stronger check, C_3, is "guaranteed to be performed in the future." Hence the compiler places the stronger check, C_3, before the weaker check, C_1, which makes check C_1 redundant. The resulting program has only two range checks and is shown in Figure 1(c).

Researchers have studied range check optimization as applications of automated program verification [8, 17], abstract interpretation [4, 5, 11], and data flow analysis [2, 9, 10]. Range check optimizers have also been implemented in several compilers such as the IBM PL.8 compiler [13], the Alsys Ada compiler [14], and the Karlsruhe Ada compiler [16].

This project builds on previous work, especially on the algorithms presented by Gupta [9, 10]; our main contributions are:

- use of partial redundancy elimination techniques [15, 12] for range check optimization;
- a study of the advantages of using induction variables [7, 18] in range check optimization;
- an implementation of a range check optimizer in our research Fortran compiler, Nascent; and
- an experimental evaluation of the compile time cost and effectiveness of various optimizations on a suite of large programs.

We present background and notation in §2. §3 describes our formulation and solution of the range check optimization problem. §4 presents the results of our experiments. §5 discusses related work, and §6 concludes.

*This work is supported by NSF Grant CCR-9113885 and grants from Intel Supercomputer Systems Division and the Oregon Advanced Computing Institute.

```
    integer A[5..10]
    ...
C1: if (not (2*N ≥ 5)) TRAP
C2: if (not (2*N ≤ 10)) TRAP
S1: A[2*N] = 0
C3: if (not (2*N-1 ≥ 5)) TRAP
C4: if (not (2*N-1 ≤ 10)) TRAP
S2: A[2*N-1] = 1
    ...
```

(a) Without any optimization

```
    integer A[5..10]
    ...
C1: if (not (2*N ≥ 5)) TRAP
C2: if (not (2*N ≤ 10)) TRAP
S1: A[2*N] = 0
C3: if (not (2*N-1 ≥ 5)) TRAP
S2: A[2*N-1] = 1
    ...
```

(b) With some optimization

```
    integer A[5..10]
    ...
C3: if (not (2*N-1 ≥ 5)) TRAP
C2: if (not (2*N ≤ 10)) TRAP
S1: A[2*N] = 0
S2: A[2*N-1] = 1
    ...
```

(c) With more optimization

Figure 1: Example of a program fragment without and with range check optimizations

2 Background and notation

We assume that the reader is familiar with the control flow graph (CFG) abstraction of programs, natural loops, data flow analysis [1], and the Static Single Assignment (SSA) representation of programs [6].

2.1 Partial redundancy elimination

Partial redundancy elimination (PRE) eliminates redundant computation of expressions in programs by moving invariant computations out of loops and also eliminating identical computations that are performed more than once on any execution path. Although we use terms defined by Morel and Renvoise [15] in this paper, our implementation of PRE uses the *safe-earliest* and *latest-not-isolated* transformations developed by Knoop, Rüthing, and Steffen [12] because these transformations are conceptually simpler and more efficient than the techniques originally proposed by Morel and Renvoise. The general strategy of PRE transformations of programs is as follows.

1. **Identify sets of equivalent expressions.** Expressions in the program are partitioned into equivalence classes. Usually, syntactic equivalence is used to determine if two expressions are equivalent.

2. **Solve data flow systems for availability and anticipatability of expressions.** An expression e is *available* at a program point p if some expression in the equivalence class of e, say e', has been computed on every path from the entry of the CFG to p and none of the operands of e have been redefined since the computation of e'. Intuitively, an expression e is available at point p if the value of e has always been computed when program execution reaches point p. Forward data flow analysis is used to determine availability of expressions in programs.

   An expression e is *anticipatable* at a program point p if e is computed on every path from p to the exit of the CFG before any of the operands of e are redefined. Intuitively, an expression e is anticipatable at point p if e is always guaranteed to be evaluated at some point after the execution of program point p. Backward data flow analysis is used to determine anticipatability of expressions in programs.

3. **Determine points in the program where new computations of expressions can be safely and profitably inserted.** It is *safe* to insert an expression at a program point only if inserting the expression does not change the behavior of the program. The principal observation is that if an expression e is anticipatable at a program point p, then it is safe to insert a computation of e at point p. It is *profitable* to insert a computation of expression e at program point p if e is not already available at point p and if the insertion of e at p makes some other computations of e (on the program execution path after point p) redundant.

4. **Eliminate redundant computations.** The computation of an expression e at program point p is *redundant* if e is available at p. A redundant computation is replaced by a copy of the result of the computation that is available.

The safe-earliest and latest-not-isolated transformations are two techniques for PRE. They differ from each other in the placement of new computations that are inserted in step 3 above: the safe-earliest strategy is to place computations as early in the program execution path as possible, whereas the latest-not-isolated strategy places computations as late as possible to minimize register pressure.

2.2 Canonical form of range checks

Range checks of the form (if (not (subscript-expression ≤ upper-bound-expression)) then TRAP) are expressed in the canonical form: (Check (range-expression ≤ range-constant)), where *range-expression* contains all the symbolic terms of the check and all the constants in the check are folded into the *range-constant* term; e.g., the check (if (not (i+1 ≤ 4*N)) then TRAP) is expressed as (Check (i−4*N ≤ −1)). The symbolic terms in range-expression are also listed in a canonical order whenever rearrangement of the terms is possible. Lower bound checks are expressed in canonical form after negating both sides of the inequality; e.g., the check (if (not (i+1 ≥ 4)) then TRAP) is expressed as (Check (−i ≤ −3)).

The canonical form is used so that semantically equivalent range checks (which may be syntactically different) fall into the same equivalence class in step 1 of the PRE algorithm. Intuition suggests that larger equivalence classes (fewer classes) usually increase the number of redundant expressions. Thus, the intent and effect of the canonical form

Program fragment	SSA representation
j = 0	$j_0 = 0$
k = 3	$k_0 = 3$
m = 5	m = 5
for i = 0 to n–1 do	for i = 0 to n – 1 do
	$j_1 = \phi\ (j_0, j_2)$
	$k_1 = \phi\ (k_0, k_2)$
j = j + 1	$j_2 = j_1 + i$
k = k + m	$k_2 = k_1 + m$
A [k] = 2 * m + 1	A [k_2] = 2 * m + 1
endfor	endfor

PRX	INX	Classification
i	h	Linear
j_2	h*(h+1)/2	Polynomial
k_2	5*h+8	Linear
2 * m + 1	11	Invariant

Figure 2: Example of induction variable analysis in the Nascent compiler

of range checks is similar to the global reassociation technique described by Briggs and Cooper [3]. The canonical form also simplifies the representation and manipulation of range checks within the compiler. We use the canonical form to denote range check statements in the rest of the paper.

2.3 Range checks using induction expressions

The Nascent compiler uses SSA-based induction variable analysis techniques to associate induction expressions with all expressions in a program [7, 18]. Each loop in the program is assigned a basic loop variable which assumes values 0,1,... for every loop iteration; each expression in the loop is associated with an induction expression which is a function of the basic loop variable. Induction expressions are classified as invariant, linear, polynomial, etc., depending on their complexity. When possible, the trip counts of natural loops are also determined by the analysis.

Figure 2 shows an example[1] of induction variable analysis on a loop that has been assigned the basic loop variable h. Induction variable analysis determines the induction expressions for all the program expressions; e.g., the program expression k_2 is associated with the induction expression 5*h+8. In the table in Figure 2, the column labeled "PRX" shows some of the program expressions, the column labeled "INX" shows induction expressions associated with the program expressions, and the last column shows the classifications of the induction expressions. Induction variable analysis also determines that the trip count of the loop is max(0,n).

Range checks are created from either program expressions (using the abstract syntax tree of the program) or from induction expressions (using induction variable analysis). When we need to distinguish between the two kinds of checks, we use the abbreviation *PRX-Check* to mean a check that is created from program expressions, and *INX-Check* to mean a check that is created from induction expressions. For example, if the upper bound of the array A in the program in Figure 2 is 100, then we can use either PRX-Check ($k_2 \leq 100$) or INX-Check ($5*h \leq 92$) as the upper bound check for A[k].

[1] For clarity in this example, the loop index variable, i, is not shown in SSA form, and the loop is not decomposed into test and branch constructs.

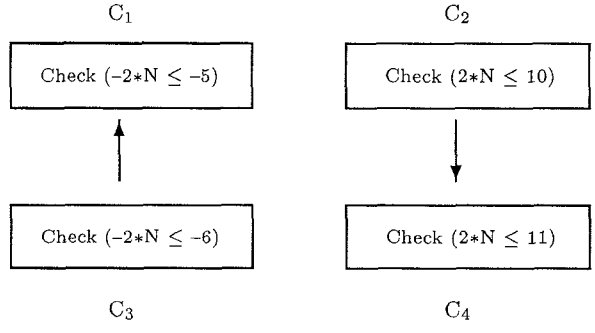

Figure 3: Check Implication Graph (CIG) of the program fragment shown in Figure 1(a)

3 Range check optimization

The goal of range check optimization is to reduce the execution overhead of range checking without affecting the behavior of programs. A range check optimization preserves the behavior of a program if: (1) an array range violation is detected in the optimized program if and only if the array range violation is detected in the unoptimized program and (2) a range violation in the optimized program is detected at compile-time or at run-time at a program execution point no later than the execution point at which the violation in the unoptimized program is detected.
Our solution to the range check optimization problem takes the program with range checks as input and optimizes it in five steps:

1. **Construct the check implication graph.** A *Check Implication Graph (CIG)* is a directed graph in which the nodes represent range checks (all equivalent checks share a node). There is an edge from node C_i to node C_j in the CIG only if check C_i implies check C_j. Figure 3 shows an example CIG.

   The CIG is required for computing the "as strong as" relation on checks: check C_i is *as strong as* check C_j if there is a path (possibly a trivial path) from node C_i to node C_j in the CIG.

2. **Compute safe insertion points for checks.** If checks that are as strong as check C are anticipatable at a program point, then it is safe to insert check C at the program point. Computing anticipatable checks requires backward data flow analysis.

3. **Insert checks at safe and profitable program points.** Usually it is safe to insert a check at multiple points in the program, and multiple checks can be inserted at each program point. Hence, the compiler has to determine the "optimal" checks to be inserted at the "optimal" program points. There are at least five different schemes for inserting these checks, which we describe in section 3.3.

4. **Compute available checks and eliminate redundant checks.** If checks that are as strong as check C are available at the program point where C occurs, then C is redundant and is eliminated from the program. Computing available checks requires forward data flow analysis.

5. **Eliminate compile time checks.** Checks in the CIG that contain only compile time constants are evaluated in this step. Compile-time checks that evaluate to false are replaced by TRAP instructions and are reported to the programmer; checks that evaluate to true are eliminated from the program.

The following subsections explain the parts of our solution in more detail.

3.1 Construction of the check implication graph

A *family* is a set of range checks which have the same range-expression; e.g., Check (i+j $\leq$ 10) and Check (i+j $\leq$ 11) are in the same family. Within each family, the list of range checks is ordered in increasing order of the range-constants of the checks. Thus, if C_i and C_j belong to the same family and C_i appears earlier in the list than C_j, then C_i is stronger than C_j.

We optimize the construction of the CIG by using families of checks as nodes of the CIG instead of range checks. For example, the CIG shown in Figure 3 after optimization consists of only two nodes: F_1 and F_2, where F_1 contains checks $\{C_3, C_1\}$, and F_2 contains checks $\{C_2, C_4\}$.

In addition to simplifying the detection of implications between checks in the same family, using families as nodes of the CIG helps discover implications between checks in different families. Whenever we discover an implication between two checks, say $C_{i'}$ (in family F_I) and $C_{j'}$ (in family F_J), we add an edge from node F_I to node F_J and give edge (F_I, F_J) a weight equal to range-constant($C_{j'}$) − range-constant($C_{i'}$). Then, for any two checks C_i in family F_I and C_j in family F_J, if range-constant(C_i) + weight(F_I, F_J) $\leq$ range-constant(C_j), then we know that check C_i is as strong as check C_j.

For example, consider families F_3 and F_4 shown in Figure 4. If we find that Check (n $\leq$ 6) $\Rightarrow$ Check (m $\leq$ 10), we add an edge from F_3 to F_4 with weight 4. Using this edge, it is simple to infer that Check (n $\leq$ 1) is as strong as Check (m $\leq$ 7), but we cannot infer that Check (n $\leq$ 1) is as strong as Check (m $\leq$ 3).

If an edge E_i has to be added between nodes in the CIG which already has edge E_j between them, the weight of E_j is modified to be the minimum of the weights of E_i and E_j.

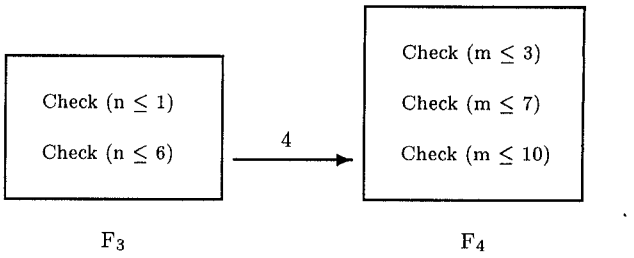

Figure 4: Example of CIG with families as nodes

3.2 Computing available and anticipatable checks

Computing available checks is a forward data flow problem whereas computing anticipatable checks is a backward data flow problem. For both data flow problems, a check is killed by a definition of any of the symbols in its range-expression.

For computing availability, a range check statement generates a check C as well as all weaker checks (i.e., checks C′ such that C is as strong as C′). Thus, at merge nodes in the CFG, a check C is available after the merge if checks C_1 and C_2 are available on the inputs to the merge such that C_1 is as strong as C and C_2 is as strong as C.

For anticipatability, we use similar but stronger conditions for determining the set of checks generated and those anticipatable before a branch. A range check statement generates a check C and all weaker checks that are in the family of C. Thus, at branch nodes in the CFG, a check C is anticipatable before the branch if checks C_1 and C_2 are anticipatable after the branch such that C_1 is as strong as C, C_2 is as strong as C, and C_1, C_2, and C are members of the same family. Since anticipatability determines where checks can be inserted, the restriction that check implications are only within families ensures that a check is not inserted before the definition of one of the symbols in its range-expression.

3.3 Insertion of checks at safe and profitable program points

This section presents five schemes for selecting program points for placing new checks: no-insertion, safe-earliest and latest-not-isolated placement, check-strengthening, and preheader insertion.

No-insertion is the simplest scheme in which steps 2 and 3 of our solution are skipped and no checks are inserted in the program. The only redundant checks are due to availability and compile-time constants (in steps 4 and 5).

The **safe-earliest** and **latest-not-isolated** transformations, developed by Knoop et al. for PRE of arithmetic expressions [12], can be applied to the problem of range check placement if the anticipatable and available checks are computed as described in the previous subsection. The safe-earliest placement is preferred to the latest-not-isolated placement because, unlike computation of arithmetic expressions, computing a check does not define any variable, and hence performing a check early has no effect on register pressure. Furthermore, performing a check as early as possible increases the number of program points where that check becomes available, which in turn increases the number of other checks that become redundant.

Although the safe-earliest transformation always provides profitable placements for arithmetic expressions, when it is applied to the problem of determining placements for range

```
integer A[1..10]

i = ...

if ( ..)
   Check (i ≤ 10)
   ...A[i]...
else
   Check (i ≤ 6)
   ...A[i+4]...
endif
```

(a) Original program fragment

```
integer A[1..10]

i =  .
Check (i ≤ 10)
if (...)
   Check (i ≤ 10)
   ...A[i]...
else
   Check (i ≤ 6)
   ...A[i+4]...
endif
```

(b) After safe-earliest placement

```
integer A[1..10]

i = ...
Check (i ≤ 10)
if (...)

   ...A[i]...
else
   Check (i ≤ 6)
   ...A[i+4]...
endif
```

(c) After redundancy elimination

Figure 5: Example of a program where safe-earliest placement is not always profitable

```
integer A[1..10]

do j = 1 to 2*n
   Check (k ≤ 10)
   ...A[k]...
   Check (j ≤ 10)
   ...A[j]...
enddo
```

(a) Original program fragment

```
integer A[1..10]

Cond-check ((1 ≤ 2*n), k ≤ 10)
Cond-check ((1 ≤ 2*n), 2*n ≤ 10)

do j = 1 to 2*n
   Check (k ≤ 10)
   ...A[k]...
   Check (j ≤ 10)
   ...A[j]...
enddo
```

(b) After preheader insertion

```
integer A[1..10]

Cond-check ((1 ≤ 2*n), k ≤ 10)
Cond-check ((1 ≤ 2*n), 2*n ≤ 10)

do j = 1 to 2*n

   ...A[k]...

   ...A[j]...
enddo
```

(c) After redundancy elimination

Figure 6: Example of a program with preheader insertion

checks (using our anticipatability algorithm for determining safe program points), it is not guaranteed to produce profitable transformations. For example, the transformation of the program in Figure 5(a) to that in 5(c) increases the number of checks performed on the path along the else branch.

Check-strengthening, which was proposed by Gupta [9, 10], considers program points just before range checks in the CFG for inserting new checks. For each check C in the program, check-strengthening computes the strongest anticipatable check C′ that implies the check C, inserts C′ just before C, and eliminates C (the actual mechanism is to replace C by C′). The optimization of the program in Figure 1(b) to the one in Figure 1(c) is an example of strengthening. Check strengthening can be viewed as a conservative form of the safe-earliest placement which misses some of the optimizations, but which avoids the profitability problem shown in Figure 5.

Preheader insertion is a simple and effective scheme for hoisting checks out of loops. If check C is anticipatable at the beginning of the loop body, and if the range-expression of check C is either invariant or linear in the index variable of the loop, then a conditional check, C′, is inserted in the preheader of the loop; the check C′ is conditional on the loop executing at least once. When the condition can be evaluated at compile time, an ordinary check is inserted instead of a conditional check. All loops in the program are processed in an inner loop to outer loop manner so that checks from inner loops are hoisted to the outermost loop possible.

Figure 6 shows an example of check placement performed by preheader insertion. Since Check (k ≤ 10) is **loop-invariant**, it is hoisted out of the loop as Cond-check ((1 ≤ 2*n), k ≤ 10); this conditional check means: if (1 ≤ 2*n) evaluates to true, then perform Check (k ≤ 10). Since Check (j ≤ 10) is linear in the loop index variable, we perform **loop-limit substitution** of the index variable, j, to get Check (2*n ≤ 10), which is then hoisted out of the loop as the conditional check Cond-check ((1 ≤ 2*n), 2*n ≤ 10). Figure 6(b) shows the conditional checks that are inserted in the preheader of the loop and Figure 6(c) shows the redundant checks eliminated from the loop.

Preheader insertion is more effective in hoisting checks out of loops than the safe-earliest placement for two reasons: (1) safe-earliest placement does not consider conditional checks and (2) even when the check to be hoisted out of a loop is not conditional (i.e., it is known at compile time that the loop executes at least once), the control flow structure of while loops prevents the check from being anticipatable at the loop preheader. (A CFG transformation such as *loop rotation* can help the safe-earliest placement in such cases by converting while loops into repeat loops.)

3.4 Implementation in Nascent

We have implemented the algorithm described in the previous subsections in our research Fortran compiler, Nascent.

The range check optimizer offers six choices for inserting

new checks at safe program points: no insertion of checks, check strengthening only, safe-earliest placement of checks, latest-not-isolated placement of checks, insertion of only loop invariant checks in preheaders of loops, and insertion of all checks that are linear (which includes loop invariant checks) in preheaders of loops. These options permit us to compare the various check placement schemes.

The range check optimizer supports three kinds of implications between checks: no implications between checks, implications between checks in different families only, and all implications between checks (within and across families). We use these options to investigate the importance of the implication property of checks for the optimizations.

The range check optimizer can create program expression checks (PRX-checks) from the abstract syntax tree representation of programs as well as induction expression checks (INX-checks) from the program representation produced by the induction variable analysis phase of Nascent. This option allows us to study the effects of using induction variable analysis on range check optimization.

4 Experimental results

This section presents experiments that answer the following questions about range check optimization:

1. Is range check optimization really needed?
2. What is the effectiveness and cost of optimization?
3. Does induction variable analysis help?
4. Is the check implication property important?

We use the dynamic counts of instructions as the measure of the execution times of programs. The C back-end of Nascent translates Fortran programs into instrumented C programs which are then compiled and executed using their standard input data sets to obtain the dynamic counts of instructions.

For our suite of test programs, we chose 10 scientific programs from the Perfect, Riceps, and Mendez benchmarks because these are known to contain substantial array-based computation [7]. The particular 10 programs chosen for this study satisfied two criteria: (1) the C back-end of Nascent is not mature and these programs required no manual editing to "fix" the C programs, and (2) they had moderate disk space and computation time requirements.

The first five columns of Table 1 show the names of the programs and counts of source lines, subroutines, and natural loops. The two columns labeled "instructions" show the static and dynamic counts of instructions without range checking in the benchmark programs. Since the benchmark programs were naively translated without any optimizations, the instruction counts represent the upper limit on the number of non-range-check instructions.

4.1 Is range check optimization really needed?

The first experiment studies programs to determine whether range check optimization is necessary. In Table 1, the two columns labeled "range checks" show the static and dynamic counts of range checks needed for unoptimized range checking of the programs. The final two columns of Table 1 show the ratios of the counts of range checks to the counts of all other instructions. Since the minimum ratio of the dynamic counts is 22%, the maximum ratio is 66%, and we expect one range check to translate into at least two instructions, we estimate that the overhead of executing range checks without any optimization is between 44% and 132%. This is a pessimistic estimate of the overhead of range checking because the programs are not optimized and we overestimated the number of non-range-check instructions. We conclude that the execution overhead of the range checks is high enough to need optimization.

4.2 What is the effectiveness and cost of optimization?

The second set of experiments measure the effectiveness and compile time cost of seven check placement schemes on two kinds of checks: checks constructed from program expressions (PRX-Checks) and checks constructed from induction expressions (INX-Checks). The seven check placement schemes are:

1. NI: redundancy elimination without any insertion of checks,
2. CS: check strengthening only,
3. LNI: latest-not-isolated placement,
4. SE: safe-earliest placement,
5. LI: preheader insertion of only loop invariant checks,
6. LLS: preheader insertion with loop-limit substitution of linear checks, and
7. ALL: loop-limit substitution followed by safe-earliest placement.

The columns of Table 2 show the percentage of checks eliminated by the various optimizations; these percentages are with respect to the dynamic counts of range checks shown in Table 1. The last two columns show the compilation times for the 10 programs, which consist of a total of 26,307 lines of Fortran source code. These times were obtained by compiling Nascent using the -O2 option of g++ and executing Nascent on a Sun SPARCcenter 2000. The penultimate column, which is labeled "Range", shows the CPU time (in seconds) taken by the range check optimization phase, and the final column (labeled "Nascent") shows the wall clock time (in minutes and seconds) required by Nascent to parse, optimize, and generate C code for the 10 programs. The range check optimization phase takes a significant fraction of the compilation time (as much as computing SSA and performing induction variable analysis) because we did not fine tune the implementation for speed; a more sophisticated implementation might halve the time required for range check optimization.

When we compare the percentage of PRX-Checks eliminated shown in the rows in Table 2, we find that check strengthening (CS) is marginally better than no check insertion (NI). The number of checks eliminated by the PRE check placement schemes (LNI and SE) are also very close to those eliminated by ordinary redundancy elimination (NI); the maximum improvement is 7% for dyfesm. As expected, safe-earliest placement (SE) eliminates more checks than the latest-not-isolated placement (LNI), but not by a lot — the maximum difference is 2.9% for spec77. Both preheader placement schemes (LI and LLS) eliminate a much larger number of checks than the other placement schemes. As expected, placing only loop-invariant checks in loop preheaders (LI) is not as good as performing loop-limit substitution of linear checks (LLS), and LLS eliminates almost 30% more

suite	program	lines	subr	loops	instructions static	instructions dynamic	range checks static	range checks dynamic	check/instr (%) static	check/instr (%) dynamic
Mendez	vortex	710	20	35	2,148	$3{,}694\times10^6$	672	950×10^6	31	26
Perfect	arc2d	3,964	39	234	16,050	$15{,}288\times10^6$	7,810	$10{,}156\times10^6$	48	**66**
	bdna	3,980	42	276	16,236	$3{,}712\times10^6$	4,177	803×10^6	25	**22**
	dyfesm	7,608	77	269	7,039	$2{,}724\times10^6$	2,765	$1{,}543\times10^6$	39	57
	mdg	1,238	16	56	4,471	$13{,}653\times10^6$	1,176	$6{,}344\times10^6$	26	46
	qcd	2,327	35	168	6,801	$1{,}649\times10^6$	2,652	788×10^6	38	48
	spec77	3,885	64	413	15,225	$14{,}920\times10^6$	5,538	$7{,}124\times10^6$	36	48
	trfd	485	7	79	2,052	$3{,}939\times10^6$	292	$2{,}332\times10^6$	14	59
Riceps	linpackd	797	11	41	1,738	135×10^6	530	61×10^6	30	45
	simple	1,313	8	75	5,615	$43{,}545\times10^6$	2,738	$26{,}255\times10^6$	48	60

Table 1: **Program characteristics of benchmark programs**

		vortex	arc2d	bdna	dyfesm	mdg	qcd	spec77	trfd	linpackd	simple	Range	Nascent
PRX-Checks	NI	89.88	84.60	90.85	69.96	79.70	78.75	81.61	61.01	65.90	92.25	6.8	1:18
	CS	89.89	85.64	90.87	69.96	80.10	78.79	84.57	61.01	65.90	93.94	15.9	1:27
	LNI	89.88	84.61	90.85	76.82	79.70	78.75	84.44	61.01	65.90	92.26	18.4	1:29
	SE	89.89	85.64	90.87	76.83	80.10	78.79	87.35	61.01	65.91	93.95	16.7	1:28
	LI	89.88	84.60	90.85	69.96	79.70	78.75	81.61	61.01	65.90	92.25	11.0	1:22
	LLS	99.99	99.96	98.44	98.74	98.53	97.00	96.67	98.74	99.73	99.97	13.0	1:24
	ALL	99.99	99.96	98.44	98.73	98.54	97.06	98.24	98.74	99.73	99.97	24.0	1:35
INX-Checks	NI	89.88	86.08	87.62	69.87	78.62	78.28	81.59	60.95	65.47	92.22	11.5	1:22
	CS	89.89	87.63	87.64	69.87	79.02	78.28	84.54	60.95	65.47	93.92	23.4	1:36
	LNI	89.88	86.08	87.63	76.73	78.62	78.28	84.38	60.95	65.47	92.24	27.1	1:37
	SE	89.89	87.63	87.64	76.73	79.02	78.50	87.32	60.86	65.47	93.93	24.9	1:36
	LI	89.88	94.21	90.87	77.99	78.83	85.19	87.53	81.77	67.63	95.96	17.9	1:29
	LLS	99.99	98.96	95.81	98.17	97.65	96.57	99.70	98.66	99.72	99.96	21.1	1:32
	ALL	99.99	98.96	95.81	98.16	97.65	96.80	99.71	98.57	99.73	99.96	36.7	1:47

Table 2: **Percentage of checks eliminated by optimizations and time required for compilation.** NI = redundancy elimination with no insertion of checks, CS = check strengthening, LNI = latest-not-isolated placement, SE = safe-earliest placement, LI = preheader placement of only loop invariant checks, LLS = preheader placement with loop-limit substitution of linear checks, ALL = LLS followed by SE. Range = CPU time (in seconds) required by range optimization, Nascent = wall clock time (in minutes and seconds) required by Nascent for all 10 programs.

checks than LI in the program dyfesm. Further optimization such as loop-limit substitution followed by the safe-earliest placement (ALL) provides a negligible improvement in the number of checks eliminated.

Comparing the times required for various range check optimizations shows that no insertion (NI) is fastest (obviously!), the preheader insertion schemes (LI and LLS) are moderately expensive, and the PRE-based schemes (CS, LNI, SE) are the slowest. As expected, within the PRE-based placement schemes, check strengthening (CS) is faster than safe-earliest placement (SE), which is faster than the latest-not-isolated placement (LNI). Similarly, within the preheader insertion schemes, LI is slightly faster than LLS. Based on the number of checks eliminated and the compile time required, preheader insertion with loop-limit substitution of linear checks (LLS) is the clear winner among all the check placement schemes

4.3 Does induction variable analysis help optimization?

Table 2 shows a surprising result when we compare the number of PRX-Checks eliminated with the corresponding number of INX-Checks: there are a number of cases such as bdna where a few more PRX-checks were eliminated than the INX-checks. This result is unexpected because induction expressions should hold at least as much semantic information as the program expressions. One explanation is a particular weakness in the implementation of induction variable analysis in Nascent that we have not repaired yet. Even so, range check optimization of INX-checks is never very bad compared to PRX-checks, and there is one case, LI optimization of trfd, where about 20% more checks were eliminated due to induction variable analysis; in this case, induction variable analysis could detect more loop invariant checks. Until we repair the implementation of induction variable analysis in Nascent and further experiments indicate otherwise, we conclude that using induction variable analysis does not help range check optimization.

4.4 Does implication between checks help optimization?

The third experiment measures the effectiveness of the check implication property in detecting and eliminating redundant checks. We used variations of the check implication graph to produce another three check placement schemes:

1. NI′: redundancy elimination without any insertion of checks and with no implications between checks,
2. SE′: safe-earliest placement with no implications between checks, and
3. LLS′: preheader insertion with loop-limit substitution of linear checks with no implications between checks in the same family, but with implications between checks

		vortex	arc2d	bdna	dyfesm	mdg	qcd	spec77	trfd	linpackd	simple	Range	Nascent
PRX-Checks	NI	89.88	84.60	90.85	69.96	79.70	78.75	81.61	61.01	65.90	92.25	6.8	1:18
	NI′	89.87	82.94	88.85	69.93	79.29	78.72	76.99	61.01	65.88	90.55	8.7	1:20
	SE	89.89	85.64	90.87	76.83	80.10	78.79	87.35	61.01	65.91	93.95	16.7	1:28
	SE′	89.87	82.94	88.85	76.79	79.29	78.72	79.79	61.01	65.88	90.57	18.9	1:30
	LLS	99.99	99.96	98.44	98.74	98.53	97.00	96.67	98.74	99.73	99.97	13.0	1:24
	LLS′	99.99	99.96	98.32	98.68	95.19	96.84	92.30	98.74	99.73	99.58	12.9	1:23
INX-Checks	NI	89.88	86.08	87.62	69.87	78.62	78.28	81.59	60.95	65.47	92.22	11.5	1:22
	NI′	89.87	84.05	85.75	69.84	78.23	78.25	76.96	60.95	65.44	90.52	14.9	1:25
	SE	89.89	87.63	87.64	76.73	79.02	78.50	87.32	60.86	65.47	93.93	24.9	1:36
	SE′	89.87	84.05	85.75	76.70	78.23	78.48	79.74	60.86	65.44	90.55	30.5	1:41
	LLS	99.99	98.96	95.81	98.17	97.65	96.57	99.70	98.66	99.72	99.96	21.1	1:32
	LLS′	99.99	98.96	95.81	98.11	97.65	96.54	99.64	98.66	99.72	99.95	22.0	1:32

Table 3: **Percentage of checks eliminated by optimizations with and without implications between checks, and time required for compilation.** NI = redundancy elimination with no insertion of checks, NI′ = NI with no implications between checks, SE = safe-earliest placement, SE′ = SE with no implications between checks, LLS = preheader placement with loop-limit substitution of linear checks, LLS′ = LLS with no implications between checks within the same family. Range = CPU time (in seconds) required by range optimization, Nascent = wall clock time (in minutes and seconds) required by Nascent for all 10 programs.

in different families; this maintains implications from conditional checks inserted in loop preheaders to the corresponding checks in the loop bodies.

Table 3 shows the results of these three check placement schemes on the two kinds of checks. When we examine the number of checks eliminated without using check implications, we find a marginal decrease ($\leq$ 3%) in almost all cases. There is only one program, spec77, where 7% fewer checks were eliminated without use of the check implication property!

Why do optimizations that use implication (NI and SE) take less time than optimizations that do not use implication (NI′ and SE′)? We did not modify the implementations of the anticipatability and availability data flow algorithms to take advantage of the fact that there are no implications between checks in NI′ and SE′; these implementations search the check implication graph for implied checks. Hence, in the case of NI′ and SE′, where no check implies any other, each check is inserted in a separate family, and the increase in the number of nodes of the CIG increases the time required by the data flow algorithms, which increases the time required for range check optimization.

The results of this experiment indicate that the property of implication between checks is not very important for range check optimization — the only important implications are those from checks inserted in loop preheaders to the corresponding checks in the loop bodies.

5 Related work

Related work on range check optimization can be partitioned into two groups. The first group concentrates on the problem of identifying range checks which can be evaluated at compile-time and eliminated from the program; this includes the automated program verification approach [8, 17] and the abstract interpretation approach [4, 5, 11, 14, 16]. The second group aims to reduce the execution overhead of range checks which cannot be evaluated and eliminated at compile-time; this includes algorithms that perform data flow analysis and insertion of checks [2, 9, 10, 13]. Our algorithms are in the second group.

Suzuki and Ishihata [17] and German [8] used Floyd-Hoare logics and theorem proving techniques to verify the absence of array range violations in programs. Two limitations of the program verification approach are that it often requires the programmer to supply assertions to aid the verification proofs and that it is restricted to programs written in a structured manner (without goto statements). Hence, we feel that this approach is not directly applicable to the problem of automatic range check optimization of arbitrary programs.

The abstract interpretation algorithms [4, 5, 11, 14, 16] perform generation, propagation, and combination of assertions about the bounds of variables to determine compile-time checks. The different algorithms vary in the sophistication of the rules used for propagation and combination of the assertions: the rules implemented in the Karlsruhe Ada compiler [16] seem the simplest (and probably are the fastest) and those proposed by Cousot and Halbwachs [5] are quite complex. Since the algorithms in the abstract interpretation approach and the program verification approach do not perform any insertion of checks in the program (steps 2 and 3 of our solution), they take advantage only of completely redundant checks and they miss opportunities for exploiting partially redundant checks. The main weakness of these algorithms is that they do not attempt to reduce the run time overhead of checks which cannot be evaluated at compile time. Hence we expect the number of checks eliminated by these algorithms to be less than algorithms which insert checks.

It seems curious that both the implementations of Ada compilers [14, 16] use partial redundancy elimination for the optimization of most program expressions, but not for optimizing range checks. Perhaps these compilers are conservative due to the Ada language requirement that the compiler is not permitted to move a computation to a program point which might change the exception handler that is invoked in case an exception occurs in the computation.

Markstein, Cocke, and Markstein [13] presented the first paper that addresses the problem of reducing the execution overhead of range-checks. They described an algorithm that is like a restricted form of preheader check insertion; the only checks that it considers for preheader insertion are the checks present in articulation nodes in the loop body (because these nodes post-dominate the loop entry nodes and dominate the loop exit nodes) and which have simple range expressions. More recent approaches [9, 10] and our algo-

rithms handle checks with more complex range expressions and use data flow analysis to relax the restriction about checks in articulation nodes. In light of our experimental results, which show that the additional sophistication may not be cost effective, it would be interesting to implement the Markstein et al. algorithm in Nascent to compare its effectiveness with the loop-limit substitution algorithm.

The range check optimization algorithm presented in this paper is based on work by Gupta [9, 10]. We use a partial redundancy elimination framework to implement the data flow analysis and optimizations described by Gupta. It is not clear how Gupta represents implications between checks in the compiler; we have proposed check implication graphs to denote and manipulate implications between checks. In contrast to Gupta's rules for determining whether loop-limit substitution is applicable to a check within a loop, we use induction expressions and induction type classifications (invariant and linear) produced by Nascent. Our experimental results match the limited results presented by Gupta.

Asuru [2] also extends Gupta's range check optimization algorithms. Our loop-limit substitution technique is similar to his conservative expression substitution algorithm. His technique of loop guard elimination is a restricted form for exploiting implications between conditional checks in the check implication graph. A weakness of Asuru's proposal to exploit range checks that post-dominate an array reference is that it does not detect a range violation until after it occurs; we avoid this problem in our optimizations.

6 Conclusions

We have synthesized an algorithm for range check optimization from previously developed techniques, and we have shown the relationship of the previous techniques with this algorithm. We implemented the range check optimizer and provided a thorough experimental evaluation of different alternatives for range check optimizations on a suite of non-trivial Fortran programs.

Our experimental results indicate that insertion of range checks in programs is beneficial and that there are significant differences in the number of checks eliminated by different check placement schemes. Simple optimizations such as preheader insertion with loop-limit-substitution of linear checks greatly reduce the execution overhead of range checks with only a moderate increase in compilation time. Increasing the sophistication of the analysis and optimization algorithms increases compilation time, but does not necessarily produce appreciable improvements.

Although our experimental results were obtained on a specific set of programs written in Fortran, we believe that they would be applicable to a larger variety of programs written in other languages. These results should improve the practical usefulness of range checking by encouraging compiler writers to perform range check optimization and programmers to use range checking in production versions of programs.

References

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

[2] J. M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, vol. 1, no. 2, 109-118, June 1992.

[3] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, 159-170, June, 1994.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the 4^{th} ACM Symposium on Principles of Programming Languages*, 238-252, January, 1977.

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

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, October, 1991.

[7] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, to appear.

[8] S. M. German. Automating proofs of the absence of common runtime errors. *Conference Record of the 5^{th} ACM Symposium on Principles of Programming Languages*, 105-118, January, 1978.

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

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

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

[12] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 224-234, June, 1992.

[13] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 114-119, June 1982.

[14] E. Morel. Data flow analysis and global optimization. in *Methods and tools for compiler construction*, B. Lorho (ed.), Cambridge University Press, New York, 289-315, 1984.

[15] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, vol. 2, no. 2, 96-103, February, 1979.

[16] B. Schwarz, W. Kirchgassner, and R. Landwehr. An optimizer for Ada – design, experiences and results. *Proceedings SIGPLAN '88 Conference on Programming Language Design and Implementation*, 175-185, June, 1988.

[17] N. Suzuki and K. Ishihata. Implementation of an array bound checker. *Conference Record of the 4^{th} ACM Symposium on Principles of Programming Languages*, 132-143, January, 1977.

[18] M. Wolfe. Beyond induction variables. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 162-174, June, 1992.