Lecture Notes on
Loop Optimizations

15-411: Compiler Design
Frank Pfenning

Lecture 17
October 21, 2014

1 Introduction

Optimizing loops is particularly important in compilation, since loops (and in particular the inner loops) account for much of the executions times of many programs. Since tail-recursive functions are usually also turned into loops, the importance of loop optimizations is further magnified. In this lecture we will discuss two main ones: hoisting loop-invariant computation out of a loop, and optimizations based on induction variables.

2 What Is a Loop?

Before we discuss loop optimizations, we should discuss what we identify as a loop. In our source language, this is rather straightforward, since loops are formed with while or for, where it is convenient to just elaborate a for loop into its corresponding while form.

The key to a loop is a back edge in the control-flow graph from a node l to a node h that dominates l. We call h the *header node* of the loop. The loop itself then consists of the nodes on a path from h to l. It is convenient to organize the code so that a loop can be identified with its header node. We then write $\mathsf{loop}(h, l)$ if line l is in the loop with header h.

When loops are nested, we generally optimize the inner loops before the outer loops. For one, inner loops are likely to be executed more often. For another, it could move computation to an outer loop from which it is hoisted further when the outer loop is optimized and so on.

3 Hoisting Loop-Invariant Computation

A (pure) expression is *loop invariant* if its value does not change throughout the loop. We can then define the predicate $\mathsf{inv}(h, p)$, where p is a pure expression, as follows:

$$\frac{c\ constant}{\mathsf{inv}(h, c)} \qquad \frac{\mathsf{def}(l, x) \quad \neg\mathsf{loop}(h, l)}{\mathsf{inv}(h, x)} \qquad \frac{\mathsf{inv}(h, s_1) \quad \mathsf{inv}(h, s_2)}{\mathsf{inv}(h, s_1 \oplus s_2)}$$

Since we are concerned only with programs in SSA form, it is easy to see that variables are loop invariant if they are not parameters of the header label. However, the definition above does not quite capture this for definitions $t \leftarrow p$ where p is loop-invariant but t is not part of the label parameters. So we add a second propagation rule.

$$\frac{l : t \leftarrow p \quad \mathsf{inv}(h, p) \quad \mathsf{loop}(h, l)}{\mathsf{inv}(h, t)}$$

Note that we do not consider memory references or function calls to be loop invariant, although under some additional conditions they may be hoisted as well.

In order to hoist loop invariant computations out of a loop we should have a *loop preheader* in the control-flow graph, which immediately dominates the loop header. When then move all the loop invariant computations to the preheader, in order.

Some care must be taken with this optimization. For example, when the loop body is never executed the code could become significantly slower. Another problem if we have conditionals in the body of the loop: values computed only on one branch or the other will be loop invariant, but depending on the boolean condition one or the other may never be executed.

In some cases, when the loop guard is inexpensive and effect-free but the loop-invariant code is expensive, we might consider duplicating the test so that instead of

$$\mathsf{seq}(pre, \mathsf{while}(e, s))$$

we generate code for

$$\mathsf{seq}(\mathsf{if}(e, \mathsf{seq}(pre, \mathsf{while}(e, s)), \mathsf{nop}))$$

where *pre* is the hoisted computation in the loop pre-header.

A typical example of hoisting loop invariant computation would be a loop to initialize all elements of a two-dimensional array:

```
for (int i = 0; i < width * height; i++)
  A[i] = 1;
```

We show the relevant part of the abstract assembly on the left. In the right is the result of hoisting the multiplication, enabled because both $width$ and $height$ are loop invariant and therefore their product is.

$\quad i_0 \leftarrow 0$	$\quad i_0 \leftarrow 0$
	$\quad t \leftarrow width * height$
$\quad$ goto loop(i_0)	$\quad$ goto loop(i_0)
loop(i_1) :	loop(i_1) :
$\quad t \leftarrow width * height$	
$\quad$ if $(i_1 \geq t)$ goto exit	$\quad$ if $(i_1 \geq t)$ goto exit
$\quad \ldots$	$\quad \ldots$
$\quad i_2 \leftarrow i_1 + 1$	$\quad i_2 \leftarrow i_1 + 1$
$\quad$ goto loop(i_2)	$\quad$ goto loop(i_2)
exit :	exit :

4 Induction Variables

Hoisting loop invariant computation is significant; optimizing computation which changes by a constant amount each time around the loop is probably even more important. We call such variables *basic induction variables*. The opportunity for optimization arises from *derived induction variables*, that is, variables that are computed from basic induction variables.

As an example we will use a function check if a given array is sorted in ascending order.

```
bool is_sorted(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
{
  for (int i = 0; i < n-1; i++)
    //@loop_invariant 0 <= i;
    if (A[i] > A[i+1]) return false;
  return true;
}
```

Below is a possible compiled SSA version of this code, assuming that we do not perform array bounds checks (or have eliminated them).

```
is_sorted(A, n) :
    i0 ← 0
    goto loop(i0)
loop(i1) :
    t0 ← n − 1
    if (i1 ≥ t0) goto rtrue
    t1 ← 4 ∗ i1
    t2 ← A + t1
    t3 ← M[t2]
    t4 ← i1 + 1
    t5 ← 4 ∗ t4
    t6 ← A + t5
    t7 ← M[t6]
    if (t3 > t7) goto rfalse
    i2 ← i1 + 1
    goto loop(i2)
rtrue :
    return 1
rfalse :
    return 0
```

Here, i_1 is the basic induction variable, and $t_1 = 4 * i_1$ and $t_4 = i_1 + 1$ are the derived induction variables. In general, we consider a variable a derived induction variable if its has the form $a * i + b$, where a and b are loop invariant.

Let's consider t_4 first. We see that common subexpression elimination applies. However, we would like to preserve the basic induction variable i_1 and its version

i_2, so we apply code motion and then eliminate the second occurrence of $i_1 + 1$.

is_sorted(A, n) :	is_sorted(A, n) :	is_sorted(A, n) :
$i_0 \leftarrow 0$	$i_0 \leftarrow 0$	$i_0 \leftarrow 0$
goto loop(i_0)	goto loop(i_0)	goto loop(i_0)
loop(i_1) :	loop(i_1) :	loop(i_1) :
$t_0 \leftarrow n - 1$	$t_0 \leftarrow n - 1$	$t_0 \leftarrow n - 1$
if $(i_1 \geq t_0)$ goto rtrue	if $(i_1 \geq t_0)$ goto rtrue	if $(i_1 \geq t_0)$ goto rtrue
$t_1 \leftarrow 4 * i_1$	$t_1 \leftarrow 4 * i_1$	$t_1 \leftarrow 4 * i_1$
$t_2 \leftarrow A + t_1$	$t_2 \leftarrow A + t_1$	$t_2 \leftarrow A + t_1$
$t_3 \leftarrow M[t_2]$	$t_3 \leftarrow M[t_2]$	$t_3 \leftarrow M[t_2]$
$t_4 \leftarrow i_1 + 1$	$t_4 \leftarrow i_1 + 1$	$i_2 \leftarrow i_1 + 1$
$t_5 \leftarrow 4 * t_4$	$t_5 \leftarrow 4 * t_4$	$t_5 \leftarrow 4 * i_2$
$t_6 \leftarrow A + t_5$	$t_6 \leftarrow A + t_5$	$t_6 \leftarrow A + t_5$
$t_7 \leftarrow M[t_6]$	$t_7 \leftarrow M[t_6]$	$t_7 \leftarrow M[t_6]$
if $(t_3 > t_7)$ goto rfalse	if $(t_3 > t_7)$ goto rfalse	if $(t_3 > t_7)$ goto rfalse
$i_2 \leftarrow i_1 + 1$	$i_2 \leftarrow t_4$	
goto loop(i_2)	goto loop(i_2)	goto loop(i_2)

In the second step we applied copy propagation and then renamed t_4 to i_2 for easier reading (but not formally required).

Next we look at the derived induction variable $t_1 \leftarrow 4 * i_1$. The idea is to see how we can calculate t_1 at a subsequent iteration from t_1 at a prior iteration. In order to achieve this effect, we add a new induction variable to represent $4 * i_1$. We call this j and add it to our loop variables in SSA form.

is_sorted(A, n) :	
$i_0 \leftarrow 0$	
$j_0 \leftarrow 4 * i_0$	@ensures $j_0 = 4 * i_0$
goto loop(i_0, j_0)	
loop(i_1, j_1) :	@requires $j_1 = 4 * i_1$
$t_0 \leftarrow n - 1$	
if $(i_1 \geq t_0)$ goto rtrue	
$t_1 \leftarrow j_1$	@assert $j_1 = 4 * i_1$
$t_2 \leftarrow A + t_1$	
$t_3 \leftarrow M[t_2]$	
$i_2 \leftarrow i_1 + 1$	
$j_2 \leftarrow 4 * i_2$	@ensures $j_2 = 4 * i_2$
$t_4 \leftarrow i_2$	
$t_5 \leftarrow 4 * t_4$	
$t_6 \leftarrow A + t_5$	
$t_7 \leftarrow M[t_6]$	
if $(t_3 > t_7)$ goto rfalse	
goto loop(i_2, j_2)	

Crucial here is the invariant that $j_1 = 4 * i_1$ when label $\mathsf{loop}(i_1, j_1)$ is reached. Now we calculate

$$j_2 = 4 * i_2 = 4 * (i_1 + 1) = 4 * i_1 + 4 = j_1 + 4$$

so we can express j_2 in terms of j_1 without multiplication. This is an example of *strength reduction* since addition is faster than multiplication. Recall that all the laws we used are valid for modular arithmetic. Similarly:

$$j_0 = 4 * i_0 = 0$$

since $i_0 = 0$, which is an example of constant propagation followed by constant folding.

```
is_sorted(A, n) :
      i0 ← 0
      j0 ← 0                  @ensures j0 = 4 * i0
      goto loop(i0, j0)
loop(i1, j1) :                @requires j1 = 4 * i1
      t0 ← n − 1
      if (i1 ≥ t0) goto rtrue
      t1 ← j1                 @assert j1 = 4 * i1
      t2 ← A + t1
      t3 ← M[t2]
      i2 ← i1 + 1
      j2 ← j1 + 4             @ensures j2 = 4 * i2
      t4 ← i2
      t5 ← 4 * t4
      t6 ← A + t5
      t7 ← M[t6]
      if (t3 > t7) goto rfalse
      goto loop(i2, j2)
```

With some copy propagation, and noticing that $n-1$ is loop invariant, we next get:

$$
\begin{array}{lll}
\mathsf{is\_sorted}(A, n): & & \\
\quad i_0 \leftarrow 0 & & \\
\quad j_0 \leftarrow 0 & & \mathsf{@ensures}\ j_0 = 4 * i_0 \\
\quad t_0 \leftarrow n - 1 & & \\
\quad \mathsf{goto\ loop}(i_0, j_0) & & \\
\mathsf{loop}(i_1, j_1): & & \mathsf{@requires}\ j_1 = 4 * i_1 \\
\quad \mathsf{if}\ (i_1 \geq t_0)\ \mathsf{goto\ rtrue} & & \\
\quad t_2 \leftarrow A + j_1 & & \\
\quad t_3 \leftarrow M[t_2] & & \\
\quad i_2 \leftarrow i_1 + 1 & & \\
\quad j_2 \leftarrow j_1 + 4 & & \mathsf{@ensures}\ j_2 = 4 * i_2 \\
\quad t_5 \leftarrow 4 * i_2 & & \\
\quad t_6 \leftarrow A + t_5 & & \\
\quad t_7 \leftarrow M[t_6] & & \\
\quad \mathsf{if}\ (t_3 > t_7)\ \mathsf{goto\ rfalse} & & \\
\quad \mathsf{goto\ loop}(i_2, j_2) & &
\end{array}
$$

With common subexpression elimination (noting the additional assertions we are aware of), we can replace $4 * i_2$ by j_2. We combine this with copy propagation.

$$
\begin{array}{lll}
\mathsf{is\_sorted}(A, n): & & \\
\quad i_0 \leftarrow 0 & & \\
\quad j_0 \leftarrow 0 & & \mathsf{@ensures}\ j_0 = 4 * i_0 \\
\quad t_0 \leftarrow n - 1 & & \\
\quad \mathsf{goto\ loop}(i_0, j_0) & & \\
\mathsf{loop}(i_1, j_1): & & \mathsf{@requires}\ j_1 = 4 * i_1 \\
\quad \mathsf{if}\ (i_1 \geq t_0)\ \mathsf{goto\ rtrue} & & \\
\quad t_2 \leftarrow A + j_1 & & \\
\quad t_3 \leftarrow M[t_2] & & \\
\quad i_2 \leftarrow i_1 + 1 & & \\
\quad j_2 \leftarrow j_1 + 4 & & \mathsf{@ensures}\ j_2 = 4 * i_2 \\
\quad t_6 \leftarrow A + j_2 & & \\
\quad t_7 \leftarrow M[t_6] & & \\
\quad \mathsf{if}\ (t_3 > t_7)\ \mathsf{goto\ rfalse} & & \\
\quad \mathsf{goto\ loop}(i_2, j_2) & &
\end{array}
$$

We observe another derived induction variable, namely $t_2 = A + j_1$. We give this a new name ($k_1 = A + j_1$) and introduce it into our function. Again we just calculate:

$k_2 = A + j_2 = A + j_1 + 4 = k_1 + 4$ and $k_0 = A + j_0 = A$.

```
is_sorted(A, n) :
    i0 ← 0
    j0 ← 0                      @ensures j0 = 4 ∗ i0
    k0 ← A + j0                 @ensures k0 = A + j0
    t0 ← n − 1
    goto loop(i0, j0, k0)
loop(i1, j1, k1) :              @requires j1 = 4 ∗ i1 ∧ k1 = A + j1
    if (i1 ≥ t0) goto rtrue
    t2 ← k1
    t3 ← M[t2]
    i2 ← i1 + 1
    j2 ← j1 + 4                 @ensures j2 = 4 ∗ i2
    k2 ← k1 + 4                 @ensures k2 = A + j2
    t6 ← A + j2
    t7 ← M[t6]
    if (t3 > t7) goto rfalse
    goto loop(i2, j2, k2)
```

After more round of constant propagtion, common subexpression elimination, and dead code elimination we get:

```
is_sorted(A, n) :
    i0 ← 0
    j0 ← 0                      @ensures j0 = 4 ∗ i0
    k0 ← A                      @ensures k0 = A + j0
    t0 ← n − 1
    goto loop(i0, j0, k0)
loop(i1, j1, k1) :              @requires j1 = 4 ∗ i1 ∧ k1 = A + j1
    if (i1 ≥ t0) goto rtrue
    t3 ← M[k1]
    i2 ← i1 + 1
    j2 ← j1 + 4                 @ensures j2 = 4 ∗ i2
    k2 ← k1 + 4                 @ensures k2 = A + j2
    t7 ← M[k2]
    if (t3 > t7) goto rfalse
    goto loop(i2, j2, k2)
```

With neededness analysis we can say that j_0, j_1, and j_2 are no longer needed and can be eliminated.

$$
\begin{array}{ll}
\mathsf{is\_sorted}(A, n): & \\
\quad i_0 \leftarrow 0 & \\
\quad k_0 \leftarrow A & \text{@ensures } k_0 = A + 4 * i_0 \\
\quad t_0 \leftarrow n - 1 & \\
\quad \mathsf{goto\ loop}(i_0, k_0) & \\
\mathsf{loop}(i_1, k_1): & \text{@requires } k_1 = A + 4 * i_1 \\
\quad \mathsf{if}\ (i_1 \geq t_0)\ \mathsf{goto\ rtrue} & \\
\quad t_3 \leftarrow M[k_1] & \\
\quad i_2 \leftarrow i_1 + 1 & \\
\quad k_2 \leftarrow k_1 + 4 & \text{@ensures } k_2 = A + 4 * i_2 \\
\quad t_7 \leftarrow M[k_2] & \\
\quad \mathsf{if}\ (t_3 > t_7)\ \mathsf{goto\ rfalse} & \\
\quad \mathsf{goto\ loop}(i_2, k_2) &
\end{array}
$$

Unfortunately, i_1 is still needed, since it governs a conditional jump. In order to eliminate that we would have to observe that

$$i_1 \geq t_0 \text{ iff } A + 4 * i_1 \geq A + 4 * t_0$$

This holds since the addition here is a on 64 bit quantities where the second operand is 32 bits, so no overflow can occur. The general case under which we can make this observation is a bit unclear. It may be one should look for induction variables that are not needed except for conditions in conditional branches (which would be the case here). Or we might make a particular effort to remove basic induction variables once derived ones have been introduced. In any case, if we exploit this we obtain:

$$
\begin{array}{ll}
\mathsf{is\_sorted}(A, n): & \\
\quad i_0 \leftarrow 0 & \\
\quad k_0 \leftarrow A & \text{@ensures } k_0 = A + 4 * i_0 \\
\quad t_0 \leftarrow n - 1 & \\
\quad \mathsf{goto\ loop}(i_0, k_0) & \\
\mathsf{loop}(i_1, k_1): & \text{@requires } k_1 = A + 4 * i_1 \\
\quad \mathsf{if}\ (k_1 \geq A + 4 * t_0)\ \mathsf{goto\ rtrue} & \\
\quad t_3 \leftarrow M[k_1] & \\
\quad i_2 \leftarrow i_1 + 1 & \\
\quad k_2 \leftarrow k_1 + 4 & \text{@ensures } k_2 = A + 4 * i_2 \\
\quad t_7 \leftarrow M[k_2] & \\
\quad \mathsf{if}\ (t_3 > t_7)\ \mathsf{goto\ rfalse} & \\
\quad \mathsf{goto\ loop}(i_2, k_2) &
\end{array}
$$

Now i_0, i_1, and i_2 are no longer needed and can be eliminated. Moreover, $A + 4 * t_0$

is loop invariant and can be hoisted.

```
is_sorted(A, n) :
    k0 ← A
    t0 ← n − 1
    t8 ← 4 ∗ t0
    t9 ← A + t8
    goto loop(k0)
loop(k1) :
    if (k1 ≥ t9) goto rtrue
    t3 ← M[k1]
    k2 ← k1 + 4
    t7 ← M[k2]
    if (t3 > t7) goto rfalse
    goto loop(k2)
rtrue :
    return 1
rfalse :
    return 0
```

It was suggested that we can avoid two memory accesses per iteration by unrolling the loop once. This make sense, but this opimization is beyond the scope of this lecture.

We have carried out the optimizations here on concrete programs and values, but it is straightforward to generalize them to arbitrary induction variables x that are updated with $x_2 \leftarrow x_1 \pm c$ for a constant c, and derived variables that arise from constant multiplication with or addition to a basic induction variable.