CSE401: Optimization
David Notkin
Autumn 2000

Example

\[ x := a[i] + b[2]; \]
\[ c[i] := x - 5; \]
\[ t1 := *(fp + ioffset) \]
\[ t2 := t1 * 4 \]
\[ t3 := t2 + 2 \]
\[ t4 := t3 * 4 \]
\[ t5 := 2 \]
\[ t6 := t5 * 4 \]
\[ t7 := t6 + t5 \]
\[ t8 := t7 + boffset \]
\[ t9 := t8 + 5 \]
\[ *(fp + xoffset) := t9 \]
\[ t10 := *(fp + xoffset) \]
\[ t11 := 2 \]
\[ t12 := t10 - t11 \]
\[ t13 := *(fp + xoffset) \]
\[ t14 := t13 * 4 \]
\[ t15 := t14 + 5 \]
\[ *(t15 + coffset) := t15 \]

Local optimization
- Analysis and optimizations within a basic block
- A basic block is a straight-line sequence of statements
  - No control flow into or out of middle of sequence
- Local optimizations are more powerful than peephole
  - Not too hard to implement
  - Can in fact be machine-independent, if done on intermediate code

Local constant propagation
- If a variable is assigned to a constant, replace downstream uses of the variable with the constant
- May enable further constant folding

Local dead assignment elimination
- If the lefthand side of assignment is never referenced again before being overwritten
  - Then remove the assignment
  - This sometimes happens as cleaning up from other optimizations
Example

```
const count : int = 10;
...
x = count * 5;
y = x * 3;
z = input
```

Intermediate code after constant propagation

Example

```
x := 50
t6 := exp(50,3)
y := t6
x := input()
```

Local common subexpression elimination
- Avoid repeating the same calculation
- Requires keeping track of available expressions

Next
- Intraprocedural optimizations
  - Code motion
  - Loop induction variable elimination
  - Global register allocation
  - Interprocedural optimizations
    - Inlining
    - After that...how to implement these optimizations
    - One more kind of optimization, way beyond the scope of this class: dynamic compilation

Intraprocedural optimizations
- Enlarge scope to entire procedure
- Provides more opportunities for optimization
- Have to deal with branches, merges and loops
- Can do constant propagation, common subexpression elimination, etc. at this level
- Can do new things, too, like loop optimizations
- This is the most common level for optimizing compilers to work

Code motion
- Goal: move loop-invariant calculations out of loops
- Can do this at the source or intermediate code level
- for i := 1 to 10 do
  a[i] := a[i] + b[i];
z := z + 10000
end
Intermediate code level

\[
*(fp+ioffset) := 1 \\
_l0: \text{for } i := 1 \text{ to } 10 \text{ do} \\
* a[i] := b[j]; \\
\text{end}
\]

\[
_l1: \text{if } *(fp+ioffset) > 10 \text{ goto } _l1 \\
t1 := *(fp+ioffset) \\
t2 := t1*4 \\
t3 := fp+t2 \\
t4 := *(t3+boffset) \\
t5 := *(fp+ioffset) \\
t6 := t5*4 \\
t7 := fp+t6 \\
t8 := *(fp+ioffset) \\
t9 := t6+1 \\
*(fp+ioffset) := t9 \\
\text{goto } _l0 \\
\text{end}
\]

Loop induction variable elimination

- For-loop index is an \textit{induction variable}
- Incremented each time through the loop
- Offsets, pointers calculated from it
- If used only to index arrays, can rewrite with pointers
  - Compute initial offsets, pointers before loop
  - Increment offsets, pointers each time around the loop
- No expensive scaling in the loop

Example

\[
\text{for } i := 1 \text{ to } 10 \text{ do} \\
a[i] := a[i] + x; \\
\text{end}
\]

\[
\text{for } p := &a[1] \text{ to } &a[10] \text{ do} \\
*p := *p + x; \\
\text{end}
\]

Global register allocation

- Try to allocate local variables to registers
- If two locals don’t overlap, then give them the same register
- Try to allocate most frequently used variables to registers first

\[
\text{procedure } \text{foo}(n:\text{int}, x:\text{int}):\text{int}; \\
\text{var } \text{sum, i:}\text{int;} \\
\text{begin} \\
\text{sum} := x; \\
\text{for } i := 1 \text{ to } n \text{ do} \\
\text{sum} := sum + i; \\
\text{end} \\
\text{return } \text{sum;} \\
\text{end \text{foo};}
\]