Lecture Notes on
Loop-Invariant Code Motion

15-411: Compiler Design
André Platzer

Lecture 17
October 21, 2010

1 Introduction

More information can be found in [App98, Ch 18.1-18.3] and [Muc97].

2 Loop-Invariant Code Motion / Hoisting

Loop-Invariant code motion is a form of partial redundancy elimination (PRE) whose purpose it is to find code in a loop body that produces the same value in every iteration of the loop. This code can be moved out of the loop so that it is not computed over and over again. It is an undecidable question if a fragment of a loop body has the same effect with each iteration, but we can easily come up with a reasonable conservative approximation.

The computation $d = a \oplus b$ (for an operator $\oplus \in \{+, -, *\}$) is loop-invariant for a loop if

1. a, b are numerical constants,
2. a, b are defined outside the loop
   (for non-SSA this means that all reaching definitions of a, b are outside the loop), or
3. a, b are loop invariants
   (for non-SSA this means that there is only one reaching definition of a, b and that is loop-invariant).

Loop-invariant computations can easily be found by collecting computations according to the three steps above repeatedly until nothing changes anymore.

If we find a loop-invariant computation in SSA form, then we just move it out of the loop to a block before the loop. When moving a (side-effect-free) loop-invariant computation to a previous position, nothing can go wrong, because the value it computes cannot be overwritten later and the values it depends on cannot have been changed before (and either are already or can be placed outside the loop by the loop-invariance condition). In fact, it's part of the whole point of SSA do be able to do simple global code motion and have the required dataflow analysis be trivial.

In order to make sure we do not needlessly compute the loop-invariant expression in the case when the loop is not entered, we can add an extra basic block like for critical edges. This essentially turns

```
j = loopinv
while (e) {
  S
}
```

into

```
if (e) {
  j = loopinv
  while (e) {
    S
  }
}
```

The transformation is often more efficient on the intermediate representation level. This, of course, depends on e being side-effect free, otherwise extra precautions have to be done.

For non-SSA form, we have to be much more careful when moving a loop-invariant computation. See Figure 1.

Moving a loop-invariant computation $d = a \oplus b$ before the loop is still okay on non-SSA if

1. that computation $d = a \oplus b$ dominates all loop exists after which d is still live (violated in Figure 1b),
2. and d is only defined once in the loop body (violated in Figure 1c),
3. and d is not live after the block before the loop (violated in Figure 1d)

a Good:

```
L0: d = 0
L1: i = i + 1
d = a ⊕ b
M[i] = d
if (i<N) goto L1
L2: x = d
```

b Bad:

```
L0: d = 0
L1: if (i>=N) goto L2
i = i + 1
d = a ⊕ b
M[i] = d
goto L1
L2: x = d
```

c Bad:

```
L0: d = 0
L1: i = i + 1
d = a ⊕ b
M[i] = d
d = 0
M[j] = d
if (i<N) goto L1
L2:
```

d Bad:

```
L0: d = 0
L1: M[j] = d
i = i + 1
d = a ⊕ b
M[i] = d
if (i<N) goto L1
L2: x = d
```

Figure 1: Good and bad examples for code motion of the loop-invariant computation d=a⊕b. **a**: good. **b**: bad, because d used after loop, yet should not be changed if loop iterates 0 times **c**: bad, because d reassigned in loop body, thus would be killed. **d**: bad, because initial d used in loop body before computing d=a⊕b.

Condition 2 is trivial for SSA. Condition 3 is simple, because d can only be defined once, which is still in the loop body, and thus cannot be live before. Condition 1 holds on SSA, if we make sure that we do not assign to the same variable in unrelated parts of the SSA graph (every variable assigned only once, statically). The node doesn't generally need to dominate all loop exits in SSA form. But if the variable is live, then it will. If it is doesn't dominate one of the loop exits (and thus the variable is not live after it), then we compute the expression in vain, but that still pays off if the loop usually executes often.

While-loops more often violate condition 1, because the loop body doesn't dominate the statements following the loop. A way around that is to turn while-loops into repeat-until-loops by prefixing them with an if statement testing if they will be executed at all.

3 Finding Loops

In source code, loops are obvious. But how do we find them in an intermediate code representation? Initially, we can easily tag loops, because they come from source code. Depending on the optimizations, this may become a little more tricky, however, if previous optimizations aggressively shuffled code around. More generally: how do find where the loops are in quite arbitrary intermediate code graphs?

We have already seen dominators in the theory behind SSA construction. There, the dominance frontier gives the minimal ϕ-node placement. Here we are not really interested in the dominance frontier, just in the dominator relation itself. We recap

Definition 1 (Dominators) *Node d dominates node n in the control-flow graph (notation $d \leq n$), iff every path from the entry node to n goes through d. Node d strictly dominates node n in the control-flow graph (notation $d < n$), iff in addition $d \neq n$. Node i is the (unique) immediate dominator of n (notation $i = \mathrm{idom}(n)$), iff $i < n$ and i does not dominate any other dominator of n (i.e., there is no j with $i < j$ and $j < n$).*

It is easy to see that $\mathrm{idom}(n)$ is unique just by the definition of dominators.

The *dominator tree* now is just the tree obtained by drawing an edge from $\mathrm{idom}(n)$ to n for all n. When the control-flow graph has an edge from node n back to a node h that dominates n ($h \leq n$), then this edge is called a *back edge*. Near such a back edge there is a loop. But where exactly is it? The *natural loop* for the back edge are all nodes a that the back edge start h (or header) also dominates and that have a path from a to the back edge end n. Note that the header does not uniquely identify the loop, because the same header node could be the target of multiple back edges coming from a branching structure into two natural loops. But the back edge uniquely identifies its natural loop.

In loop optimization, it makes sense to optimize inner loops first, because that's where most of the time is generally spent. That is, optimizations generally work inside-out. When the same header starts multiple loops, we cannot really say which of those is the inner loop, but have to consider all at once. If, instead, we have two loops starting at headers h and h' with $h \neq h'$ where h' is part of the (natural) loop of h, then the loop of h' is an inner loop nested inside the outer loop at h,

4 Strength Reduction

The basic idea of strength reduction is to replace computationally expensive operations by simpler ones that still have an equivalent effect. The primary application is to simplify multiplication by index variables to additions within loops. This optimization is crucial for computers where addition is a lot faster than multiplication and can gain a factor of 3 for numerical programs.

The simple-most instance of strength reduction turns a multiplication operation $x * 2^n$ into a shift operation $x \ll n$. More tricky uses of strength reduction occur frequently in loop traversals. Suppose we have a programming language with two-dimensional array operations occurring in a loop

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        ... use a[i,j] ...;
```

The address arithmetic for accessing a[i,j] is more involved, because it uses the base address a of the array and the size s of the base type to compute

$$a + i * m * s + j * s$$

This address computation needs 3 multiplications and 2 additions per access. When accessing several array locations in the loop, this address arithmetic quickly starts contributing significantly to the actual computation on the base types.

Since the array is represented contiguously in memory with row-major representation in C, one idea would be to just traverse the memory linearly.

```
t ← a;
e ← a + n*m*s;
if (t >= e) goto E;
L:
use *t ...;
t ← t + s;
E:
```

This optimized version only needs one addition per loop. It is essentially based on the insight that a[i+1,j] is the same memory location as a[i,j+m-1] The optimization we have used here assumes that i and j are not used otherwise in the loop body, so that their computation can be eliminated. Otherwise, they stay.

In order to perform this strength reduction, however, we need to know which of the variables change linearly in the loop. It certainly would be incorrect if there was a nonlinear change of i, like in $i = i * (i + 1)$.

References

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

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.