Optimizing Array Bound Checking During Trace-Based Compilation

Muzeich Baig
Faculty Honors Advisor: Dr. Michael Franz
University of California, Irvine

June 1, 2007

Abstract

This paper focuses on minimizing the number of lower bound and upper bound checks in a trace based just-in-time compiler. Using the advantages provided by trace-based compilation, we are able to show that we can eliminate many instances of bounds checking while still running test programs safely.

Background

Array bounds checking is a built-in feature in Java. Virtual machines implementing Java must guarantee it. However, it can be expensive to perform, especially in loops that involve multiple array accesses. For each array access, at least three checks are required: null check, lower bound check, and upper bound check. Normally, the Java virtual machine implements the bound checks in the interpreter. However, a just-in-time compiler must also implement or otherwise guarantee the bound checks in the compiled machine code.

A just-in-time (JIT) compiler is used to convert bytecode to native code at runtime. A mixed mode JIT compiler starts off in interpreter mode. When it detects a section of code with high activity (e.g. a loop or method), it compiles the section of code and switches to native mode. Among the methods of compilation is the trace based compilation method developed by Gal, et al [1].

Trace-based compilation of code has several advantages. First, the code is linear; branches are either side exits that return the compiler to interpreter mode, or branch off into another linear section of the trace tree. The branches only share code "upstream"; downstream code is linear. This simplifies code analysis tremendously and allows one to do optimizations that are not easily done otherwise. It allows easier analysis than traditional range checking techniques, where branching code can cause complications (see [2] for examples of the complications).

Second, the compiler knows exactly which variables are modified and which remain constant throughout the trace. Since the compilation is done at run-time, the compiler also knows the values of all variables. This will be important later.

Third, any exit out of the trace returns the virtual machine to interpreter mode, wherein it interprets the original bytecode. Thus, one can modify the trace (for example, one can tighten a loop condition), without modifying the semantics of the code since the compiler will return to interpreter mode which will continue from where the compiled code left off.

Bounds Checking

Our method simplifies bounds checking in a trace. The method works as long as each array variable is fixed to a particular array and the maximum number of loop iterations can be computed correctly.
1. The trace is recorded and converted into a variant of Static Single Assignment form called Trace Static Single Assignment (TSSA) as per Gal, et al. [1]. TSSA is discussed below.
2. Each instruction is marked as either a constant or non-constant. If it is a constant, the value it represents is stored within the intermediate representation.
3. The array bound checks are inserted and “normalized”. Redundant and implied checks are removed.
4. The number of loop iterations is computed along with the extreme values of a variable involved in an array access.
5. The remaining boundary checks are checked if they will ever be violated; if not, they are removed.

Some Definitions

\[ \text{min}(a) - \text{the minimum value of an index into array } a. \]
\[ \text{max}(a) - \text{the maximum value of an index into array } a \text{ minus one.} \]
\[ \text{endValue}(x) - \text{the value of } x \text{ at the end of the current trace recording session.} \]
\[ \text{curvValue}(x) - \text{the value of } x \text{ at the current instruction.} \]
\[ \text{step}(x) - \text{the step increment of } x \text{ in each iteration of the loop it belongs to} \]
\[ \text{maxValue}(x), \text{minValue}(x) - \text{the maximum and minimum values } x \text{ will take on during the course of the loop} \]
\[ \text{valueAtTopOfLastIteration}(x) - \text{the value of } x \text{ just prior to beginning the final iteration of its loop. Equivalent to endValue}(x) \text{ in the second last iteration of the loop.} \]
\[ \text{isConstant}(i) - \text{indicates whether instruction } i \text{ computes a constant value} \]
\[ \text{value}(i) - \text{the constant value computed by instruction } i \]

Marking Constants

This is performed easily using a linear scan. We first check the variables, speculating that variables that do not change during the course of the trace are constants; this is necessary to perform the optimizations discussed in the paper. Thus, all non-phi variables are marked as constants. The values of these variables are stored in the intermediate representation.

Once the variables have been checked, we check each instruction, beginning from the top of the trace. If all of a non-branching instruction’s parameters are constants, then that instruction represents a constant as well, which is computed and stored.

Inserting Array Bound Checks

Array accesses involve an array variable and index. To take advantage of common subexpression elimination, typically an array access instruction is subdivided into several intermediate representation instructions for address computation followed by a load or store instruction using the address.

The address is computed by multiplying the index by the width of the array and then adding the base address of the array. For example:

\[ i = a[j]; \quad // \text{a is an integer array, } j \text{ is an integer} \]
The array load instruction will typically consist of the following three instructions:

\[
\begin{align*}
  d_0 &= \text{SHL } j, 2 \quad /\text{ same as multiply } j \text{ by } 4 \text{ since an integer is } 4 \text{ bytes} \\
  d_1 &= \text{ADD } d_0, \text{base}(a) \\
  i_0 &= \text{ALOAD } d_1
\end{align*}
\]

Two bound checks must be inserted prior to the address computation. If any of these checks fails, the program exits from the trace and returns control to the interpreter. (This paper does not discuss null checks, which have to be handled separately).

\[
\begin{align*}
  \text{LBCHECK } j, \text{min}(a) \\
  \text{UBCHECK } j, \text{max}(a) - 1
\end{align*}
\]

\text{LBCHECK} is the lower bound check, which ensures that \( j \geq \text{min}(a) \).
\text{UBCHECK} is the upper bound check, which ensures that \( j \leq \text{max}(a) - 1 \).

\text{min}(a) and \text{max}(a) are replaced by the actual bounds of array \( a \). These are available at run time.
The assumption here is that \( a \) always refers to the same array in the trace. If \( a \) switches it may be possible to construct another trace with the other value of \( a \) and construct a trace tree. However, that is not discussed in this paper.
If \( a \) refers to a multidimensional array, it is assumed to be rectangular and the size of each dimension is assumed to remain constant. A check is inserted to ensure that the array is rectangular; if not, a side exit occurs.

\text{TSSA “Untwirling”}

In TSSA/TTSSA (Trace Static Single Assignment/Trace Tree Static Single Assignment) form, each assignment to a variable results in a new variable. An example is shown in figure 1.

<table>
<thead>
<tr>
<th>Code</th>
<th>TSSA Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 0; )</td>
<td>( x_0 = 0 )</td>
</tr>
<tr>
<td>( x = x + 1; )</td>
<td>( x_1 = x_0 + 1 )</td>
</tr>
<tr>
<td>( y = x \times 2; )</td>
<td>( y_0 = x_1 \times 2 )</td>
</tr>
</tbody>
</table>

\text{Figure 1} Java code and corresponding TSSA form.

If we start from a use of a variable (e.g. the use of \( x_1 \) in defining \( y_0 \)) and traverse “upwards”, we will encounter the uses before the definition; after encountering the definition, no further use of the variable will be found. Thus, we can “untwirl” a variable by traversing upwards and replacing each use with its definition, eventually obtaining an expression in terms of the initial variable.

For example, we can obtain \( y_0 = (x_0 + 1) \times 2 = (1 + 1) \times 2 = 4 \)

\text{Normalization of Bound Checks}

To allow different bound checks to be easily compared, they are normalized. First, the expression for the index is untraveled to obtain an expression in the form of:

\[
\text{index} = x_1 \times c_1 + x_2 \times c_2 + \ldots + x_n \times c_n + d
\]

where \( x \)'s are variables, \( c \)'s are constant coefficients, and \( d \) is a constant.
The constant \( d \) is then moved to the left sides of both inequalities above.

Thus,

\[
x_1 c_1 + x_2 c_2 + \ldots + x_n c_n + d \geq \min(a) \quad \text{becomes} \\
x_1 c_1 + x_2 c_2 + \ldots + x_n c_n \geq \min(a) - d
\]

Similarly,

\[
x_1 c_1 + x_2 c_2 + \ldots + x_n c_n + d \leq \max(a) - 1 \quad \text{becomes} \\
x_1 c_1 + x_2 c_2 + \ldots + x_n c_n \leq \max(a) - 1 - d
\]

Next, the inequalities are simplified by reducing coefficients using their greatest common divisors. In case there is only one variable, the coefficient is moved to the right side to obtain an expression in terms of the variable alone.

For example,

\[
x_1 c_1 + d \leq \max(a) - 1 \quad \text{becomes} \\
x_1 \leq \left( \frac{\max(a) - 1 - d}{c_1} \right)
\]

The variables must have the following properties:

- They must be monotonically increasing or decreasing.
- Their increment or decrement (step) must be calculable and constant.

Note that the constant \( d \) may represent a variable in the actual code that does not change during the course of the trace. An example is provided in Figure 2.

```c
int b = 50;
for ( int i = 0; i < 100; i++ )
    a[i] = a[i - b];
```

Figure 2. Here, we would have \( i - b \geq 0 \) or \( i - 50 \geq 0 \) which normalizes to \( i \geq 50 \).

Normalization allows us to detect and remove redundant or implied checks. Consider two upper bound checks with the same set of variables and coefficients:

\[
x_1 c_1 + x_2 c_2 + \ldots + x_n c_n \leq L_1 \\
x_1 c_1 + x_2 c_2 + \ldots + x_n c_n \leq L_2
\]

The only difference between the two bound checks is the \( L \) constant in both. Note that if \( L_1 \) happens to be the smaller constant, then if the expression on the left satisfies the condition, it will satisfy any condition with a larger expression on the right. As such, there is no need to maintain the second bound check. Thus, whenever we encounter a pair of such upper bound checks in a trace, we keep the check with the smaller value of \( L \) and move it “upstream” (i.e. in place of the bound check that was higher up in the trace). The other bound check is removed. This is similar to local range check optimization described in [2]. Note that replacing the highest bound check is necessary to ensure that the array access that the former was guarding is still guarded. Generally, this is safe, as it potentially forces an earlier side exit than would have occurred otherwise. However, in case another trace used the same expression with a larger \( L \) constant (i.e. a less
constricted upper bound), it is possible that the compiled trace prematurely exits to the interpreter, affecting performance.

Similarly, in the case of two similar lower bound checks, we retain the bound check with the largest value of the constant on the right hand side. (Note that the constants need not be zero, despite the fact that the minimum array index in Java is zero. See figure 2 for an example.)

Note that in both cases, the bound checks that we compare need not refer to the same array. In fact, they need not be bound checks at all; we can apply the same to loop conditions. This is what makes our method more general than that in [2]. If a bound check is found to be "tighter" than a loop condition, we can update the loop condition to match the bound check and remove the bound check. Similarly, if a loop condition is "tighter" than the tightest bound check involving the same expression, we can remove the bound check. This modification of the loop condition is usually not possible in static compile-time analysis, since only one copy of the code can exist there, but can be done in a trace based just-in-time compiler in the native code, since the interpreter retains the original code.

A note on updating loop conditions or loop guards. In a single trace, tightening a loop guard will result in an exit to the interpreter, which runs the original loop. However, in a trace tree (e.g. of a nested loop), tightening a loop guard may result in the native code prematurely exiting a deeply nested loop and returning to an outer loop. In this case the code will continue to run smoothly, but incorrectly instead of exiting back to the interpreter. To overcome this we remove all other branches that are attached to the loop condition from the tree, forcing a return to the interpreter once the loop terminates.

After computing the normalized bound checks and removing redundant loop checks, we turn to eliminating the remaining bound checks. First, we check whether the bounds will ever be violated during the course of the loop. This requires us to compute the extreme values of the variables. Since we assume that the variables are changing at a constant rate, the extreme values are:

\[
\begin{align*}
\text{max} & (\text{current value, ending value}) \\
\text{min} & (\text{current value, ending value})
\end{align*}
\]

**Step Computation**

To obtain the ending value, we compute the step of a variable. Step computation begins from the last renamed value of a variable in TTSSA form and recursively traversing the parameters of an instruction "upstream" until we hit an instruction that only uses the phi-function for the variable and a constant as parameters. The value of the step is the net change (by adding and subtracting constants).

Each instruction is of the form

\[ \text{op} \ (a), \ (b) \]

where \text{op} is an operation of the types \text{add} or \text{subtract} and \(a\) and \(b\) are parameters.

At each level of recursion, at least one parameter must be a constant or refer to a constant instruction. This requirement ensures that the variable is inductive and monotonic. The algorithm is illustrated in pseudocode below:
Algorithm:

bool computeStep(variable v, trace t)
{
  instruction i = (last definition of v in t)
  return computeStepFrom( i, v ); // the step is stored in step(v).
}

bool computeStepFrom( instruction instr, variable v )
{
  a = param1(instr)
  b = param2(instr)
  op = opcode(instr)

  if ( op is PHI or op is READ ) then
    if variable(instr) is v then
      return true;
    else
      return false;
  else if ( op is ADD ) then
    if ( isConstant(a) ) then
      step(v) += value(a);
      return computeStepFrom( b, v );
    else if ( isConstant(b) ) then
      step(v) += value(b);
      return computeStepFrom( a, v );
    else
      return false;
  else if ( op is SUB ) then
    if ( isConstant(b) ) then
      step(v) -= value(b);
      return computeStepFrom( a, v );
    else
      return false;
  else
    return false;
}

Note that in subtraction, the constant must be on the right side. Otherwise, the sign of the
variable will continuously fluctuate, which makes it non-monotonic.

Number of Remaining Iterations

The next step to obtaining the ending value of a variable is to compute the number of remaining
iterations of the loop in which the variable is updated. First, we normalize the loop condition.

Assume we have a comparison of the form:

\[ x_1 c_1 + x_2 c_2 + \ldots + x_n c_n \text{ comp } L \]

where \( x \)'s are variables, \( c \)'s are coefficients, \( L \) is the loop limit, and \text{ comp } \text{ is the comparison,}
which could be of types \( <, >, \leq, \) or \( \geq \). The variables must be incremented within the loop.
The current value is computed by substituting the values of the variables:

\[ \text{currValue} = \text{currValue}(x_1) \cdot c_1 + \text{currValue}(x_2) \cdot c_2 + \ldots + \text{currValue}(x_n) \cdot c_n \]

If the current value already violates the bounds, trace recording is aborted.

The net change in the value of the expression is the sum of the products of the individual steps of the variables and their coefficients:

\[ \text{netStep} = \text{step}(x_1) \cdot c_1 + \ldots + \text{step}(x_n) \cdot c_n \]

The maximum value of the expression is \[ \max \{ \text{currValue}, \text{currValue} + \text{netStep} \cdot \text{numberOfIterationsRemaining} \} \]. Similarly, the minimum value of the expression is \[ \min \{ \text{currValue}, \text{currValue} + \text{netStep} \cdot \text{numberOfIterationsRemaining} \} \].

If the maximum value satisfies the upper bound check, we eliminate the bound check. Similarly, if the minimum value satisfies the lower bound check, we eliminate the lower bound check.

If, however, a potential violation is detected, we compute the number of iterations that can be safely executed before the value of the expression crosses the bounds. For example, consider an upper bounds check. The inequality in a future iteration (numbered \( y \)) is given as:

\[ \text{currValue} + y \cdot \text{netStep} \leq L, \text{ where } L \text{ is the limit.} \]

Thus the number of iterations that can be safely executed is:

\[ y \leq \text{floor} \left( \frac{L - \text{currValue}}{\text{netStep}} \right) \]

We convert the loop condition for the current loop to ensure that a maximum of \( y \) iterations are executed before control returns to the interpreter, and we remove the bounds check. (If the loop condition was already more restrictive, we do not modify it.) The loop condition is now guaranteed to guard the array access.

**Variables in Different Nesting Levels**

When the code contains nested loops, a trace recorder typically first finds the deepest nested loop. Less deeply nested loops are discovered later through side exits off the former. Thus, a trace tree typically flips a nested loop "inside-out" [1].

For each upper bound check inequality, we substitute the maximum values of \( x_1 \ldots x_n \). If the inequality is satisfied, then we are guaranteed that the bound check will never be violated and thus can remove it safely. Similarly, we substitute the minimum values into each lower bound inequality and remove those that are not violated. For the remaining bound checks, we simplify the tests to determine which variable plays the largest role in causing the violation.

We order the variables in order of nested loops, keeping the deepest variable last in the expression. The variable that is updated in the most recently added trace is considered belonging to the outermost loop and so on. Next, we try to locate the troublesome variable starting with \( x_n \) (the variable updated in the deepest nested loop and thus typically updated the most often).
same nesting level and others belonging to different levels. Such a situation can be handled by treating each set of variables that belong to the same nesting level as a single variable whose current value, step, maximum, and minimum values are computed as described in the prior section.

Case Analyses

We applied several of the Java Grande Benchmarks (Section 2). While unable to test them using a running program, we simulated them on paper. Here are some results.

FFT – Fast Fourier Transform

The most loop-intensive function in this program is transform_internal() function. This function has a triply nested loop. The outermost loop runs rarely compared to the inner loops. The innermost loop runs the most. A schematic of the code is provided in figure 3.

```java
for (int bit = 0, dual = 1; bit < logn; bit++, dual *= 2) // Loop 4
{
    for (int b = 0; b < n; b += 2 * dual) // Loop 1
    {
        {contains array accesses of the forms data[2 * b], data[2 * b + 1],
        data[2 * (b + dual)], data[2 * (b + dual) + 1]}
    }
    for (int a = 1; a < dual; a++) // Loop 3
    {
        ... 
        for (int b = 0; b < n; b += 2 * dual) // Loop 2
        {
            {contains array accesses of the form data[2*(b+a)] and data[2 *
            (b + a+ dual)]}
        }
    }
}
```

**Figure 3.** Schematic of transform_internal() function. The loops are numbered in order of discovery by the trace recorder. Loop 4 is never discovered, but is labeled as such to allow ease of reference in the discussion below.
Since \( n \) is usually a very large number, our method works well. Here is a highly probable scenario. The first trace to be recorded and compiled is loop 1, shown in figure 4. The variable dual is treated as a constant. The initial bound checks are as follows: \( \text{limit} \) is the size of the array, which in the code is \( 2*n \).

\[
\begin{align*}
2*b & \leq \text{Limit} \quad (1) \\
2*b & \geq 0 \quad (2) \\
2*(b + \text{ dual}) & \leq \text{Limit} \quad (3) \\
2*(b + \text{ dual}) & \geq 0 \quad (4) \\
2*b + 1 & \leq \text{Limit} \quad (5) \\
2*b + 1 & \geq 0 \quad (6) \\
2*(b + \text{ dual}) + 1 & \leq \text{Limit} \quad (7) \\
2*(b + \text{ dual}) + 1 & \geq 0 \quad (8)
\end{align*}
\]

There are two instances of each bound check, making a total of 16 bound checks per iteration of the loop. The duplicate ones are eliminated instantly.

The first step is normalization:

\[
\begin{align*}
b & \leq \text{Limit}/2 \quad (1') \\
b & \geq 0 \quad (2') \\
b & \leq \text{Limit}/2 - \text{ dual} \quad (3') \\
b & \geq -\text{ dual} \quad (4') \\
b & \leq (\text{Limit} - 1)/2 \quad (5') \\
b & \geq -1/2 \quad (6') \\
b & \leq (\text{Limit} - 1)/2 - \text{ dual} \quad (7') \\
b & \geq -1/2 - \text{ dual} \quad (8')
\end{align*}
\]

Using what we know about the code, we can make the following assumptions about the runtime values of the variables for the paper analysis:

- \( \text{limit} = 2*n \)
- \( \text{ dual} \geq 1 \)
Thus, (3) is tighter than (1) and (5), (7) is tighter than (3), (2) is tighter than (4), (6), and (8). The tighter checks cancel the others, which are removed. We are left with:

\[ b \leq \frac{\text{Limit} - 1}{2} - \text{dual} \quad (7') \]
\[ b \geq 0 \quad (2') \]

The variable \( b \) is increasing positively, and is most likely already greater than zero, so (2) is removed easily. Since \( n = \text{limit} / 2 \), the loop condition \( (b < n) \) is somewhat less tight than the condition computed in (7). Thus, the condition in (7') replaces the loop condition and we have removed all bound checks.

![Figure 5. Trace recorded for loop 2. All variables other than b are constant.](image)

![Figure 6. Trace record for loops 2 and 3. Note that variable \( a \) is no longer a constant. However, it is bounded by dual, which is constant.](image)

Once the code exits from loop 1, the trace recorded attempts to record for a while but disengages when the interpreter encounters loop 2. This loop is eventually discovered on its own as a location of hot code and is recorded (see figure 5). The variables \( a \) and \( \text{dual} \) are again treated as constants, since they are not modified during the first instance. The analysis of the bound checks is similar to the one we did above, and all bound checks are removed.

Once the control returns to the interpreter via a side exit from the trace for loop 2, it will attempt to record the bytecode again. In the first iteration of loop 2, the outer loop (loop 3) runs only once so the interpreter returns to the top of loop 4, after which it re-encounters loop 1.

At this point, the value of \( \text{dual} \) has now been updated. Thus, the old code that had been recorded is no longer valid. The trace must be re-recorded and compiled. While this adds overhead, it is still far more efficient than leaving in the original bound checks.

After this trace exits, we again return to the trace for loop 2. This code has to be recompiled as well because the values of \( a \) and \( \text{dual} \) have changed. After finishing, it returns to Loop 3. At this point, the top of loop 3 is added as a branch to the existing trace and the trace becomes a trace tree (see figure 6). Variable \( a \) can no longer be treated as a constant, as it is updated in the trace. However, \( \text{dual} \) remains a constant.

The bound check involving the variables \( b \) and \( a \) is of the form:
2 \cdot (b + a + \text{dual}) \leq \text{Limit} \quad (9)

(The other bound check only involves b and is not interesting for the purpose of this discussion).

This is normalized to:

\[ b + a \leq (\text{Limit} - \text{dual}) / 2 \quad (9') \]

and reordered to:

\[ a + b \leq (\text{Limit} - \text{dual}) / 2 \quad (9') \]

Now, we compute the maximum values for a and b:

\[
\text{maxValue}(a) = \text{dual} - 1.
\text{maxValue}(b) = n / (2 \cdot \text{dual})
\]

At this point, \text{dual} \approx 2.
So, \((\text{dual} - 1) + \left(\frac{n}{2 \cdot \text{dual}}\right) = 2 - 1 + n/(2\cdot2) = 1 + n/4\)

Check:

1 + n/4 \leq (\text{limit} - \text{dual}) / 2 \quad ?
1 + n/4 \leq (2\cdot n - 2) / 2 \quad ?
1 + n/4 \leq n - 1 \quad ?
2 \leq 5n/4 \quad ?
8/5 \leq n \quad ?

Yes, as long as n is at least 2, which is almost always the case. Bound (9') can thus be removed here as well. Again, there are no remaining bound checks.

After we exit from the trace tree and resume the next iteration of loop 4, we will have to recompile the trace tree since the value of dual will have changed. However, all bound checks will again be removed.

If we continue in a likewise manner, we can show that the bound checks are always removed for the loops in the transform_internal() method of FFT. While this requires recompilation of traces for loop 1 and the trace tree for loops 2 and 3 in each iteration of loop 4, the overhead of recompilation is very small compared to the overhead due to the bound checks. Loop 4 iterates \(\log(n)\) times, which is negligible compared to the number of cycles that loops 1, 2, and 3 use.

**SOR – Successive OverRelaxation Method Algorithm**

The loop-intensive function in this benchmark is `sor_run()`. There are two sets of nested loops in the function. The more intensive loop is displayed in figure 7.

```c
for (int p=0; p<num_iterations; p++) // Loop 3
{
    for (int i=1; i<Nml; i++) // Loop 2
    {
        double [] Gi = G[i];
        double [] Gml = G[i-1];
        double [] Gip1 = G[i+1];
```
for (int j=1; j<Nnl; j++) // Loop 1
    Gi[j] = omega_over_four * (Gim1[j] + Gip1[j] + Gi[j-1]
    + Gi[j+1]) + one_minus_omega * Gi[j];

Figure 7. One set of nested loops in array-intensive SOLrun() function.

Figure 8. One plausible trace tree for the set of nested loops in SOLrun(). Note that some code is repeated in the right-most branch and its neighbor. This is normal in trace recording. Note also that the step of i is the same in all paths that use i. This enables us to compute the maximum value of i.

This is a triply nested loop, with all loop conditions depending on variables defined outside the loops. This nice property of the loops protects us from having to recompile the trace every time we iterate around loop 3.

One plausible trace tree is shown in figure 8. First, loop 1 is detected and a trace is recorded for it. Array G is multidimensional, assumed to be rectangular. Thus, the dimensions of Gim1, Gip1, and Gi are the same and are fixed for each value of i.

The bound check inequalities are straightforward:
\[
\begin{align*}
j & \leq \max(Gi) - 1 \\
j & \geq 0 \\
j & \leq \max(Gip1) - 1 \\
j - 1 & \leq \max(Gi) - 1 \\
j - 1 & \geq 0 \\
j + 1 & \leq \max(Gi) - 1 \\
j + 1 & \geq 0 \\
j & \leq \max(Gi) - 1
\end{align*}
\]

Again, we normalize and eliminate redundant checks. We are left with:
\[
j \leq \max(Gi) - 2
\]
$j \geq 1$

When we analyze the code by hand (to obtain the values for the constants), it turns out that \( \max(G_l) = \text{Nml} + 1 \). Thus, the first bound check translates into \( j \leq \text{Nml} - 1 \).

Upon checking the loop conditions we obtain the following extrema for \( j \):

- \( \minValue(j) = 1 \)
- \( \maxValue(j) = \text{Nml} - 1 \).

Note that these values are identical to the lower and upper bounds of \( j \), respectively. Thus, we can remove both bound checks.

After we exit from the trace, we encounter loop 2 again. Its code is recorded and added to the trace, forming a trace tree. Since the sizes of the arrays remain the same and no variable external to the inner loop is used in the array indices, there is no need to insert bound checks again. Further, we can ensure that the array sizes will always remain the same, by inserting checks after the array assignments in loop 2. Finally, the outermost loop is added to the trace and the entire set of loops is compiled.

Further Work

The above method has been tested on code with simple loops that generate single traces. The method for trace trees and nested loops was simulated on test programs.

The method described above does not work for ragged arrays nor for array variables that switch arrays. Further, loop variables that are multiplied by a constant factor (e.g. doubled in each iteration) need to be explored. It is possible to analyze the maximum values of such variables, but in case there is a bounds violation, it is sometimes difficult to compute the number of iterations that can be executed safely before exiting to the interpreter.

Summary

Using trace-based compilation and traditional array range check optimizations, we have been able to effectively decrease the number of bound checks required without adding much overhead. We are able to both modify loop guard conditions as well as compute precisely at runtime how many iterations of a loop remain (as long as the loop does not have any premature breaks). Further, the analysis is simpler because it only focuses on the code that is recorded due to its high activity.

References