Lecture Notes on Peephole Optimizations and Common Subexpression Elimination

15-411: Compiler Design
Frank Pfenning, Jan Hoffmann, and Seth Goldstein

Lecture 17
October 28, 2021

1 Introduction

In lecture we discussed Peephole optimizations, discovering natural loops, and two loop optimizations: Loop invariant code motion (LICM) and Loop Induction variable elimination. LICM will be discussed again next week when we discuss Partial Redundancy Elimination (PRE), also known as Lazy Code Motion, which subsumes both LICM and common subexpression elimination, covered in these lecture notes.

Peephole optimizations are optimizations that are performed *locally* on a small number of instructions. The name is inspired from the picture that we look at the code through a peephole and make optimization that only involve the small amount code we can see and that are indented of the rest of the program.

There is a large number of possible peephole optimizations. The LLVM compiler implements for example more than 1000 peephole optimizations [LMNR15]. In this lecture, we discuss three important and representative peephole optimizations: *constant folding*, *strength reduction*, and *null sequences*.

The idea of common subexpression elimination (CSE) is to avoid performing the same operation twice by replacing (binary) operations with variables. To ensure that these substitutions are sound we introduce *dominance*, which ensures that substituted variables are always defined. Please note that PRE will do a better job at recognizing and eliminating redundant operations than either LICM or CSE alone.

2 Constant Folding

Optimizations have two components: (1) a condition under which they can be applied and the (2) code transformation itself. The optimization of *constant folding* is

a straightforward example of this. The code transformation itself replaces a binary operation with a single constant, and applies whenever $c_1 \odot c_2$ is defined.

$$l : x \leftarrow c_1 \odot c_2 \;\} \longrightarrow \{\; l : x \leftarrow c \qquad \textit{(where } c = c_1 \odot c_2\textit{)}$$

We can write the similar operation for when an mathematical operation is undefined:

$$l : x \leftarrow c_1 \odot c_2 \;\} \longrightarrow \{\; l : \mathsf{raise}(\mathsf{arith}) \qquad \textit{(where } c_1 \odot c_2 \textit{ is not defined)}$$

Constant folding can also be used to rewrite conditionals:

$$l : \mathsf{if}\ c_1 \,?\, c_2\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2 \;\} \longrightarrow \{\; l : \mathsf{goto}\ l_1 \qquad \textit{(if } c_1 \,?\, c_2 \textit{ is true)}$$

$$l : \mathsf{if}\ c_1 \,?\, c_2\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2 \;\} \longrightarrow \{\; l : \mathsf{goto}\ l_2 \qquad \textit{(if } c_1 \,?\, c_2 \textit{ is false)}$$

Turning constant conditionals into gotos may cause entire basic blocks of code to become unnecessary. You can also perform constant folding across multiple instructions.

$$\left.\begin{array}{l} l_1 : y \leftarrow x + c_1 \\ l_2 : z \leftarrow y + c_2 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l_1 : y \leftarrow x + c_1 \\ l_2 : z \leftarrow x + c \end{array}\right. \qquad \textit{(if } c_1 + c_2 = c\textit{)}$$

The advantage of this optimization is that line l_1 can become dead code if y is not needed in the successors of line l_2. Of course, we have to ensure that y in line l_2 has always been defined in l_1 as opposed to a different line in the original program. This is always the case if the program is in SSA form.

These operations are straightforward because they can be performed without checking any other part of the code. Most other optimizations have more complicated conditions about when they can be applied.

3 Strength Reduction

Strength reduction in general replaces and expensive operation with a simpler one. Sometimes it can also eliminate an operation altogether, based on the laws of modular, two's complement arithmetic. Recall that we have the usual laws of arithmetic modulo 2^{32} for addition, subtraction, multiplication, but that comparisons are more difficult to transform.[1]

Common simplifications (and some symmetric counterparts):

$$\begin{array}{lcl} a + 0 & = & a \\ a - 0 & = & a \\ a * 0 & = & 0 \\ a * 1 & = & a \\ a * 2^n & = & a \mathbin{<<} n \end{array}$$

[1]For example, $x + 1 > x$ is not true in general, because x could be the maximal integer, $2^{31} - 1$.

but one can easily think of others involving further arithmetic of bit-level operations. (Remember, however, that $a/2^n$ is not equal to $a >> n$ unless a is positive.) Another optimization that may be interesting for optimization of array accesses is the distributive law:

$$a * b + a * c = a * (b + c)$$

where a could be the size of an array element and $(b + c)$ could be an index calculation.

4 Null Sequences

During register allocation, we noted that it is beneficial to produce self moves like $r \leftarrow r$ by assigning two temps t, s the same register r. The reason that self moves are beneficial is because they can be removed from the code since say have no effect. We call an operation or a sequence of operations that does not have an effect a *null sequence*.

$$\left.\begin{array}{l} l : x \leftarrow x \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l : \mathsf{nop} \end{array}\right.$$

When we spill a temp t to the stack and perform two operations like $t \leftarrow t + c$ on this temp in a row. Then we might move t into a register before the addition and move it back after the addition. As a result, we have a move $r \leftarrow t$ directly followed by a move $t \leftarrow r$, which seems redundant.

$$\left.\begin{array}{l} l_1 : x \leftarrow y \\ l_2 : y \leftarrow x \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l_1 : x \leftarrow y \\ l_2 : \mathsf{nop} \end{array}\right.$$

We can not remove the first move because x might be needed after line l_2. However, we might be able to remove it with dead-code elimination.

Another null sequence that you have seen already is a jump to the next line. Such jumps can be eliminated but they destroy basic blocks. So be careful to only perform this optimization when you do not rely on basic blocks anymore.

$$\left.\begin{array}{l} l_1 : \mathsf{goto}\ l_2 \\ l_2 : \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l_1 : \mathsf{nop} \\ l_2 : \end{array}\right.$$

There are plenty of other peephole optimizations that you can find in the compiler literature.

5 Common Subexpression Elimination

Copy propagation allows us to have optimizations with this form:

$$\left.\begin{array}{l} l : x \leftarrow y \\ \ldots \\ l' : instr(x) \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l : x \leftarrow y \\ \ldots \\ l' : instr(y) \end{array}\right.$$

It is natural to ask about transforming a similar computation on compound expressions:

$$\left.\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ \cdots \\ l' : instr(x) \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ \cdots \\ l' : instr(s_1 \oplus s_2) \end{array}\right.$$

However, this will not work most of the time. The result may not even be a valid instruction (for example, if $instr(x) = (y \leftarrow x \oplus 1)$. Even if it is, we have made our program bigger, and possibly more expensive to run. However, we can consider the *opposite*: In a situation

$$\begin{array}{rcl} l & : & x \leftarrow s_1 \oplus s_2 \\ & & \cdots \\ k & : & y \leftarrow s_1 \oplus s_2 \end{array}$$

we can replace the second computation of $s_1 \oplus s_2$ by a reference to x (under some conditions), saving a reduction computation. This is called *common subexpression elimination* (CSE).

The thorny issue for common subexpression elimination is determining when the optimization above is performed. Consider the following program in SSA form:

```
Lab1 :                Lab2 :          Lab3 :          Lab4(w) :
  x ← a ⊕ b             y ← a ⊕ b       z ← x ⊕ b       u ← a ⊕ b
  if a < b              goto Lab4(y)    goto Lab4(z)    ...
    then goto Lab2
    else goto Lab3
```

If we want to use CSE to replace the calculation of $a \oplus b$ in Lab4, then there appear to be two candidates: we can rewrite $u \leftarrow a \oplus b$ as $u \leftarrow x$ or $u \leftarrow y$. However, only the first of these is correct! If control flow passes through Lab3 instead of Lab2, then it will be an error to access y in Lab4.

In order to rewrite $u \leftarrow a \oplus b$ as $u \leftarrow x$, in general we need to know that x will have the right value when execution reaches line k. Being in SSA form helps us, because it lets us know that the right-hand sides will always have the same meaning if they are syntactically identical. But we also need to know x even be defined along every control flow path that takes us to Lab4.

What we would like to know is that every control flow path from the beginning of the code (that is, the beginning of the function we are compiling) to line k goes through line l. Then we can be sure that x has the right value when we reach k. This is the definition of the *dominance* relation between lines of code. We write $l \geq k$ if l dominates k and $l > k$ if it l strictly dominates k. We see how to define it in the next section; once it is defined we use it as follows:

$$\left.\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ \cdots \\ k : y \leftarrow s_1 \oplus s_2 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ \cdots \\ k : y \leftarrow x \end{array}\right. \quad \text{(provided } l > k\text{)}$$

It was suggested in lecture that this optimization would be correct even if the binary operator is effectful. The reason is that if l dominates k then we always execute l first. If the operation does *not* raise an exception, then the use of x in k is correct. If it does raise an exception, we never reach k. So, yes, this optimization works even for binary operations that may potentially raise an exception.

6 Dominance

We covered Dominance when discussing SSA. We briefly review it here. On general control flow graphs, dominance is an interesting relation and there are several algorithms for computing this relationship. We can cast it as a form of *forward dataflow analysis*.

Most algorithms directly operate on the control flow graph. A simple and fast algorithm that works particularly well in our simple language is described by Cooper et al. [CHK06] which is empirically faster than the traditional Lengauer-Tarjan algorithm [LT79] (which is asymptotically faster). We will not discuss these algorithms in detail.

The approaches we are taking exploits the simplicity of our language and directly generates the dominance relationship as part of code generation. The drawback is that if your code generation is slightly different or more efficient, or if your transformation change the essential structure of the control flow graph, then you need to update the relationship. In this lecture, we consider just the basic cases.

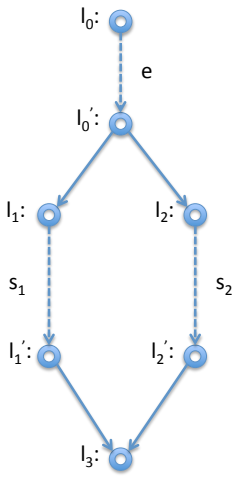
For *straight-line code* the predecessor of each line is its immediate dominator, and any preceding line is a dominator.

For conditionals, consider

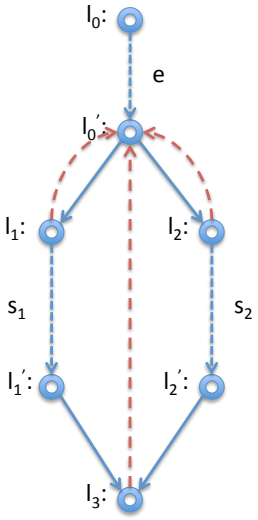
$$\mathsf{if}(e, s_1, s_2)$$

We translate this to the following code, $\check{e}$ or $\check{s}$ is the code for e and s, respectively and $\hat{e}$ is the temp through which we can refer to the result of evaluating e.

l_0 : $\check{e}$
l_0' : if $(\hat{e}$!= $0)$ goto l_1 else goto l_2
l_1 : $\check{s}_1$; l_1' : goto l_3
l_2 : $\check{s}_2$; l_2' : goto l_3
l_3 :

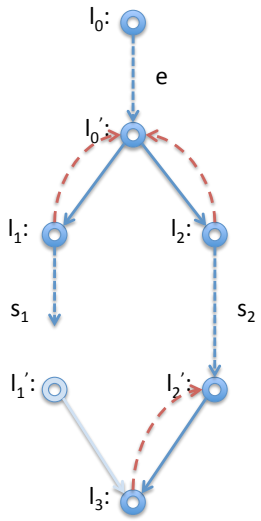


On the right is the corresponding control-flow graph. Now the immediate dominator of l_1 should be l_0' and the immediate dominator of l_2 should also be l_0'. Now for l_3 we don't know if we arrive from l_1' or from l_2'. Therefore, neither of these nodes will dominate l_3. Instead, the immediate dominator is l_0', the last node we can be sure to be traversed before we arrive at l_3'. Indicating immediate dominators with dashed read lines, we show the result below.



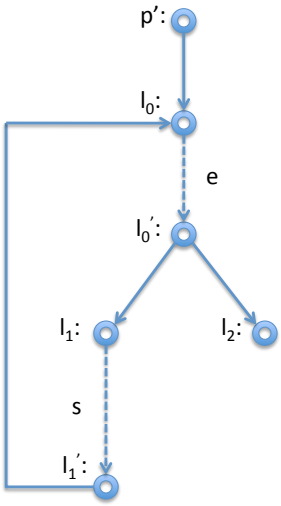
However, if it turns out, say, l_1' is not reachable, then the dominator relationship looks different. This is the case, for example, if s_1 in this example is a return state-

ment or is known to raise an error. Then we have instead:



In this case, l_1' : goto l_3 is *unreachable* code and can be optimized away. Of course, the case where l_2' is unreachable is symmetric.

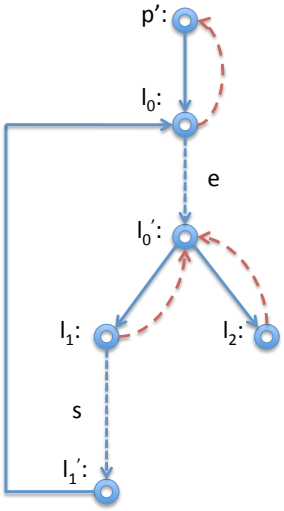
For loops, it is pretty easy to see that the beginning of the loop dominates all the statements in the loop. Again, considering the straightforward compilation of a while loop with the control flow graph on the right.



l_0 : $\check{e}$
l_0' : if $(\hat{e}$ == $0)$ goto l_2 else goto l_1
l_1 : $\check{s}$
l_1' : goto l_0
l_2 :

Interesting here is mainly that the node p' just before the loop header l_0 is indeed the immediate dominator of l_0, even l_0 has l_1' as another predecessor. The definition

makes this obvious: when we enter the loop we have to come through p' node, on subsequent iterations we come from l_1'. So we cannot be guaranteed to come through l_1', but we are guaranteed to come through p' on our way to l_0.



7 Implementing Common Subexpression Elimination

To implement common subexpression elimination we traverse the program, looking for definitions $l : x \leftarrow s_1 \odot s_2$. If $s_1 \odot s_2$ is already in the table, defining variable y at k, we replace l with $l : x \leftarrow y$ if k dominates l. Otherwise, we add the expression, line, and variable to the hash table.

Dominance can usually be checked quite quickly if we maintain a dominator tree, where each line has a pointer to its immediate dominator. We just follow these pointers until we either reach k (and so $k > l$) or the root of the control-flow graph (in which case k does not dominate l).

8 Termination

When applying code transformations, we should always consider if the transformations terminate. Clearly, each step of dead code elimination reduces the number of assignments in the code. We can therefore apply it arbitrarily until we reach *quiescence*, that is, neither of the dead code elimination rules is applicable any more. Quiescence is the rewriting counterpart to saturation for inference, as we have discussed in prior lectures. Saturation means that any inference we might apply only

has conclusions that are already known. Quiescence means that we can no longer apply any rewrite rules.

A single application of constant propagation reduces the number of variable occurrence in the program and must therefore reach quiescence. It also does not increase the number of definitions in the code, and can therefore be mixed freely with dead code elimination.

It is more difficult to see whether copy propagation will always terminate, since the number of variable occurrences stays the same, as does the number of variable definitions. In fact, in a code pattern

$$\begin{array}{lcl} l & : & x \leftarrow y \\ k & : & w \leftarrow x \\ m & : & instr(w) \\ m' & : & instr(x) \end{array}$$

we could decrease the number of occurrence of x by copy propagation from line l and then increase it again by copy propagation from line k. However, if we consider a string partial order $x > y$ among variables if the definition of x uses y (transitively closed), then copy propagation reduces the occurrence of a variable by a strictly smaller one. This order is well-founded since in SSA we cannot have a cycle among the definitions. If x is defined in terms of y, then y could not be defined in terms of x since it the single definition of y must come before x in the control flow graph.

References

[CHK06] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. Technical Report TR-06-33870, Department of Computer Science, Rice University, 2006.

[LMNR15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, 2015.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):115–120, July 1979.