Lecture Notes on Basic Optimizations

15-411: Compiler Design
Frank Pfenning

Lecture 16
Oct 17, 2013

1 Introduction

The opportunities for optimizations[1] in compiler-generated code are plentiful. Generally speaking, they arise more from the tensions between the high-level source language and the lower-level target language, rather than any intrinsic inefficiencies in the source. One common source, sometimes estimated to constitute as much as 70% of optimization opportunities, is address arithmetic and is therefore tied to structs and arrays.

In this lecture we discuss basic optimizations that apply pervasively during the compilation process. In the next two lectures we will discuss specifically optimizations of loops. Another class of optimizations is concerned with functions calls, like tail-call optimization and inlining. You have the opportunity to consider these in Assignment 3.

2 Dead Code Elimination

Optimizations have two components: (1) a condition under which they are application and the (2) code transformation itself. The applicability condition can come in various forms, but often requires a dataflow analysis.

As a warm-up exercise, we reconsider dead code elimination from Section 4 of Lecture 5. We defined there a predicate $\mathsf{needed}(l, x)$ which is defined via a backward dataflow analysis. Instructions of the forms

$$\begin{array}{lcl} l & : & x \leftarrow s_1 \oplus s_2 \\ l & : & x \leftarrow s \end{array}$$

[1]Very little in a compiler is actually optimal, so "optimizations" should be interpreted as "efficiency improvements".

with a pure (effect-free) right-hand side are considered dead code if x is not needed in the successor $l+1$ of l.

We can describe the transformation by stating how the given lines are affected, and which new lines should perhaps be added. We use the notation

$$\frac{P_1 \ldots P_m}{L_1 \ldots L_n}$$

which replaces $P_1 \ldots P_m$ with $L_1 \ldots L_n$. Any condition of applicability is also listed among the premises. This is an example of *linear inference*, where the process of inference consumes the premises and adds the conclusions.[2] In this notation, dead code elimination could be described as

$$\frac{l : x \leftarrow s_1 \oplus s_2 \quad \mathsf{succ}(l, l') \quad \neg\mathsf{needed}(l', x)}{l : \mathsf{nop}}$$

$$\frac{l : x \leftarrow s \quad \mathsf{succ}(l, l') \quad \neg\mathsf{needed}(l', x)}{l : \mathsf{nop}}$$

where $\oplus$ is an effect-free operation. We replace the instruction with a nop instead of just deleting it so that, for example, jumps to line l will continue to remain value. At a later stage of optimization, spurious no-ops can be deleted from the code.

3 Constant Propagation

Another straightforward optimization is *constant propagation*. If we have definition $l : x \leftarrow c$ for a constant c, we might want to replace an occurrence of x by c in the hope that we may be able to eliminate the assignment (and x) altogether. Moreover, we may be able to apply other optimizations where we have substituted c for x, such as constant folding.

The tricky question is when this is a correct optimization. For example, in the code

$$\begin{array}{lcl} l & : & x \leftarrow c \\ & & \ldots \\ k & : & y \leftarrow x + 1 \end{array}$$

it depends on what happens in the lines between l and k. Jumps may target lines in between, or there may be another assignment to x so that x no longer has the value c when execution reaches k.

[2]Linear inference gives rise to a relatively new kind of logic called *linear logic* which we do not discuss further in the class. Some materials on linear inference and linear logic can be found at `http://www.cs.cmu.edu/~fp/courses/15816-s12/`

Rather than give a general solution to this question, we greatly simplify our lives by assuming that the code has been transformed in static single-assignment (SSA) form. In that form, any variable is defined exactly one so a reference to x must be the correct one. We use the notation $l' : \mathit{instr}(x)$ for an instruction that *uses* x (that is, $\mathsf{uses}(l', x)$) and $\mathit{instr}(c)$ for the result of replacing x by c.

$$\frac{l : x \leftarrow c \qquad l' : \mathit{instr}(x)}{l : x \leftarrow c \qquad l' : \mathit{instr}(c)}$$

Note that we need to repeat $l : x \leftarrow c$ in the conclusion since the premise is consumed when the inference rule is applied.

4 Copy Propagation

Copy propagation is very similar to constant propagation, except that one variable is defined in terms of another.

$$\frac{l : x \leftarrow y \qquad l' : \mathit{instr}(x)}{l : x \leftarrow y \qquad l' : \mathit{instr}(y)}$$

Again, we should ask if this is sound, assuming the program is in SSA form. We know there is exactly one definition of y that is available at line l. Since x is available at line l', y must also be available there so the replacement is sound.

5 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 linear inference counterpart to saturation for ordinary 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 linear inference.

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

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

we could for 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 progation 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.

6 Constant Folding

Constant folding evaluates a constant epxression at compile time. In the three-address form, this is simply:

$$\frac{l : x \leftarrow c_1 \odot c_2 \quad c_1 \odot c_2 = c}{l : x \leftarrow c}$$

where $\oplus$ doubles as a syntactic binary operation and its arithmetic counterpart. We need to make sure that $c_1 \odot c_2$ is defined in this case (and should not raise an exception at runtime). There is really not other precondition to this transformations.

7 Common Subexpression Elimination

It is natural to try to apply a transformation similar to copy or constant propagation to of the form

$$\begin{array}{lcl} l & : & x \leftarrow s_1 \oplus s_2 \\ & & \ldots \\ k & : & \mathit{instr}(x) \end{array}$$

where we replace x by $s_1 \oplus s_2$. However, this will not work most of the time, because the result may not even be a valid instruction (for example, if $\mathit{instr}(x) = (y \leftarrow x \oplus 1)$. Moreover, the program becomes bigger, plus we are computing an expression more than once instead of just once, so this is likely to make the code slower rather than faster.

However, we can consider the *opposite*: In a situation

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

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).

For this to be correct we need to know that x will have the right value when execution reaches line k. Because we are in SSA form, the right-hand sides will always have the same meaning if they are syntactically identical. But will x be available at k?

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 dominated k. We see how to define it in the next section; once it is defined we use it as follows:

$$\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ k : y \leftarrow s_1 \oplus s_2 \\ l > k \\ \hline l : x \leftarrow s_1 \oplus s_2 \\ k : y \leftarrow x \end{array}$$

8 Dominance

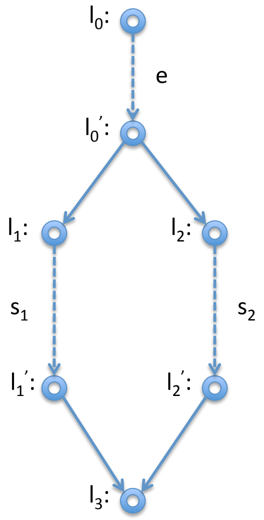
On general control flow graphs, dominance is an interesting relation and there are several algorithms for computing this relationship. We can cast it as a form of *forward data-flow analysis*. One of the approaches exploits the simplicity of our language to directly generate the dominance relationship as part of code generation. We briefly discuss this here. 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. 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). In this lecture, we consider just the basic cases.

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

For conditionals, consider

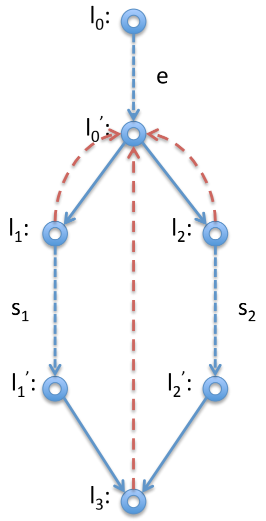
$$\mathsf{if}(e, s_1, s_2)$$

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.

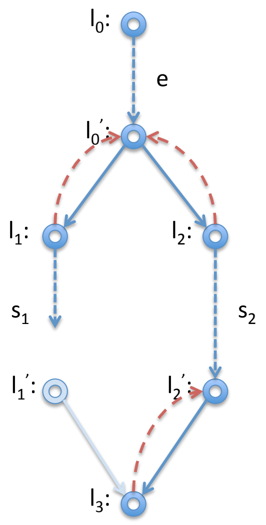


```
$l_0$   :  $\check{e}$
$l'_0$  :  if $(\hat{e}$ != $0)$ goto $l_1$ ; goto $l_2$
$l_1$   :  $\check{s}_1$ ; $l'_1$ : goto $l_3$
$l_2$   :  $\check{s}_2$ ; $l'_2$ : goto $l_3$
$l_3$   :
```

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.

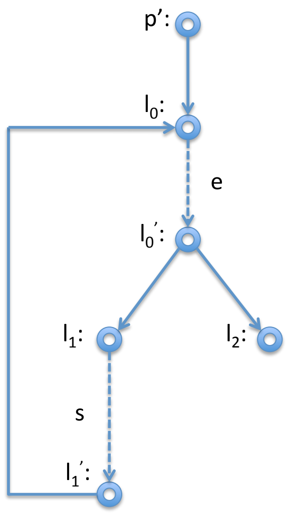


However, if it turns out, say, l'_1 is not reachable, then the dominator relationship looks different. This is the case, for example, if s_1 in this example is a return statement or is known to raise an error. Then we have instead:



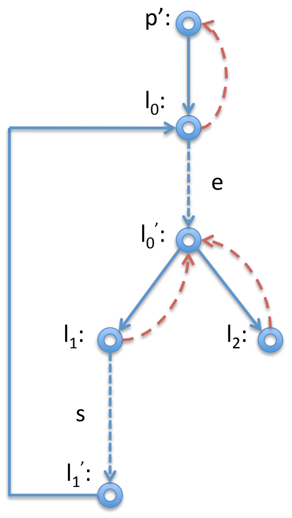
In this case, l'_1 : goto l_3 is *unreachable* code and can be optimized away. Of course, the case where l'_2 is unreachable is symmetric.

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.



$$\begin{array}{lcl} l_0 & : & \check{e} \\ l'_0 & : & \text{if } (\hat{e} == 0) \text{ goto } l_2 \text{ ; goto } l_1 \\ l_1 & : & \check{s} \\ l'_1 & : & \text{goto } l_0 \\ l_2 & : & \end{array}$$

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.



9 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).

10 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[3]

Common simplifications (and some symmetric counterparts):

$$\begin{array}{rcl} a+0 & = & a \\ a-0 & = & a \\ a*0 & = & 0 \\ a*1 & = & a \end{array}$$

but one can easily think of others involving further arithmetic of bit-level operations. One that may be interesting for optimization of array accesses is the distributive law:

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

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

11 A Simple Example

Let's consider the rather innocuous C0 code fragment

```
A[i] = A[i] + 1
```

Assuming we perform no null or array bound checking, and a holds the address of the array, we would obtain something like the following. The semantics of C0 require left-to-right evaluation, so we first obtain the address of $A[i]$ in t_1 (lines $l_0 - -l_1$), then we evaluate the right-hand-side (lines $l_2 - -l_5$), and then we write to the memory at address t_3 (line l_6). The number 4 is the size of $|\mathsf{int}|$, which is the type of the array elements.

$$\begin{array}{lcllll} l_0 & : & t_0 & \leftarrow & \mathbf{4*i} & \text{\# cse} \\ l_1 & : & t_1 & \leftarrow & a+t_0 & \\ l_2 & : & t_2 & \leftarrow & \mathbf{4*i} & \text{\# cse} \\ l_3 & : & t_3 & \leftarrow & a+t_2 & \\ l_4 & : & t_4 & \leftarrow & M[t_3] & \\ l_5 & : & t_5 & \leftarrow & t_4+1 & \\ l_6 & : & M[t_1] & \leftarrow & t_5 & \end{array}$$

[3]For example, $x+1 > x$ is false in general, because x could be the maximal integer, $2^{31}-1$.

We notice that l_0 and l_2 both compute $4 * i$ so we obtain the code on the left. This is now subject to copy propagation from l_2 to l_3 to obtain the code on the right.

$$\begin{array}{lclcl}
l_0 & : & t_0 & \leftarrow & 4 * i \\
l_1 & : & t_1 & \leftarrow & a + t_0 \\
l_2 & : & \mathbf{t_2} & \leftarrow & \mathbf{t_0} \\
l_3 & : & t_3 & \leftarrow & a + \mathbf{t_2} \\
l_4 & : & t_4 & \leftarrow & M[t_3] \\
l_5 & : & t_5 & \leftarrow & t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow & t_5
\end{array}
\qquad
\begin{array}{lclcll}
l_0 & : & t_0 & \leftarrow & 4 * i & \\
l_1 & : & t_1 & \leftarrow & \mathbf{a + t_0} & \text{\# cse} \\
l_2 & : & t_2 & \leftarrow & t_0 & \\
l_3 & : & t_3 & \leftarrow & \mathbf{a + t_0} & \text{\# cse} \\
l_4 & : & t_4 & \leftarrow & M[t_3] & \\
l_5 & : & t_5 & \leftarrow & t_4 + 1 & \\
l_6 & : & M[t_1] & \leftarrow & t_5 &
\end{array}$$

The code on the right yields another opportunity for common subexpressions elimination for lines l_1 and l_3. The result is pictured on the left, followed again by copy propagation on the right.

$$\begin{array}{lclcl}
l_0 & : & t_0 & \leftarrow & 4 * i \\
l_1 & : & t_1 & \leftarrow & a + t_0 \\
l_2 & : & t_2 & \leftarrow & t_0 \\
l_3 & : & \mathbf{t_3} & \leftarrow & \mathbf{t_1} \\
l_4 & : & t_4 & \leftarrow & M[\mathbf{t_3}] \\
l_5 & : & t_5 & \leftarrow & t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow & t_5
\end{array}
\qquad
\begin{array}{lclcll}
l_0 & : & t_0 & \leftarrow & 4 * i & \\
l_1 & : & t_1 & \leftarrow & a + t_0 & \\
l_2 & : & \mathbf{t_2} & \leftarrow & t_0 & \text{\# dead} \\
l_3 & : & \mathbf{t_3} & \leftarrow & t_1 & \text{\# dead} \\
l_4 & : & t_4 & \leftarrow & M[\mathbf{t_1}] & \\
l_5 & : & t_5 & \leftarrow & t_4 + 1 & \\
l_6 & : & M[t_1] & \leftarrow & t_5 &
\end{array}$$

A pass of dead code elimination yields the code in which the address of $A[i]$ is computed only once.

$$\begin{array}{lclcl}
l_0 & : & t_0 & \leftarrow & 4 * i \\
l_1 & : & t_1 & \leftarrow & a + t_0 \\
l_2 & : & \mathsf{nop} & & \\
l_3 & : & \mathsf{nop} & & \\
l_4 & : & t_4 & \leftarrow & M[t_1] \\
l_5 & : & t_5 & \leftarrow & t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow & t_5
\end{array}$$

This example illustrates the *cascading* of optimizations: initially, we only had two common subexpressions, but after some optimizations more were uncovered. Techniques such as global value numbering help to avoid multiple passes over code by combining several iterations into one. Neededness analysis is another example where multiple lines are declared dead code at once, rather than in sequence with new analysis in between.

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.

[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.