Optimizing Array Bound Checks Using Flow Analysis

RAJIV GUPTA
University of Pittsburgh

Bound checks are introduced in programs for the run-time detection of array bound violations. Compile-time optimizations are employed to reduce the execution-time overhead due to bound checks. The optimizations reduce the program execution time through *elimination* of checks and *propagation* of checks out of loops. An execution of the optimized program terminates with an array bound violation if and only if the same outcome would have resulted during the execution of the program containing all array bound checks. However, the point at which the array bound violation occurs may not be the same. Experimental results indicate that the number of bound checks performed during the execution of a program is greatly reduced using these techniques.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*error handling and recovery*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms: Languages, Reliability

Additional Key Words and Phrases: Available checks, check hoisting, data-flow analysis, very busy checks

1. INTRODUCTION

To aid in the debugging of programs under development, many compilers generate run-time checks to detect errors due to array bound violations dynamically. The overhead of these checks is quite high, resulting in inefficient code with high execution times. Earlier investigations indicate that execution times for programs can double if run-time checks are performed [Chow 1983]. This is true for both *optimized* and *unoptimized* code, because traditional optimizations are ineffective in reducing the overhead due to array bound checks [Chow 1983]. Most compilers allow the programmer to control the generation of run-time checks through a switch that is specified at compile time. The programs are compiled with run-time checks only during

This work was supported in part by National Science Foundation Presidential Young Investigator Award CCR-9157371 to the University of Pittsburgh. A preliminary version of this paper appeared in the *Proceedings ACM SIGPLAN 90 Conference on Programming Language Design and Implementation.*
Author's address: Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1993 ACM 1057-4514/93/0300-0135 $03.50

the debugging phase. When the software is being used in a production environment, it does not include run-time checks. Thus, even if the software appears to execute normally, it may be providing incorrect results due to the array bound violations. To ensure high reliability, the run-time checks should not be removed from the software. This reasoning has led to research into the optimization of run-time checks. In this paper optimizations that significantly reduce the run-time overhead due to array bound checks are presented. Thus, the security of correct execution can be achieved at an acceptable run-time cost.

Preceding each array reference, a *bound check* corresponding to the array reference is introduced. A bound check is essentially a Boolean expression that checks the lower and upper bounds of a subscript expression. If the Boolean evaluates to true, then there is no array bound violation. If it evaluates to false, a trap is taken that terminates the execution and reports an error. The reduction of run-time overhead due to bound checks is treated as an optimization performed through compile-time analysis.

The optimizations described in this paper reduce the run-time overhead through *elimination* and *propagation* of bound checks. The checks that can either be performed at compile time or are made unnecessary by other checks are *eliminated.* This is analogous to the traditional optimizations of constant folding and common subexpression elimination. In certain instances a check can also be eliminated by combining it with another check. The *propagation* of bound checks out of loops reduces the number of times a check is executed at run time. This is analogous to the loop invariant code motion optimization. However, propagation of bound checks differs from invariant code motion in that it replaces a series of distinct bound checks executed during different loop iterations by a single check outside the loop. Although the optimizations in this paper eliminate and propagate bound checks, they do not degrade the reliability of the software. Any array bound violation that is detected by a program with all bound checks included will also be detected by the program after bound check optimization. However, the violations may not be detected at the same point in the program. The bound check optimizer can be integrated into a traditional *code optimizer*. The results of other optimizations, such as constant propagation, can enable the elimination of some bound checks that otherwise may not have been eliminated.

In subsequent sections, first we briefly describe the earlier work done in this area. Next, the bound check optimizations are described, and algorithms to perform these optimizations are discussed in detail. Finally, some experimental results demonstrating the effectiveness of the bound check optimizations are presented.

2. BACKGROUND

Markstein et al. [1982] also treated the elimination and propagation of bound checks as an optimization problem. Their work provided the inspiration for this work, and we use the same approach. The elimination and propagation algorithms developed by Markstein et al. [1982] were developed in conjunction with a traditional code optimizer. In some situations their algorithms are able to benefit from the application of other code optimizations [Aho et al. 1986; Callahan et al. 1986; Wegman and Zadeck 1984]. The algorithms described in this work do not rely on other code optimizations. However, their effectiveness can be enhanced by performing traditional optimizations. The analysis algorithms presented in this paper exploit semantic information that is not exploited by Markstein's algorithms. Semantic properties of variable definitions, namely, monotonic definitions, that do not kill previously performed bound checks on the redefined variable are exploited. In addition, semantic properties of bound checks that cause one check to subsume another check are also exploited. Thus, certain bound checks optimized by our algorithms cannot be handled using Markstein's algorithms. Some important differences will be discussed in subsequent sections, which describe our algorithms in detail.

Harrison [1977] also used compile-time analysis to reduce the overhead due to bound checks. Compile-time techniques of range propagation and range analysis are employed yielding bounds on the values of variables at various points in a program. The range information is used to eliminate redundant bound checks on array subscripts. However, in contrast to Markstein's work, the techniques developed by Harrison do not reduce the run-time overhead due to bound checks that cannot be eliminated at compile time. Harrison's techniques will not be effective in the presence of dynamic arrays, since value range analysis will not be effective.

Suzuki and Ishihata [1977] discussed the implementation of a system that performs array bound checks on a program. The system creates logical assertions immediately before array element accesses that must be true for the program to be valid. These assertions are then proved, by a theorem prover, using techniques similar to inductive assertion methods. Such techniques are significantly more expensive than the techniques based on data-flow analysis. Suzuki and Ishihata's approach cannot reduce the run-time overhead due to bound checks that cannot be eliminated at compile time.

3. BOUND CHECK OPTIMIZATIONS

In this section we discuss intraprocedural bound check optimizations for eliminating and propagating bound checks. We assume that the optimizer can distinguish the bound checks from the remainder of the program and that no control flow instructions are introduced in the code by the bound checks. The upper and lower bounds of an array a are referred to as $MAX(a)$ and $MIN(a)$, respectively. The lower and upper bound checks are treated separately, as sometimes it is possible to optimize only one of the checks. After the optimizations have been applied, we are guaranteed that the optimized program will report an error if and only if the original program would have also terminated with an error. However, after the propagation and elimination of checks, the optimized program may detect an error at a different point in the program. This is not a serious limitation, because even if no bound check optimizations are applied, the point of error detection may be altered due to other program optimizations, such as invariant code motion.

3.1 Local Elimination

In certain situations checks can be eliminated from a basic block through very simple local analysis. If a bound check is *identical* to another check or if it is *subsumed* by another check, it can be eliminated. These situations are described below. In general, more opportunities for combining checks may exist. However, we limit the combining of checks to situations that can be easily identified by examining the checks. Furthermore, the simple situations described below arise often in real programs.

Identical checks. Assume that checks C and C' are two identical checks in a basic block and that the execution of C precedes the execution of C'. If the variables used in the checks are not redefined after C and before C', then C' is eliminated.

Subsumed checks with identical bounds. Consider two bound checks whose lower (or upper) bounds are identical, although the subscript expressions on which the bound checks are performed are not the same. If the two subscript expressions are functions of the same variable and the compiler determines that one expression is always greater than the other, then one check can be eliminated, because it is subsumed by the other check:

$$\begin{array}{lllll} MIN \leq f(v) & \text{and} & MIN \leq g(v) \equiv MIN \leq f(v) & \text{if} & f(v) < g(v), \\ f(v) \leq MAX & \text{and} & g(v) \leq MAX \equiv g(v) \leq MAX & \text{if} & f(v) < g(v). \end{array}$$

Subsumed checks with identical subscript expressions. Consider two bound checks on the same subscript expression with different lower (or upper) bounds. This situation can occur if the same subscript expression is used to refer to elements of two arrays of different dimensions. In this case also one of the checks can be eliminated because it is subsumed by the other as shown below. Assuming that the bounds MIN_1, MIN_2, MAX_1, and MAX_2 are compile-time constants, this optimization is beneficial:

$$\begin{array}{lll} MIN_1 \leq f & \text{and} & MIN_2 \leq f \equiv maximum(MIN_1, MIN_2) \leq f, \\ f \leq MAX_1 & \text{and} & f \leq MAX_2 \equiv f \leq minimum(MAX_1, MAX_2). \end{array}$$

The techniques developed by Markstein et al. [1982] use common subexpression elimination, which can only eliminate identical checks. Their techniques cannot eliminate subsumed checks in the manner discussed above. It should be noted that the application of subsumption optimizations may cause an error to be reported at an earlier program point. The example in Figure 1 demonstrates the results of local elimination. In this example, optimization resulted from the first two situations.

3.2 Global Elimination

Global flow analysis is employed to collect information that is used to eliminate bound checks from a program. Opportunities for the elimination of checks arise because often the same check is performed repeatedly or some of the checks performed subsume other checks performed during execution.

```
-- MIN(a) ≤ i+1 ≤ MAX(a)          -- MIN(a) ≤ i, i+1 ≤ MAX(a)
temp ← a[i+1]                     temp ← a[i+1]
-- MIN(a) ≤ i+1 ≤ MAX(a)          a[i+1] ← a[i]
-- MIN(a) ≤ i ≤ MAX(a)            a[i] ← temp
a[i+1] ← a[i]
-- MIN(a) ≤ i ≤ MAX(a)
a[i] ← temp
Before Optimization               After Optimization
```

Fig. 1. Local elimination of bound checks.

```
if ( ) then                       if ( ) then
    -- 10 ≤ i ≤ 50                    -- 10 ≤ i ≤ 50
    ....                              ....
else                              else
    -- 20 ≤ i ≤ 100                   -- 20 ≤ i ≤ 100
    ....                              ....
fi                                fi
-- 5 ≤ i ≤ 200                    ....
....
Before Optimization               After Optimization
```

Fig. 2. Global elimination of redundant bound checks.

There are two types of situations that can lead to the elimination of bound checks. First, a check at a program point can be eliminated if bound checks can be found that are executed prior to reaching this point and if these checks subsume the current check. In other words, the check is *redundant* because it will not cause an error if the control reaches the point at which the check appears. This optimization is illustrated by the example in Figure 2. Since the check $10 \leq i$ or the check $20 \leq i$ is performed before $5 \leq i$, the check $5 \leq i$ is redundant. Similarly, one of two checks $i \leq 50$ or $i \leq 100$ is performed before $i \leq 200$, which causes the check $i \leq 200$ to be redundant.

The second situation in which a check can be removed arises when, along all paths following the execution of a check, other checks are encountered, and the former check is subsumed by the latter checks. By appropriately *modifying* the former check, at least one of the latter checks can be eliminated. In other words, the modification of an earlier check causes a later check to become redundant. Consider the example in Figure 3. The execution of the check $5 \leq i$ is followed by the execution of $10 \leq i$ or $20 \leq i$. If we replace $5 \leq i$ by $10 \leq i$, then the later execution of $10 \leq i$ becomes redundant. Similarly, we observe that the execution of $i \leq 200$ is followed by either the execution of $i \leq 50$ or the execution of $i \leq 100$. We can replace the execution of $i \leq 200$ by $i \leq 100$, which causes the later execution of $i \leq 100$ to become redundant. As shown in Figure 3, we can perform this optimization in two steps: First, we modify the checks, and then we eliminate the redundant checks created through the modification. The error will be reported at an earlier program point following the optimization.

```
-- 5 ≤ i ≤ 200          -- 10 ≤ i ≤ 100          -- 10 ≤ i ≤ 100
if ( ) then             if ( ) then              if ( ) then
    -- 10 ≤ i ≤ 50          -- 10 ≤ i ≤ 50           -- i ≤ 50
    ....                    ....                     ....
else                    else                     else
    -- 20 ≤ i ≤ 100         -- 20 ≤ i ≤ 100          -- 20 ≤ i
    ....                    ....                     ....
fi                      fi                       fi
Before Optimization     After Modification       After Elimination
```

Fig. 3. Global elimination by modification of bound checks.

To perform the above optimizations, we develop two algorithms: The first algorithm is used to modify bound checks in order to create additional redundant checks. The second algorithm carries out the elimination of redundant checks. Since the first algorithm creates additional redundant checks, it is applied before the second algorithm for the removal of redundant checks is applied. These algorithms are based on the notions of *available* checks and *very busy* checks, which is analogous to the notions of available expressions and very busy expressions. However, there are important differences between bound checks and ordinary expressions, which allow us to be more aggressive in the propagation of bound checks. First, a bound check is not always killed by a definition of a variable used in the subscript expression. In the presence of certain definitions, a check may be propagated. For example, the check $10 \leq i$ can be propagated from the point before statement $i \leftarrow i + 1$ to the point after the statement. The definitions of interest are termed as *monotonic definitions*. Second, bound checks can also be propagated across points where multiple paths meet in an aggressive fashion. If checks $10 \leq i$ and $20 \leq i$ arrive at a point along two distinct paths, we may consider the weaker check $10 \leq i$ to be available at that point. The conditions under which a bound check can be eliminated are also less restrictive than the conditions under which an expression evaluation can be eliminated. If a check is subsumed by an available check, it can be removed. However, an expression at a given point in the program can only be eliminated during global common subexpression elimination if an identical expression is available at that point. Next, we present algorithms for modification and elimination of bound checks that exploit the above semantic information.

Algorithm for modifying checks to create redundant checks. In order to identify checks that can be modified, we carry out data-flow analysis, which propagates the checks performed at various points in the program in the backward direction along the program paths. This process enables us to compute *very busy* checks, which can be precisely defined as follows:

Definition 3.1. A bound check C is *very busy* at a program point p if it is guaranteed that, along each path starting at point p, either C is performed or a check that subsumes C is performed.

We formulate the task of computing very busy checks as a data-flow problem. The result of solving the data-flow problem is the set of very busy checks at all points in the program. Following this analysis we examine each bound check in conjunction with the set of very busy checks immediately after the check and determine whether the check should be modified. Next, the steps of the algorithm are discussed in detail.

Step 1: *compute local information for all basic blocks.* For each basic block B, we compute the set of checks $C\_GEN[B]$ such that at the start of this basic block we can assert that these checks will be performed inside B. Thus, these are the very busy checks at the start of B that are performed in B. The set $C\_GEN[B]$ is computed by examining the checks in B and the definitions of variables examined by the checks. In addition, we also compute the effect that a block B has on the value of each variable v, which is used by some subscript expression in the program. A monotonic definition of a variable is one for which the compiler can predict whether the value of the variable is either definitely going to increase or definitely going to decrease. It is beneficial to take advantage of monotonic definitions, because the definitions of variables used in subscript expressions are very often monotonic. In this work we use a simple approach for detecting monotonic definitions. A more general algorithm for detecting monotonic computations can be found in Gupta and Spezialetti [1991]. The effect of B on variable v is summarized by $EFFECT(B, v)$, as shown below. Here, v_{before} and v_{after} denote the values of variable v before and after a block, respectively, and c is a positive compile-time constant:

$$EFFECT(B, v) = \begin{cases} \textit{unchanged}: & v_{after} \leftarrow v_{before} \\ \textit{increment}: & v_{after} \leftarrow v_{before} + c, \quad \text{where} \quad c \text{ is a positive integer}, \\ \textit{decrement}: & v_{after} \leftarrow v_{before} - c, \quad \text{where} \quad c \text{ is a positive integer}, \\ \textit{multiply}: & v_{after} \leftarrow v_{before} * c, \quad \text{where} \quad c \text{ is a positive integer}, \\ \textit{div} > 1: & v_{after} \leftarrow v_{before} \operatorname{div} c, \quad \text{where} \quad c \text{ is a positive integer}, \\ \textit{div} < 1: & v_{after} \leftarrow v_{before} \operatorname{div} c, \quad \text{where} \quad 0 < c < 1, \\ \textit{changed}: & \text{relationship between } v_{after} \text{ and } v_{bcfore} \text{ is not known.} \end{cases}$$

Step 2: *Compute very busy checks.* With each block B, we associate the set $C\_IN[B]$, which is the set of very busy checks at the entry to the basic block, and $C\_OUT[B]$, which is the set of very busy checks at the exit of the basic block. These sets are computed by solving the data-flow equations given below. $Succ(B)$ denotes the set of basic blocks that are successors of B. The

```
backward( $C\_OUT[B], B$ ) {
    S = ∅
    for each check $C \in C\_OUT[B]$ do
        case $C$ of
        $lb \le v$:
            case AFFECT($B$,$v$) of
                unchanged: S = S ∪ {$lb \le v$}
                increment:  /* the check is killed */
                decrement: S = S ∪ {$lb \le v$}
                multiply:   /* the check is killed */
                div>1:      S = S ∪ {$lb \le v$}
                div<1:      /* the check is killed */
                changed:    /* the check is killed */
            end case
        $v \le ub$:
            case AFFECT($B$,$v$) of
                unchanged: S = S ∪ {$v \le ub$}
                increment: S = S ∪ {$v \le ub$}
                decrement: /* the check is killed */
                multiply:   S = S ∪ {$v \le ub$}
                div>1:      /* the check is killed */
                div<1:      S = S ∪ {$v \le ub$}
                changed:    /* the check is killed */
            end case
        $lb \le f(v)$:
            case AFFECT($B$,$v$) of
                unchanged: S = S ∪ {$lb \le f(v)$}
                increment, multiply, div<1: if $f(v)$ decreases when $v$ increases then S = S ∪ {$lb \le f(v)$} fi
                decrement, div>1: if $f(v)$ decreases when $v$ decreases then S = S ∪ {$lb \le f(v)$} fi
                changed:    /* the check is killed */
            end case
        $f(v) \le ub$:
            case AFFECT($B$,$v$) of
                unchanged: S = S ∪ {$f(v) \le ub$}
                increment, multiply, div<1: if $f(v)$ increases when $v$ increases then S = S ∪ {$f(v) \le ub$} fi
                decrement, div>1: if $f(v)$ increases when $v$ decreases then S = S ∪ {$f(v) \le ub$} fi
                changed:    /* the check is killed */
            end case
        end case
    od
    return ( S )
}
```

Fig. 4. Backward propagation of bound checks.

function *backward* (see Figure 4) computes the checks that are very busy on entry to a basic block as a consequence of the checks being very busy at the exit of the block. The implementation of backward propagation takes advantage of monotonic definitions. If there is no change in the value of a variable involved in a bound check, the check is passed through unchanged. If a change is made through a monotonic definition, the same check may be propagated. Finally, if the change to the variable's value is unpredictable at compile time, the bound check is killed:

$$C\_IN[B] = C\_GEN[B] \vee backward(C\_OUT[B], B),$$

$$C\_OUT[B] = \bigwedge_{S \in Succ(B)} C\_IN[S], \quad \text{where } B \text{ is not the terminating block,}$$

$$C\_OUT[B] = \varnothing, \text{ where } B \text{ is the terminating block;}$$

$$S_1 \wedge S_2 \wedge \cdots \wedge S_n$$
$$= \{C : \forall S_i, 1 \le i \le n, (C \in S_i \vee \exists C' \in S_i \wedge C' \text{ subsumes } C)\},$$

$$S_1 \vee S_2 \vee \cdots \vee S_n$$
$$= \{C : (\exists S_i, 1 \le i \le n, C \in S_i) \wedge (\nexists\, C' \in S_i, 1 \le i \le n, C' \text{ subsumes } C)\}.$$

Step 3: *Modify checks.* The set of checks very busy at each point inside a block B is determined from $C\_OUT[B]$. A check C is modified if we determine that there is another check C' that is very busy at the point immediately following C and C' subsumes C. The modification replaces C by C'. Recall that the conditions under which one check subsumes another were described in Section 3.1.

Algorithm for the elimination of redundant checks. In order to identify checks that are redundant, we carry out data-flow analysis, which propagates the checks performed at various points in the program in the forward direction along the program paths. This process enables us to compute *available* checks, which can be precisely defined as follows:

Definition 3.2. A bound check C is *available* at a program point p if it is guaranteed that, along each path leading to p, either C is performed or a check that subsumes C is performed.

We formulate the task of computing available checks as a data-flow problem. The result of solving the data-flow problem is the set of available checks at all points in the program. Following this analysis we examine each bound check in conjunction with the set of available checks immediately before the check and determine whether the check is redundant. Next, the steps of the algorithm are discussed in detail.

Step 1: *Compute local information for all basic blocks.* For each basic block B, we compute the set of checks $C\_GEN[B]$ such that at the end of the basic block we can assert that these checks were performed inside B. Thus, these are the available checks at the end of B that were performed in B. The set $C\_GEN[B]$ is computed by examining the checks in B and the definitions of variables examined by the checks. In addition, we assume that the *EFFECT* information computed during the detection of very busy checks is also available.

Step 2: *Compute available checks.* For each basic block B, we compute the set $C\_IN[B]$, which is the set of checks available on entry to the basic block, and $C\_OUT[B]$, which is the set of checks available at the exit of the basic block. These sets are computed by solving the data-flow equations given

```
forward( $C\_IN[B]$, $B$ ) {
    S = $\emptyset$
    for each check $C \in C\_IN[B]$ do
        case $C$ of
        $lb \le v$:
            case AFFECT($B$,$v$) of
                unchanged: S = S $\cup$ {$lb \le v$}
                increment: S = S $\cup$ {$lb \le v$}
                decrement: /* the check is killed */
                multiply:  S = S $\cup$ {$lb \le v$}
                div>1:     /* the check is killed */
                div<1:     S = S $\cup$ {$lb \le v$}
                changed:   /* the check is killed */
            end case
        $v \le ub$:
            case AFFECT($B$,$v$) of
                unchanged: S = S $\cup$ {$v \le ub$}
                increment: /* the check is killed */
                decrement: S = S $\cup$ {$v \le ub$}
                multiply:  /* the check is killed */
                div>1:     S = S $\cup$ {$v \le ub$}
                div<1:     /* the check is killed */
                changed:   /* the check is killed */
            end case
        $lb \le f(v)$:
            case AFFECT($B$,$v$) of
                unchanged: S = S $\cup$ {$lb \le f(v)$}
                increment, multiply, div<1: if $f(v)$ increases when $v$ increases then S = S $\cup$ {$lb \le f(v)$} fi
                decrement, div>1: if $f(v)$ increases when $v$ decreases then S = S $\cup$ {$lb \le f(v)$} fi
                changed:   /* the check is killed */
            end case
        $f(v) \le ub$:
            case AFFECT($B$,$v$) of
                unchanged: S = S $\cup$ {$f(v) \le ub$}
                increment, multiply, div<1: if $f(v)$ decreases when $v$ increases then S = S $\cup$ {$f(v) \le ub$} fi
                decrement, div>1: if $f(v)$ decreases when $v$ decreases then S = S $\cup$ {$f(v) \le ub$} fi
                changed:   /* the check is killed */
            end case
        end case
    od
    return ( S )
}
```

Fig. 5. Forward propagation of bound checks.

below. *Pred*(B) denotes the set of basic blocks that are predecessors of B. The forward propagation of checks through a basic block is implemented using the function *forward* (see Figure 5), which takes advantage of monotonic definitions:

$$C\_OUT[B] = C\_GEN[B] \vee forward(C\_IN[B], B),$$

$$C\_IN[B] = \bigwedge_{P \in Pred(B)} C\_OUT[P], \text{ where } B \text{ is not the initial block,}$$

$$C\_IN[B] = \emptyset, \text{ where } B \text{ is the initial block.}$$

Step 3: *Eliminate redundant checks.* A check C is eliminated if we determine that there is a check C' that is available at the point immediately

```
-- MIN(a) ≤ i ≤ MAX(a)
S1: a[i] ← ...
S2: if X then
        -- MIN(a) ≤ i ≤ MAX(a)
        S3: a[i] ← a[i] + 1 fi
-- MIN(a)-1 ≤ i ≤ MAX(a)-1
S4: a[i+1] ← ...
Before Optimization
```

```
-- MIN(a) ≤ i ≤ MAX(a)-1
S1: a[i] ← ...
S2: if X then
        -- MIN(a) ≤ i ≤ MAX(a)-1
        S3: a[i] ← a[i] + 1 fi
        -- MIN(a)-1 ≤ i ≤ MAX(a)-1
S4: a[i+1] ← ...
After Modification
```

```
-- MIN(a) ≤ i ≤ MAX(a)-1
S1: a[i] ← ...
S2: if X then
        S3: a[i] ← a[i] + 1 fi
S4: a[i+1] ← ...
After Elimination of Redundant Checks
```

Fig. 6. Global elimination of checks.

preceding C and that either C' is identical to C or C' subsumes C. The set of checks available at each point inside a block B is determined from $C\_IN[B]$.

In Figure 6, since after the execution of statements $S1$ and $S3$, and before the execution of statement $S5$, variable i is incremented, the check $i \leq MAX(a)$ before $S1$ and $S3$ is *modified* to $i \leq MAX(a) - 1$. The bound checks performed before the execution of $S1$ are available during the execution of $S3$ and $S4$. Thus, no bound checks need to be performed before executing $S3$ and $S4$. Markstein's algorithm would not have eliminated the check $i \leq MAX(a)$ before $S4$ because it does not take advantage of modified checks.

The elimination process described above is more general than the algorithms described in previous work [Gupta 1990]. In the earlier version of this work, modification of checks, which creates additional redundant checks, was not carried out. Thus, the propagation algorithms *forward* and *backward* developed in this paper are more general. Although the forward and backward propagation algorithms modify checks during propagation, this cannot cause an unbounded increase in the sizes of data-flow sets. The size of a data-flow set is limited to $2 \times V + N$, where V is the number of unique variables in checks of the form $lb \leq v$ and $v \leq ub$, and N is the number of bound checks of the form $lb \leq f(v)$ and $f(v) \leq ub$.

3.3 Propagation of Checks Out of Loops

The goal of propagation is to reduce the number of times the checks are executed by moving them out of loops. Since propagation moves the checks to an earlier point in the code, an error detected following this optimization will be detected at a point different from the point at which it would have been detected in the original code. In this section an algorithm to propagate bound checks out of a loop is presented. The innermost loops are processed first, and the outermost loops are processed last. Thus, a bound check may be propagated across multiple nesting levels. Consider the example shown in Figure 7. Assume that there are no definitions of variables i and j in the loop. The bound checks $1 \leq i \leq 10$ can be moved out of the loop because they are

```
repeat
    if ( ) then
        -- $10 \le i \le 100$
        -- $1 \le j \le 10$
        ....
    else
        -- $5 \le i \le 50$
        -- $1 \le j \le 10$
        ....
    fi
until ( )
```

Before Optimization

```
-- $1 \le j \le 10$
-- $5 \le i \le 100$
repeat
    if ( ) then
        -- $10 \le i$
        ....
    else
        -- $i \le 50$
        ....
    fi
until ( )
```

After Propagation

Fig. 7. Propagation of bound checks.

executed during each loop iteration. When we consider the bound checks on j, we find that the checks on the **then** part and the **else** part are not identical. In this case, the weaker checks $5 \le i \le 100$ can be moved out of the loop, whereas the stronger checks $10 \le i$ and $i \le 50$ must be left in the loop. If the loop iterates only once, we would perform three checks on j in the optimized code and only two checks in the optimized code. However, if the loop iterates more than once we are guaranteed that the number of checks executed after propagation will be lower. Since typically loops iterate more than once, it is beneficial to perform propagation of such checks. In previous version of this work, only identical checks were propagated [Gupta 1990]. Thus, the bound checks on variable i would not have been propagated out of the loop.

Algorithm for propagation. This algorithm first identifies the bound checks that are potential candidates for propagation. These checks are either invariant with respect to the loop or can be suitably modified to allow their propagation. If a check is moved out of a loop, we must be sure that if the loop is executed so will the check. If a candidate check belongs to a basic block that dominates all loop exits, we can be sure that the check can be moved outside the loop. In addition, it may be possible for us to propagate some checks from blocks that do not dominate loop exits to blocks that do dominate loop exits. This process is analogous to code hoisting and creates additional opportunities for the propagation of checks outside the loop. Next, we describe the steps of the algorithm in detail. We assume that prior to applying the following algorithm we have identified the loops and have computed *use-def* chains and dominator sets.

Step 1: *Identify candidates for propagation.* A check C is a candidate for moving out of the loop if once executed it need not be executed in subsequent loop iterations. This situation arises if the bound check is loop invariant or if the definitions of a variable used in a bound check consistently cause the value of the variable to either increase or decrease. In case of some loops, it is possible to modify a bound check and place it outside the loop. For simplicity, we assume that the loop is executed at least once. However, this assumption is not necessary, since the checks propagated out of a loop can be preceded by a predicate that ensures that the check is only executed if the loop is executed at least once.

(i) *Invariants*. A check C is a candidate for propagation if it uses only definitions from outside the loop body (i.e., C is a loop invariant). This is determined from the *use-def* information.

(ii) *Increasing values*. A check C is a candidate for propagation if it is of the form $lb \leq i$ and if all definitions of i inside the loop are of the form $i \leftarrow i + c$, $i \leftarrow c + i$, $i \leftarrow i * c$, or $i \leftarrow c * i$, where c is a positive integer constant.

(iii) *Decreasing values*. A check C is a candidate for propagation if it is of the form $i \leq ub$ and if all definitions of i inside the loop are of the form $i \leftarrow i - c$, $i \leftarrow -c + i$, or $i \leftarrow i/c$, where c is a positive integer constant.

(iv) *Loops with increment/decrement of one*. Consider a check C of the form "$-lb \leq i\, op\, c,\ i\, op\, c \leq ub$," where i is incremented/decremented by one during each iteration, $op \in \{+, -, \text{div}, *\}$, c is a constant, and min and max are the minimum and maximum values taken by the variable i. This check is a candidate for propagation because it can be replaced by the check "$lb \leq \text{min}\ op\, c,\ \text{max}\ op\, c \leq ub$" outside the loop. This situation arises primarily for loops. It should be noted that the bounds min and max need not be compile-time constants, although they must be constant for the loop under consideration. In a compiler that parallelizes scientific programs, we can take advantage of Banerjee's [1988] techniques for computing min and max for subscript expressions in nested loops. Using this information, bound checks may be moved out of a loop nest. In our approach, checks can only be moved out of a loop nest one loop at a time.

Step 2: *Check hoisting*. A check that is a candidate for propagation can be propagated only if it is in a block that dominates loop exits. In order to increase the number of checks that are propagated out of a loop, an attempt is made to hoist candidate checks, identified in Step 1, from blocks that do not dominate all loop exits to blocks that dominate all loop exits. In other words, we hoist checks from blocks that are conditionally executed to blocks that are executed unconditionally during each loop iteration. The set ND contains blocks from which checks will be hoisted, and $C(n)$ contains the checks in block n that are candidates for hoisting. The details of this step are presented in Figure 8.

Step 3: *Propagate checks out of the loop*. Propagate all candidate checks from blocks that dominate all loop exits. In accordance with the rules described in Step 1, some checks are modified in this process, whereas others are propagated unchanged.

Figures 9 and 10 demonstrate the propagation algorithm. In the example shown below, the lower bound check on i and the upper bound check on j are moved out the loop, since the value of variable i increases with each iteration

```
hoist {
    $ND = \{n$: block $n$ does not dominate all loop exits$\}$
    for each block n do
        $C(n) = \{c$: at the entry to $n$ we can assert that *candidate* check $c$ will be executed in $n\}$
    od
    change = true
    while change do
        change = false
        for each block $n \ni Succ(n) \cap ND \neq \emptyset \wedge n$ is the unique predecessor of nodes in $Succ(n)$ do
            $prop = \bigwedge_{S \in Succ(n)} C(S)$
            if $prop \neq \emptyset$ then
                change = true
                hoist checks in $prop$ to $n$
                for each check $c \in prop$ do
                    if $c \in S$, $S \in Succ(n)$ then eliminate $c$ from $S$ fi
                od
            fi
        od
    od
}
```

Fig. 8. Hoisting checks out of loops.

```
                                    -- MIN(a) ≤ i, j ≤ MAX(a)
while                               while
   -- MIN(a) ≤ i ≤ MAX(a)                  -- i ≤ MAX(a)
   -- MIN(a) ≤ j ≤ MAX(a)                  -- MIN(a) ≤ j
   a[i] ≠ a[j]                             a[i] ≠ a[j]
do                                  do
   i = i + 1                               i = i + 1
   j = j - 1                               j = j - 1
od                                  od
Before Propagation                  After Propagation
```

Fig. 9. Propagation out of loops with unknown bounds for subscript variables.

```
for i ← min to max do                     -- MIN(a) ≤ min, max ≤ MAX(a)
   if (inc) then -- MIN(a) ≤ i ≤ MAX(a)   for i ← min to max do
      sum ← sum + a[i]                        if (inc) then
   else -- MIN(a) ≤ i ≤ MAX(a)                          sum ← sum + a[i]
      sum ← sum - a[i]                        else sum ← sum - a[i]
   fi                                         fi
od                                        od
Before Propagation                        After Propagation
```

Fig. 10. Propagation out of loops with known bounds for subscript variables.

and the value of j decreases with each iteration. These optimizations would not have been performed using Markstein's algorithm. Markstein's algorithm only propagates checks out of loops if the loop exit condition is of the form $ic \leq MAX$ or $ic \geq MAX$, where ic is an induction variable and the subscript expressions are linear functions of the induction variable ic. In Figure 10 the

Table I. Effects of Bound Check Optimization

Program	UNOPT	LELIM +	GELIM +	PROP =	Total deleted
BUBBLE	59,400	39,600 +	9,900 +	9,900 =	59,400 ≈ 100%
QUICK	271,184	72,784 +	10,014 +	54,347 =	137,145 ≈ 51%
QUEEN	13,784	2,288 +	1,748 +	1,778 =	5,814 ≈ 42%
TOWERS	556,262	261,944 +	97,844 +	0 =	359,788 ≈ 65%
LLOOP6	20,160	8,064 +	0 +	12,096 =	20,160 ≈ 100%
FFT	37,414	24,568 +	0 +	5,930 =	30,498 ≈ 82%
MATMUL	1,043,200	640,000 +	256,000 +	147,200 =	1,043,200 ≈ 100%
PERM	80,624	10,078 +	0 +	7,240 =	73,384 ≈ 91%

UNOPT = total number of bound checks before optimization; LELIM = number of checks eliminated by local elimination; GELIM = number of checks eliminated by global elimination; PROP = number of checks eliminated by propagation.

bound check on variable i is first hoisted to the beginning of the loop. Next, the bound check is propagated out of the loop. The propagation algorithm proposed by Markstein et al. [1982] does not perform code hoisting.

4. EXPERIMENTAL RESULTS AND CONCLUSION

The bound check optimizations have been applied to a small set of programs. The programs have been first profiled to determine the number of bound checks that would be performed at run time if no optimization was carried out. Next, the optimizations have been applied by hand, and the checks have been eliminated and propagated. The profiling information has been used again to determine the reduction in the number of checks performed after optimization.

In general, the elimination of bound checks can create opportunities for propagation that did not exist before, and vice versa. However, in these experiments we have not applied these optimizations repeatedly. The local and global elimination was performed first followed by propagation. It has also been observed that for these test programs repeated application of optimizations would not have resulted in any further improvement. This can be attributed to the small program sizes.

The results of bound check optimization are presented in Table I. The total number of bound checks before optimization is given by UNOPT. The number of checks that were eliminated by local elimination (LELIM), global elimination (GELIM), and propagation (PROP) are given. The results indicate that substantial reduction in the number of run-time checks results by applying bound check optimizations. As mentioned earlier, previous research showed that similar results cannot be obtained through traditional optimizations [Chow 1983].

ACKNOWLEDGMENTS

The comments from the reviewers have been helpful in improving the algorithms and the presentation of the paper. In particular, observations of one reviewer resulted in the generalization of the algorithms.

REFERENCES

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

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Mass.

CALLAHAN, D., COOPER, K. D., KENNEDY, K., AND TORCZON, L. 1986. Interprocedural constant propagation. In *Proceedings 14th ACM Symposium on Principles of Programming Languages.* (St. Petersburg Beach, Fla., Jan. 13–15). ACM, New York, 152–161.

CHOW, F. 1983. A portable machine-independent global optimizer—Design and measurements. Tech. Rep. 83-254, Ph.D. thesis, Computer Systems Lab, Stanford Univ., Calif.

GUPTA, R. 1990. A fresh look at optimizing array bound checking. In *Proceedings ACM SIGPLAN 90 Conference on Programming Language Design and Implementation* (White Plains, N.Y., June 20–22). ACM, New York, 272–282.

GUPTA, R., AND SPEZIALETTI, M. 1991. Loop monotonic computations: An approach for the efficient run-time detection of races. In *Proceedings of the SIGSOFT Symposium on Testing, Analysis, and Verification.* (Victoria, B.C., Oct. 8–10). ACM, New York, 98–111.

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

MARKSTEIN, V., COCKE, J., AND MARKSTEIN, P. 1982. Optimization of range checking. In *Proceedings of SIGPLAN 82 Symposium on Compiler Construction.* (Boston, Mass., June 23–25). ACM, New York, 114–119.

SUZUKI, N., AND ISHIHATA, K. 1977. Implementation of array bound checker. In *Proceedings 4th ACM Symposium on Principles of Programming Languages.* ACM, New York, 132–143.

WEGMAN, M. N., AND ZADECK, F. K. 1984. Constant propagation with conditional branches. In *Proceedings 12th ACM Symposium on Principles of Programming Languages.* (Salt Lake City, Ut., Jan. 15–18). ACM, New York, 152–161.

Received August 1991; revised March and November 1993; accepted December 1993