Lecture Notes on
Array-Bounds Checks

15-411: Compiler Design
André Platzer

Lecture 19
October 28, 2010

1 Introduction

More information can be found in [App98, Ch 18.4].

2 Array-Bounds Checks

In unsafe languages like C, the compiler does not have to take any precautions against array access out of bounds, because C does not define what happens in that case. Thus, it's not the compiler's fault if weird things happen, and every implementation is automatically declared to conform to the C standard.

For type-safe programming languages like C0, array access must be guarded with array-bounds checks in order to make sure that the implementation complies with the type-safety requirements. Array access out of bounds and null pointer dereferences are essentially the only parts of C0 that need special attention, since overflows are taken care of by the CPU.

While C0 programmers will appreciate the reduced annoyance coming from reliable type-compliance, they still have to pay the price of increased execution overhead coming from the dynamic array checks. This effect is particularly drastic in JVM implementations. It might sound tempting to rely on turning off runtime checks when the program is delivered. But that would require a lot more quality assurance than is usually done, because production-quality programs still contain bugs.

Yet a smart compiler can take care of the array-bounds checks and optimize some of those away. Either optimize them away entirely if the array declaration can be found locally, or at least use loop-invariant code motion to optimize repeated array-bounds checks away.

The easiest and most common instance of loop-invariant code motion for array access is the following.

```
for (int i = L; i < U; i++) {
  check(a,i)
  use a[i]
  // no definitions of i or a or U
}
```

We explicitly write $check(a, i)$ here for the action of checking if the array access $a[i]$ is safely within the array bounds. We may have two goals with array bounds checks. Either moving them out of the loop to reduce the number of times that array bounds need to be checked during a program run, or removing the array bounds checks altogether, because its outcome can be determined statically. By moving the array access check out of the loop, the above program can be optimized to the following program that has no array access checks within the loop body anymore:

```
check(a,L)
check(a,U-1)
for (int i = L; i < U; i++) {
  use a[i]
  // no definitions of i or a or U
}
```

The reason why this is okay is that i changes monotonically from L to U. Here it even changes linearly because i is an induction variable. The other condition is that the array that i refers is loop-invariant and U is loop-invariant. A third condition is that only the increments by 1 justify that $a[U - 1]$ will actually be the last array access requiring bounds checks. That could be a different element if the loop step is i+=2.

More generally, what are the conditions that we need to ensure in order for this array-bounds check optimization to work? Consider a loop in intermediate representation. One option is to require the following conditions:

1. There is a loop end check statement. That is, there is an induction variable j and a loop-invariant U at a node $L1$ of the form (or similar forms like $U \leq j$)

   ```
   L1: if (j ≥ U) goto Lexit
   ```

   where Lexit is out of the loop and U loop-invariant.

2. There is a statement $L2$ that checks the range of an induction variable k, e.g., compared to an array size N. That is a statement of the form

   ```
   L2: if (0≤ k<N) goto L3 else goto L4
   ```

   where N is loop-invariant and $L1 < L2$ (dominates), and k is an induction variable *coordinated with* j. That is, $(k - b_k)/a_k = (j - b_j)/a_j$ stays true during the loop. This holds for the variables introduced by strength reduction based on the same basic induction variable; see [App98, Ch 18.3] for details.

3. k increases when j increases, i.e., their linear terms satisfy $a_j/a_k > 0$. Where the induction variables satisfy $j = a_j * i + b_j$ and $k = a_k * i + b_k$. A similar optimization applies for $a_j/a_k < 0$.

4. No nested loop defines k.

Under which circumstances can we remove the check in L2? First, consider the lower bound part $0 \leq k$ of the check. We can easily remove the $0 \leq k$ check if the initial value k_0 of k before the loop is $k_0 \geq 0$ and all (loop-invariant) increments c_k (for $k = k + c_k$) of k in the loop body are $c_k \geq 0$. Because then, only non-negative values can be added to k (if it is a basic induction variable), which will stay non-negative if it has been initially. If k is a derived induction variable (by $k = a_k * i + b_k$) from the (say basic) induction variable i, then this increment principle translates to: all increments $i = i + c_i$ have a sign such that $a_k * c_i \geq 0$.

If these are constants, the check can be performed at compile-time by constant folding. Otherwise, the check can be turned into a runtime check for non-negativity of $k_0 \geq 0$ and $c_k \geq 0$ for all increments in the loop prefix. In some cases, this check may fail even if the program would otherwise run normally, just because the negative increments are unreachable. This aggressive optimization thus needs to fall back to the old loop with check statements in the loop body if the check in the loop prefix fails.

How can we remove the check $k < N$ in L2? What we can use is the knowledge that $j < U$ has been true after L1 (unless the jump left the loop, which does not affect the loop body statements). Since j is coordinated with k, we can translate this to a bound on k. The variables are coordinated by $(k - b_k)/a_k = (j - b_j)/a_j$. But, in addition, k might have changed since L1. If it cannot change by too much and we manage to relate U to N, then we will be fine. Let K be the sum of all (loop-invariant) increments that are added to k on any path between L1 and L2 that does not pass through L1 twice (note that k is not redefined in any nested loop). We want to make sure that $k < N$ holds at L2 so that this range check is unnecessary. If we knew that $k < N - K$ at L1, then we also knew $k < N$ at L2, because K is an upper bound on the maximum increment from L1 to L2. Because k is coordinated with j, i.e., $(k - b_k)/a_k = (j - b_j)/a_j$, the test $k < N - K$ at L1 is equivalent to

$$(j - b_j) * a_k/a_j + b_k < N - K$$

This, in turn, is equivalent to

$$j < a_j/a_k * (N - K - b_k) + b_j$$

if j increases when k increases, i.e., $a_j/a_k > 0$ (otherwise the $<$ inequality flips to a $>$ inequality if $a_j/a_k < 0$). Since $j < U$ is checked at L1 and the loop exits if this fails, we know that $k < N$ will be true at L2 if

$$U \leq a_j/a_k * (N - K - b_k) + b_j$$

This test only consists of loop-invariant expressions. It can thus either be evaluated statically (if sufficiently many parts of it are constant) or it can be checked in the loop prefix.

Another common special case is when N and U are the same variable (e.g., array length), $K = 0$, and the induction variables coevolve, because $a_j = a_k$ and $b_j = b_k$. Then the test is trivially true.

In general, for removing array bounds checks, it makes sense to track the possible variable ranges.

3 Loop Unrolling

Another useful loop optimization is loop unrolling, which unrolls the loop body k times. In addition to the reduced amount of time that may be wasted with program counter jumps, loop unrolling may also enable smarter scheduling of instructions. Consider, how instructions may be scheduled for a program computing the scalar product of two vectors a and b.

```
   r = 0
   i = 0
L: t1 = 4*i
   i = i + 1
   ta = M(a + t1)
   tb = M(b + t1)
   nop              // wait for load
```

```
  p = ta*tb
  if (i<n) goto l  // already start executing, done later
  nop
  r = r + p
```

The loop needs 9 cycles to execute. After loop unrolling once, we obtain the following instruction schedule

```
  r = 0
  i = 0
L:t1 = 4*i
  j = i + 1
  t1' = 4*j
  ta = M(a + t1)
  tb = M(b + t1)
  ta' = M(a +t1')
  tb' = M(b +t1')
  p = ta*tb
  p' = ta'*tb'
  i = i + 2
  r = r + p
  if (i<n) goto l  // already start executing, done later
  r = r + p'
  nop
```

This loop needs 14 cycles to execute, but, of course, its progress is twice as fast, because it only needs half the number of loop iterations than before. Note that we also need extra postprocessing for odd n after the loop, but not within the loop.

Loop unrolling just works by copying the loop body and changing the back edges in the first body to jumps to the second body. Yet loop unrolling is not always useful; see Figure 1.

If, instead, we know that i is a (basic) induction variable, and that every increment $i = i + c_i$ of i strictly dominates all loop back edges, then we know that all of those increments will be executed in every loop repetition. Let C be the sum of all these loop increments. Then we know that each loop body execution that leads to a repetition will increase i by exactly C (assuming no inner loop changes i). Hence, we can adapt the loop repetition check by multiples $r * C$ of C when unrolling r times.

For example, see the following optimization of the program in Figure 1, where $C = 4$ and $r = 2$. The loop suffix starting at L2 takes care of odd iterations of the loop (loop peeling). In fact, the last instruction could even be

```
L1: x = *i
  s = s + x
  i = i + 4
  if (i<N) goto L1 (else goto L2)
L2:
```

unrolling still has the same number of branchings and edges:

```
L1: x = *i
  s = s + x
  i = i + 4
  if (i<N) goto L1 (else goto L3)
L3: x = *i
  s = s + x
  i = i + 4
  if (i<N) goto L3 (else goto L2)
L2:
```

Figure 1: Pretty pointless loop unrolling

removed with some arithmetic reasoning, because we have only unrolled once here.

```
if (i<N-8) goto L1 else L2
L1: x = *i
  s = s + x
  x = *(i + 4)
  s = s + x
  i = i + 8
  if (i<N-8) goto L1 (else goto L2)
L2: x = *i                    // loop peeling postprocessing
  s = s + x
  i = i + 4
  if (i<N) goto L2 (else goto L3)  // remove if r=2 unrollings
L3:
```

Simple loops can be rolled out completely. Another constraint on loop unrolling is that the resulting loop body still needs to fit into the instruction cache.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.