Lecture Notes on Value Propagation

15-411: Compiler Design
Frank Pfenning, Rob Simmons

Lecture 17
October 27, 2015

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.

This is not our first discussion of compiler optimization; this lecture revisits material that we covered back in the lecture on dataflow analysis. In that lecture we covered *neededness analysis*, a backwards dataflow analysis (like liveness) underlying the optimization of *dead code elimination*. We also covered *reaching definitions*, a forward dataflow analysis that can be used as the basis for the optimization of *constant propagation*.

In this lecture, we will discuss two completely *local* analyses, constant folding and strength reduction, that can be applied anywhere, as well as two analyses, constant propagation and copy propagation, that require either reaching definitions or SSA to reach their full potential.

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

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

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 propagation 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. All 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.[2]

Common simplifications (and some symmetric counterparts):

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

but one can easily think of others involving further arithmetic of bit-level operations. (Remember, however, that $a/2^n$ is not equal to $a \ \mathsf{>>}\ n$ unless a is positive.) Another optimization 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.

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

4 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

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

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.

If there are no jumps or labels between lines l and k (because they are in the same basic block or extended basic block), then this is an easy condition to check: we just check that there are no other assignments to x between lines l and k. To perform this operation in general, we would need to use a reaching definitions analysis to check that the definition of x on line l is the only definition that reaches line k.

Rather than limiting our optimization to a single basic block or performing an expensive reaching definitions analysis, we can 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, $\mathsf{use}(l', x)$) and $\mathit{instr}(c)$ for the result of replacing x by c.

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

Note that we to repeat $l : x \leftarrow c$ on the right-hand side because it remains unchanged If we perform *all* such replacements at the same time, then we know that line l is no longer needed. Alternatively, we can let dead code elimination, which we considered previously, get rid of unnecessary lines after performing constant propagation.

5 Copy Propagation

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

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

This optimization is always applicable if 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.

Copy propagation has the potential downside that y now may be live for longer than it was before, making register allocation more difficult. On the other hand, it can also allow allow additional simplifications to be made in SSA programs. In the example below, which is in extended basic block form, after first propagating x and eliminating y and z, we see that it is possible to remove the first parameter to lab2 according to our SSA minimization algorithm, replacing p with x everywhere in the program.

```
lab1(a, b) :                lab1(a, b) :                lab1(a, b) :
  x ← f(a, b)                 x ← f(a, b)                 x ← f(a, b)
  if a < b                    if a < b                    if a < b
    then                        then                        then
      y ← x
      goto lab2(y, a)             goto lab2(x, a)             goto lab2(a)
    else                        else                        else
      z ← x
      goto lab2(z, b)             goto lab2(x, b)             goto lab2(b)

lab2(p, q) :                lab2(p, q) :                lab2(q) :
  ...                         ...                         ...
```

6 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

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

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 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.