A Fresh Look at Optimizing Array Bound Checking

Rajiv Gupta
Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor, NY 10510
E-mail: gupta@philabs.Philips.com

Abstract - This paper describes techniques for optimizing range checks performed to detect array bound violations. In addition to the elimination of range checks, the optimizations discussed in this paper also reduce the overhead due to range checks that cannot be eliminated by compile-time analysis. The optimizations reduce the program execution time and the object code size through **elimination** of redundant checks, **propagation** of checks out of loops, and **combination** of multiple checks into a single check. A **minimal control flow graph (MCFG)** is constructed using which the minimal amount of data flow information required for range check optimizations is computed. The range check optimizations are performed using the MCFG rather the CFG for the entire program. This allows the global range check optimizations to be performed efficiently since the MCFG is significantly smaller than the CFG. Any array bound violation that is detected by a program with all range checks included, will also be detected by the program after range check optimization and *vice versa*. Even though the above optimizations may appear to be similar to traditional code optimizations, similar reduction in the number of range checks executed can not be achieved by a traditional code optimizer. Experimental results indicate that the number of range checks performed in executing a program is greatly reduced using the above techniques.

Keywords - Range Analysis, Flow Analysis, Minimal Control Flow Graph, Constant Propagation, Induction Variable Elimination.

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

1. Introduction

To aid in the debugging of programs under development, many compilers generate run-time checks to detect dynamic errors due to array bound violations. 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[3]. This is true for both *optimized* and *unoptimized* code because traditional optimizations are ineffective in reducing the overhead due to array bound checks[3]. Most compilers allow the programmer to control the generation of run-time checks through a switch to be specified at compile-time. The programs are compiled with run-time checks only during the debugging phase. When the software is being used in production environment it does not include the run-time checks. Thus, even if the software appears to execute normally, it may be providing incorrect results due the run-time errors. To ensure high reliability, the run-time checks should not be removed from the software. In this paper optimizations that greatly reduce the run-time overhead due to array bound checks are presented. Thus, the security of correct execution can be achieved at a small run-time cost.

The reduction of run-time overhead due to range checks is treated as an optimization performed through compile time analysis. The optimizations described in this paper reduce the run-time overhead through *elimination*, *propagation*, and *combination* of range checks. The *elimination* of checks that can either be performed at compile-time or are unnecessarily performed repeatedly is carried out. This is analogous to the traditional optimizations of constant folding and common subexpression elimination. The *propagation* of range 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. A *combination* of multiple checks into a single check is possible in certain instances.

The *range check optimizer*, that performs the optimizations, first constructs a control flow graph (CFG) for the program. The statements in this flow graph are the source level statements rather than intermediate code statements. Preceding each program statement range checks for all array accesses in that statement are introduced. It is assumed that the range check optimizer can distinguish between the original program statements and the range checks. From this augmented CFG a **minimal control flow graph (MCFG)** is constructed which consists only of the range check expressions and those program statements that define identifiers used in these expressions. After the removal of irrelevant code, and subsequently irrelevant basic blocks, a significantly smaller MCFG results. The MCFG is used to perform the range check optimizations and computing the *use-def* data flow information required to perform these optimizations. The size of the MCFG is smaller and therefore the global range check optimizations can be performed efficiently. Furthermore, the use-def information computed to perform the range checks is only a subset of the data flow information for the entire program.

Markstein *et al*[8] developed compiler techniques for eliminating range checks and propagating checks out of inner loops. The elimination algorithm presented in this paper is more general than the one proposed in [8]. This algorithm takes advantage of the **monotonic** nature of definitions. Repeated execution of a definition of the type $i=i+1$ causes the value of i to increase monotonically. Therefore there is no need to repeatedly examine whether i is greater than the lower bound of an array. A very high percentage of subscript variables are monotonic in nature and therefore range checking overhead can be significantly reduced by exploiting monotonicity. The propagation algorithm presented is also more general than the one proposed in [8]. First, it takes advantage of monotonicity. Second, it performs code hoisting prior to propagation which moves checks out of loops that cannot be moved by the algorithm in [8]. The combination optimizations are not performed at all in [8]. Therefore, the techniques presented in this paper optimize more range checks than those developed by Markstein *et al*[8].

Harrison also used compile time analysis to reduce the overhead due to range checks in his work[4]. Compile-time techniques of range propagation and range analysis are employed yielding bounds on the ranges of variables at various points in a program. The range information is used to eliminate redundant range checks on array subscripts. The techniques presented in this paper also reduce the overhead due to range checks through compile-time analysis. However, in contrast to Harrison's work the overhead reduction is not achieved only by elimination of range checks. The techniques presented in this paper reduce the run-time overhead due to range checks that cannot be eliminated by moving the checks out of loops so that they are executed less frequently and combining several checks into one. Harrison's techniques will be less effective when applied to dynamic arrays since value range analysis will not be effective. The elimination and propagation techniques presented in this paper are equally effective for static and dynamic arrays. Harrison's techniques require *use-def* and *def-use* information for the entire program while techniques in this paper require the computation of a subset of *use-def* information and no *def-use* information. The range check optimizations are performed much more efficiently using the MCFG than range analysis and range propagation. Consider the situation where the program contains a statement $i = j + k$ and variable i is used in an array subscript expression. In Harrison's approach value range analysis for i would require range analysis for j and k. However, in the techniques presented in this paper only data flow information regarding i will be computed.

Suzuki and Ishihata discuss the implementation of a system that performs array bound checks on a program in[6]. The system creates logical assertions immediately before array element accesses that must be true for the program to be valid. These assertions are then proven, by a theorem prover, using techniques similar to inductive assertion methods. Such techniques are significantly more expensive than the techniques presented in this paper. Suzuki's system has several limitations. As an example it cannot check the correctness of an array accesses in a loop if the correctness depends on some data whose value is set before the execution of the loop. The techniques presented in this paper are based upon flow analysis thus avoiding such limitations. The array accesses that cannot be verified by Suzuki's system still require dynamic checking. The techniques presented in this paper will also reduce the overhead due to checks that cannot be eliminated.

Although the optimizations in this paper eliminate, propagate, and combine range checks, they do not degrade the reliability of the software. Any array bound violation that is detected by a program with all range checks included, will also be detected by the program after range check optimization and *vice versa*. It is assumed that the range check optimizer is implemented separately from the traditional code optimizer. An environment in which a debugger for optimized code[5] is to be provided, the integration of the range check optimizer with the traditional code optimizer will make it possible to provide highly efficient code during debugging phases.

In subsequent sections the construction of the *minimal CFG* and descriptions of various range check optimizations and their implementations are discussed. These techniques are not only highly efficient but also much more effective in optimizing range checks than traditional code optimizations. Finally some experimental results demonstrating the effectiveness of the range check optimizations are presented.

2. The Minimal Control Flow Graph (MCFG)

For efficient implementation of range check optimizations, the Minimal Control Flow Graph for a program is constructed from the CFG augmented with the range checks to be performed. The basic blocks of the CFG contain a sequence of source level statements rather than intermediate code statements.

Definition: Given a CFG $G=(N,E)$, where N is the set of nodes corresponding to the basic blocks and E is the set of directed edges connecting the nodes. The corresponding MCFG $G_m=(N_m,E_m)$ consists of the following:

(i) $N_m \subseteq N$: A node n_i belongs to N_m if it contains a range check or a definition for an identifier used in a range check.

(ii) E_m: A set of edges that *preserve* the control flow of the original CFG. The control flow is *preserved* if for every directed path $p=<n_{i_1},n_{i_2}..n_{i_s}>$ in G, there is a corresponding path p' in G_m which is obtained by eliminating all nodes $n_{i_j} \notin N_m$ in p.

To construct the MCFG first all statements other than the range checks and those statements that define identifiers used in array subscript expressions are eliminated. Next all redundant

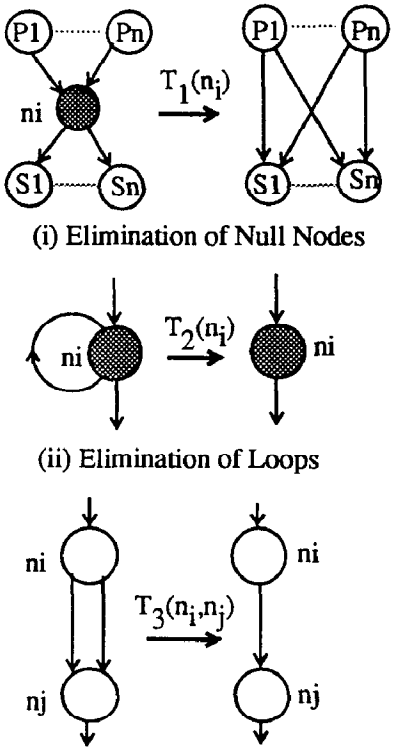


Fig. 1. Transformations for Constructing the MCFG

nodes from this CFG are eliminated to obtain a smaller flow graph which is the MCFG. To eliminate redundant nodes the transformations in Fig. 1 are applied repeatedly. Transformation $T_1(n_i)$ eliminates an empty node n_i and introduces edges that connect every predecessor of n_i with every successor of n_i. $T_2(n_i)$ eliminates self loops to enable elimination of an empty node n_i by $T_1(n_i)$. Transformation $T_3(n_i,n_j)$ is used to eliminate a redundant edge from n_i to n_j.

Claim: Application of transformations T_1, T_2 and T_3 in any order, till no more transformations can be applied, yields a **unique** and **valid** MCFG.

Proof: Each of the above transformation *preserves* the control flow:

(i) $T_1(n_i)$: For each predecessor-successor pair (p_j, s_k) of n_i this transformation replaces the path $<p_j,n_i,s_k>$ in the original flow graph by the path $<p_j,s_k>$ in the transformed flow graph.

(ii) $T_2(n_i)$: The paths containing n_i prior to application of this transformation are of the form $<p_j,n_i^+,s_k>$, where p_j is a predecessor of

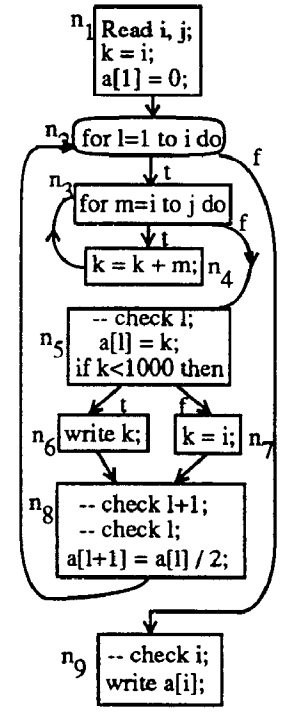
n_i, s_t is a successor of n_i and n_i^+ denotes one or more occurrences of n_i. These paths are replaced by the path $<p_j, n_i, s_t>$ in the transformed flow graph.

(iii) $T_3(n_i, n_j)$: This transformation eliminates redundant edges and therefore all paths present in the original flow graph are also present in the transformed flow graph.

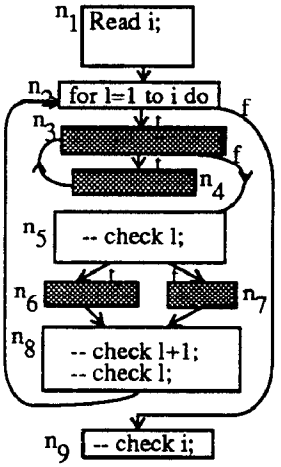
From the above discussion it is clear that each of the above transformations preserve the control flow. Consider any two nodes, say n_i and n_j, that belong to the MCFG. If there is a directed path from n_i to n_j in the original flow graph then there will also be such a path in the MCFG. Similarly if there is no path from n_i to n_j in the original flow graph there cannot be one in the MCFG. This is because when edges are added to the flow graph in transformation T_1 these edges connect nodes which already had a path between them. Furthermore, no edges are added by T_1 which connect nodes that do not have a path between them. The transformation T_3 removes redundant edges and therefore does not create or destroy any paths. The repeated application of these transformations preserves the control flow resulting in a *valid* MCFG.

The MCFG is unique by definition. The nodes contained in the MCFG are the non-empty set of nodes which is unique. The set of edges in the MCFG is also unique. This is because if we consider the existence of two valid MCFG's then there must be at least one execution path in one MCFG that is absent in the other MCFG. However, this implies that one of the MCFG does not preserve the flow of control and therefore cannot be valid. Hence, the MCFG for a given CFG is unique. The order in which the transformations are applied does not effect the outcome. This is because candidates for T_1, T_2, and T_3 remain candidates, even if some other applications of the transformations are made first. □

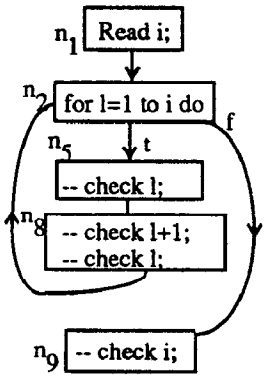
The construction of the MCFG is illustrated in Fig. 2. The code corresponding to the range checks is preceded by '--'. Fig. 2(ii) shows the same CFG after elimination of irrelevant statements. Application of transformations yields the MCFG shown in Fig. 2(iii). Node n_4 is eliminated by applying $T_1(n_4)$. Following this n_3 is eliminated by applying $T_2(n_3)$ followed by $T_1(n_3)$. Transformation T_1 also eliminates n_6 and n_7. This results in two edges from n_5 to n_8 one of which is eliminated using $T_3(n_5, n_8)$.



(i) Augmented CFG



(ii)



(iii) MCFG

Fig. 2. An Example Minimal Control Flow Graph

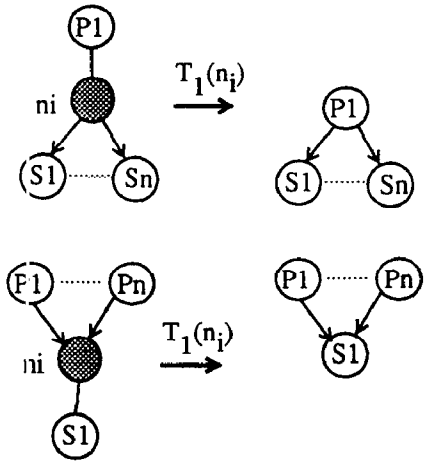


Fig. 3. Elimination of Null Nodes

The purpose of removing empty nodes is to reduce the size of the flow graph so that the range check optimizations can be performed more efficiently. However, removal of certain empty nodes may increase the number of edges in the flow graph. This will increase the amount of computation performed during the range check optimization phase. The increase in the number of edges is caused by transformation $T_1(n_i)$ which connects every predecessor of n_i with every successor of n_j. The increase in the number of edges can be avoided by restricting the application of transformation $T_1(n_i)$ to only those nodes which have either a single predecessor or a single successor (see Fig. 3). It should be noted that if this approach is taken then all empty nodes will not be removed from the CFG.

After the MCFG is constructed *use-def* information for only the *range checks* is computed. If a definition included in the MCFG is not used by any of the range checks it is eliminated. This may result in additional empty nodes which can also be eliminated using the transformations in Fig. 1. The size of the MCFG is further reduced by applying the transformations in Fig. 4. In some instances a node n_i that contains nothing but range checks can be eliminated and its checks moved to its predecessor or successor. The transformation $T_5(n_i)$ shown in Fig. 4 moves checks into the predecessor node of n_i. Similarly $T_6(n_i)$ moves checks to a successor node. Self loops are removed using $T_4(n_i)$ and this may create additional opportunities for applying $T_5(n_i)$. The transformations T_4, T_5

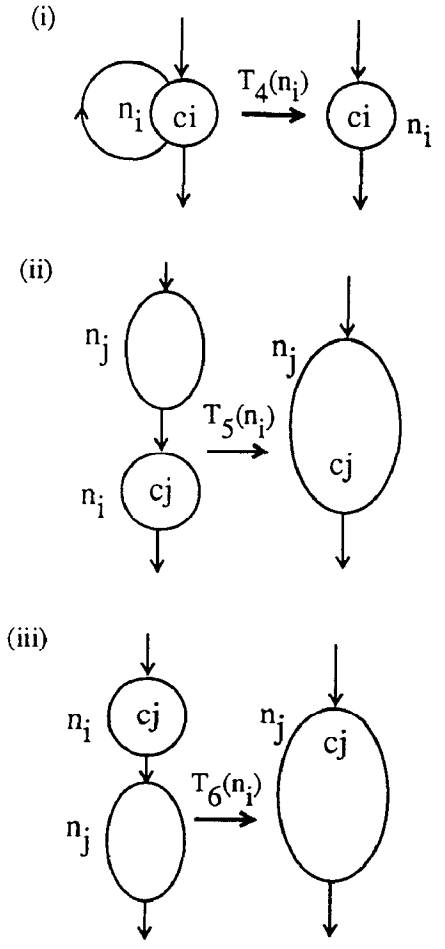


Fig. 4. Additional Transformations

and T_6 also preserve the control flow. The above transformations not only reduce the size of the flow graph but also cause the checks to *propagate*. The application of transformation $T_4(n_i)$ followed by the application of $T_5(n_i)$(or $T_6(n_i)$) propagates the checks belonging to n_i out of a loop.

In the example in Fig.2(iii) nodes n_5 and n_8 are combined using $T_5(n_8)$. The resulting MCFG is shown in Fig. 5(i). After these nodes are combined the check "-- check 1" appears twice in n_5 and will be eliminated through local analysis. It should be noted that elimination of this check would have required global analysis if the original CFG were used instead of the MCFG. The experimental results show that the size of the MCFG is half that of the CFG in most cases. After the MCFG has been constructed range check optimizations are applied. These optimizations are described in subsequent sections. The range checks present in the optimized MCFG will be performed at runtime. The original CFG is modified to include the range checks present in the optimized MCFG. This CFG should be used by the compiler for code gen-

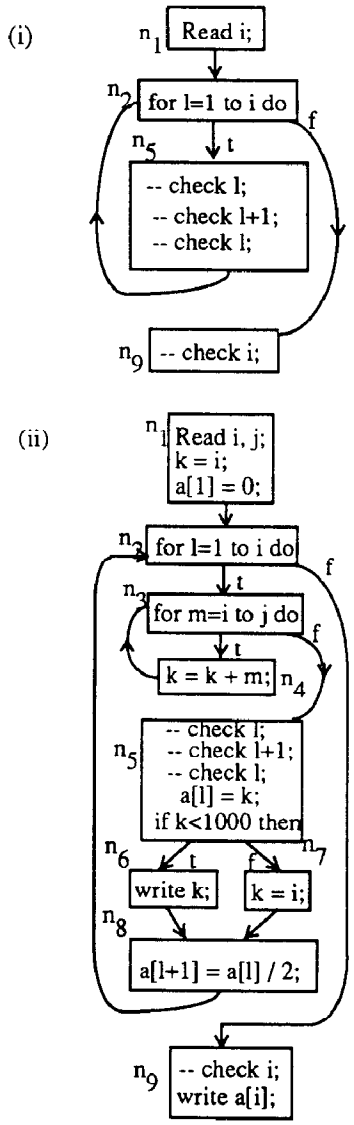


Fig. 5. (i) MCFG; (ii) CFG with Range Checks

eration. The CFG with range checks corresponding to the MCFG in Fig. 5(i) is shown in Fig. 5(ii).

3. Range Check Optimizations

If an array subscript is a constant the range check is performed at compile-time. In this paper, additional optimizations, that are not as direct and require analysis are discussed. 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 one and not the other.

3.1. Local Elimination

If the same check appears more than once in a basic block, and the use-def information indicates that same definitions are used in these checks, all but one of the checks is eliminated. In the example of Fig. 2(iii) after basic blocks n_5 and n_8 are combined the check "-- check l" appears twice in the same basic block. Thus, one of these is eliminated.

3.2. Global Elimination

Global elimination of range checks is carried out through global flow analysis. At a given point in the program the checks that have been performed previously and are still valid are considered to be live. By propagating live checks other checks may be eliminated. Analysis similar to that for available expressions is carried out by formulating the problem as a forward data flow equation[1]. The algorithm for this optimization is shown below. The checks killed by a basic block are computed differently from available expressions since a check may not be killed if an identifier used in the expression is redefined by a **monotonic** definition.

Definition: A definition of an identifier kills a range check if the range check uses that identifier except in the following situations:

(i) *Monotonically Increasing*: A check $lb \leq i$ is not killed by a definition of the form $i \leftarrow i+c$, $i \leftarrow c+i$, and $i \leftarrow c*i$, where constant c takes values 1,2,3,4,...; and

(ii) *Monotonically Decreasing*: A check $i \leq ub$ is not killed by definitions of the form $i \leftarrow i-c$, $i \leftarrow -c+i$, and $i \leftarrow i/c$.

Algorithm for Global Elimination:

Step 1: Compute the following sets for all basic blocks:
C_KILL[B] - set of checks performed outside B that are killed by definitions inside B.
C_GEN[B] - set of checks that reach the end of the basic block B. A check performed in a basic block reaches the end if the definitions between the point at which the check is performed and the end of the basic block do not kill the check.

Step 2: Compute sets C_IN and C_OUT, where C_IN is the set of checks live at the entry of the basic block and C_OUT is the set of checks live upon exit of the basic block by solving the follow-

ing:

$C\_IN[B_1]=\phi$, where B_1 is the initial basic block.
$C\_IN[B]=\bigcap_{P \in Pred(B)} C\_OUT[P]$, where Pred(B) is the set of basic blocks that are predecessors of B.
$C\_OUT[B]=(C\_IN[B]-C\_KILL[B]) \cup C\_GEN[B]$

Step 3: Eliminate redundant checks. A check c_i in basic block B is redundant if there is another check $c_j \epsilon C\_IN[B]$ such that c_j performs the same check as c_i and it is not killed by a definition before it reaches the check c_i.

```
-- MIN(a) ≤ i,  i ≤ MAX(a)
S1: a[i] ← ...
S2: if X then
    -- MIN(a) ≤ i,  i ≤ MAX(a)
    S3: a[i] ← a[i] + 1 endif
S4: i ← i + 1;
-- MIN(a) ≤ i,  i ≤ MAX(a)
S5: a[i] ← ...
```

Before Elimination

```
-- MIN(a) ≤ i,  i ≤ MAX(a)
S1: a[i] ← ...
S2: if X then
    S3: a[i] ← a[i] + 1 endif
S4: i ← i + 1;
-- i ≤ MAX(a)
S5: a[i] ← ...
```

After Global Elimination

In the above example a range check performed for the execution of statement $S1$ is valid during the execution of statement $S3$. Thus, no range check needs to be performed before to executing $S3$. After $S4$ increments i, the check $i \leq MAX(a)$ is killed while the check $MIN(a) \leq i$ is still alive. Thus, the latter check is not made during the execution of $S5$. Monotonicity property is useful because a large percentage of definitions are monotonic.

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. As mentioned earlier some amount of propagation is carried out during the construction of the MCFG. In this section an algorithm to propagate additional range checks out of a loop is presented. The innermost loops are processed first and the outermost loops are processed last. Thus, a range check may be propagated across multiple nesting levels. This optimization is illustrated by the example shown below. As shown, to move a check involving the for loop variable outside a loop, the check must be modified.

```
for i←min to max do
   -- MIN(a) ≤ i,  i ≤ MAX(a)
   -- MIN(a) ≤ j,  j ≤ MAX(a)
   -- MIN(a) ≤ k,  k ≤ MAX(a)
   a[i] ← a[j] + a[k]
   j ← j + 1
end for
```

Before Propagation

```
-- if min ≤ max then
       MIN(a) ≤ min,  max ≤ MAX(a)
-- MIN(a) ≤ j
-- MIN(a) ≤ k,  k ≤ MAX(a)
for i←min to max do
   -- j ≤ MAX(a)
   a[i] ← a[j] + a[k]
   j ← j + 1
end for
```

After Propagation

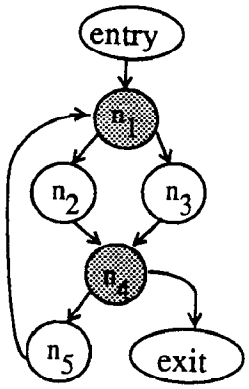
In the above example the check $MIN(a) \leq j$ is moved out of the loop because the value of j increases monotonically. The algorithm by Markstein *et al*[8] does not take advantage of monotonicity. The algorithm that takes a single loop and moves the checks out of the loop is presented next. The checks propagated in Step 2 of this algorithm will not be propagated by the algorithm used by Markstein *et al* [8].

Algorithm for Propagation:

This algorithm first performs code hoisting to propagate checks to those basic blocks that dominate the loop exit. Next checks belonging to the basic blocks dominating the loop exit are considered for propagation out of the loop. The checks that can be propagated are identified and propagated by this algorithm.

Step 1: Determine the blocks from which checks can be propagated out of loops. Checks from a block b are considered for propagation if b *dominates* all loop *exits*, i.e., every path from the *entry*

node to all loop *exits* passes through b. In the loop shown next the shaded blocks satisfy this criterion.



Step 2: Code hoisting - Propagate checks from nodes that do not dominate all loop exits to nodes that dominate all loop exits as follows:

Let ND - be the set of nodes that do not dominate all loop exits,

$C(n)$ - set of checks performed in n, and

P - set of nodes such that for each node $n \epsilon P$, $succ(n) \cap ND \neq \phi$ and n is the unique predecessor of nodes in $succ(n)$, where $succ(n)$ is the successor set of n.

loop {
Propagate a check e into block n from blocks in $succ(n)$ if
$e \epsilon \bigcap_{s \epsilon succ(n)} C(s)$
where $n \epsilon P$, and e uses the same definitions in each block that belongs to $succ(n)$.
} until no more checks can be propagated

In the loop shown above the checks common to nodes n_2 and n_3 may be moved to n_1 in this step.

Step 3: A check e from a block that satisfied the criterion in *step 1* is moved out of the loop if:

(i) Check e uses only definitions from outside the loop body (i.e., e is a loop invariant). This is determined from the *use-def* information computed prior to applying the optimizations; or

(ii) Check e is of the type $lb \leq i$ and 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 constant c takes values 1,2,3,4,..; or

(iii) Check e is of the type $i \leq ub$ and 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 constant c takes values 1,2,3,4...; or

(iv) Check e is of the form "$--lb \leq i$ *op* c, i *op* $c \leq ub$", where i is the for loop variable, $op \epsilon \{+,-,/,*\}$, and c is a constant. The check is modified to "--if min $\leq$ max *then* $lb \leq$ min *op* c, max *op* $c \leq ub$", where ***min*** and ***max*** are the minimum and maximum values taken by the for loop variable.

3.4. Combination of Range Checks

The combination optimizations are local optimizations that combine multiple checks into a single check. Range checks on a variables v_i and v_j, due to their use as array subscripts for elements of the same array, are combined if the only definition of v_j that reaches the check is of the form $v_j \leftarrow v_i$ *op* c, where $op \epsilon \{+,-,/,*\}$ and c is a constant. The range checks for the same array and different arrays can also be combined as long as their subscript expressions are also related in a similar manner as the above optimization. The following example demonstrates this optimization. Since j is obtained by incrementing i the checks $C1$ and $C2$ are combined using the first optimization. The resulting check is combined with with $C3$ using the second optimization.

```
j ← i + 1
C1 -- MIN(a) ≤ i,  i ≤ MAX(a)
C2 -- MIN(a) ≤ j,  j ≤ MAX(a)
C3 -- MIN(b) ≤ i,  i ≤ MAX(b)
a[i] ← a[j] + b[i]
```

Before Combination

```
j ← i + 1
-- maximum(MIN(a), MIN(b)) ≤ i,
   i ≤ minimum(MAX(a)-1, MAX(b))
a[i] ← a[j] + b[i]
```

After Combination

3.5. Interprocedural Optimizations

The techniques described above are applied to a single procedure. It is also possible to perform interprocedural range check optimizations. *Interprocedural propagation* can be performed by moving checks from the start of a procedure to each of the call sites. In other words the checks are propagated from a called procedure to the caller. In the example shown below the checks

$MIN(a) \leq i$ and $j \leq MAX(a)$ are eliminated after they are moved at the call site because i and j are constants at the call site. Interprocedural constant propagation would also have resulted in elimination of these checks if i and j are constants at every call site of procedure *printarray*. Interprocedural propagation is applicable to non-recursive procedures. It should not be applied to recursive and mutually recursive procedures.

```
procedure printarray(a: vector; i,j: int; var k: int)
 var l: int;
 begin
  -- MIN(a) ≤ i,  j ≤ MAX(a)
  for l←i to j do print(a[l]);
  k ← (i + j) div 2;
  -- MIN(a) ≤ k,  k ≤ MAX(a)
  print(a[k]);
 end

begin
 ...
 printarray(a, 1, 5, k);
 -- MIN(a) ≤ k,  k ≤ MAX(a)
 a[k] ← a[k] + 1;
 ...
end
```

Before Interprocedural Optimization

```
procedure printarray(a: vector; i,j: int; var k: int)
 var l: int;
 begin
  for l←i to j do print(a[l])
  k ← (i + j) div 2;
  -- MIN(a) ≤ k, k ≤ MAX(a)
  print(a[k]);
 end

begin
 ...
 -- MIN(a) ≤ 1,  5 ≤ MAX(a)
 printarray(a, 1, 5, k)
 -- MIN(a) ≤ k,  k ≤ MAX(a)
 a[k] ← a[k] + 1;
 ...
end
```

After Interprocedural Propagation

Interprocedural elimination of checks can be achieved by propagating live checks across procedure calls. The checks on value of k are live at the end of the procedure *printarray*. By propagating this information to the caller the checks on k are eliminated in the example shown. However, if there had been other execution paths to the checks for k in the caller this may not have been possible.

```
procedure printarray(a: vector; i,j: int; var k: int)
 var l: int;
 begin
  for l←i to j do print(a[l])
  k ← (i + j) div 2;
  -- MIN(a) ≤ k, k ≤ MAX(a)
  print(a[k]);
 end

begin
 ...
 printarray(a, 1, 5, k)
 a[k] ← a[k] + 1;
 ...
end
```

After Interprocedural Elimination

4. Ordering the Optimizations

The order in which the optimizations are applied is important. The local, global, and interprocedural elimination of range checks should be performed prior to propagation of checks. This is because redundant checks will not be considered as candidates for propagation. Following intraprocedural propagation, the checks that have moved to the top of procedures, are propagated across procedure boundaries. This creates new opportunities for elimination. Thus local and global elimination are repeated again. The combination of checks is carried out last because it may not be possible to eliminate checks after they have been modified by combination. Range check optimization does not degrade the reliability of the software because any array bound violation that is detected by a program with all range checks included, will also be detected by the program after range check optimization.

5. Traditional Code Optimizations vs Range Check Optimizations

The results of range check optimizations can be influenced by other code optimizations. This interaction should be taken into account if the range check optimizations are being implemented as part of a traditional code optimizer. *Constant propagation*[7,2] may enable certain range checks to be performed at compile-time. *Copy propaga-*

tion and *induction variable elimination* may result in removal of checks. If a program performs range checks on variables v_i and v_j and variable v_j is computed by $v_j \leftarrow v_i$ or $v_j \leftarrow v_i + 1$, using copy propagation or induction variable elimination the check on v_j can be eliminated.

```
while (condt) loop
  i ← i + 4;
  T1=not(MIN(a)<=j<=MAX(a));
  if T1 then error;
  a[j] ← 4;
  T2=not(MIN(a)<=j<=MAX(a));
  if T2 then error;
  a[j] ← i + 4;
endwhile;
```

Before Code Optimization

```
T=not(MIN(a)<=j<=MAX(a));
while (condt) loop
  i ← i + 4;
  if T1 then error;
  a[j] ← 4;
  if T2 then error;
  a[j] ← i + 4;
endwhile;
```

After Code Optimization

Although traditional optimizations may help in range check optimization they are not a substitute for the latter. This observation is also supported by the experimental results obtained by Chow[3]. The combination of range checks will not be carried out by a traditional code optimizer. Propagation of checks out of for loops and elimination and propagation of checks that rely on the observation that the value of the subscript variable monotonically increases or decreases in a loop are also not performed by a traditional code optimizer. The example shown above illustrates a situation where one might expect a traditional optimizer to perform well. The two range checks are identical as well as loop invariant. A traditional code optimizer will neither combine the checks nor move them out of the loop. This is because it cannot distinguish between range checks and rest of the code. However, if range check elimination and propagation are used one check is eliminated and the other is moved out of the loop thus requiring a check to be executed once at runtime.

```
while (condt) loop
  i ← i + 4;
  -- MIN(a)<=j, j<=MAX(a))
  a[j] ← 4;
  -- MIN(a)<=j, j<=MAX(a))
  a[j] ← i + 4;
endwhile;
```

Before Range Check Optimization

```
-- MIN(a)<=j, j<=MAX(a))
while (condt) loop
  i ← i + 4;
  a[j] ← 4;
  a[j] ← i + 4;
endwhile;
```

After Range Check Optimization

6. Experimental Results

The results of applying *intraprocedural* range check optimizations to a small set of programs are shown in the Table below. The execution times of the programs without range check optimizations (NOPT), after elimination (ELIM), after elimination and propagation (PROP), and finally after elimination, propagation, and combination (COMB) are shown. These times are normalized with respect to the program execution time without range checks. As can be seen from the results the overhead due to range checks is drastically reduced. Around 25% reduction in code size also resulted after range check optimization. As shown in section 2 the size of the MCFG is half that of the CFG in most cases. Thus the optimizations can be performed efficiently using the MCFG. The percentage of monotonic definitions is high and therefore greater degree of optimization is achieved by taking advantage of monotonicity.

	NOPT	*ELIM*	*PROP*	*COMB*
FFT	2.00	1.60	1.31	1.26
MATMUL	1.87	1.45	1.00	1.00
PERM	2.33	1.33	1.22	1.22
QUEEN	1.78	1.70	1.51	1.51
QUICK	3.25	1.98	1.46	1.46
TOWERS	2.40	1.40	1.40	1.40

The number of nodes in the MCFG relative to the number of the nodes in the original CFG for

some programs is shown below (SIZE). The size of the MCFG is half that of the CFG in most cases. The percentage of definitions of variables used in subscript expressions that were found to be monotonic (MDEFS) are shown below. The programs with 100% monotonic definitions only used for loop variables in their subscript expressions. The non-monotonic definitions were mostly initializations which are often constants and therefore can be checked at compile-time.

	SIZE	MDEFS
FFT	58%	56%
MATMUL	92%	100%
PERM	40%	100%
QUEEN	38%	40%
QUICK	57%	71%
TOWERS	42%	20%

In this paper optimizations for reducing the number of range checks that have to be performed to detect array bound violations were presented. These optimizations are performed efficiently using a smaller MCFG. Only partial *use-def* information is computed. Furthermore, the use of *monotonicity* makes the algorithms presented in this paper more effective than previous algorithms. The elimination and propagation optimizations are also effective in optimizing range checks for dynamic arrays. Substantial reduction in run-time overhead results after the range check optimizations are applied.

Acknowledgements - I am grateful to Mary Lou Soffa and the referees for their suggestions in improving this paper.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
2. D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural Constant Propagation," *Proceedings 14th ACM Symposium on Principles of Programming Languages*, pp. 152-161, 1986.
3. F. Chow, "A Portable Machine-independent Global Optimizer - Design and Measurements," Ph.D. Thesis, Technical Report 83-254, Computer Sytems Lab, Stanford University, Dec., 1983.
4. W. Harrison, "Compiler Analysis of the Value Ranges for Variables," *IEEE Transactions on Software Engineering*, vol. 3, no. 3, pp. 243-250, 1977.
5. J. Hennessy, "Symbolic Debugging of Optimized Code," *Trans. on Programming Languages and Systems*, vol. 4, no. 3, pp. 323-344, July, 1982.
6. N. Suzuki and K. Ishihata, "Implementation of Array Bound Checker," *Proceedings 4th ACM Symposium on Principles of Programming Languages*, pp. 132-143, 1977.
7. M.N. Wegman and F.K. Zadeck, "Constant Propagation with Conditional Branches," *Proceedings 12th ACM Symposium on Principles of Programming Languages*, pp. 152-161, 1984.
8. V. Markstein, J. Cocke, and P. Markstein, "Optimization of Range Checking," *Proceedings of SIGPLAN'82 Symposium on Compiler Construction*, 1982.