# CSE401: Optimization

David Notkin
Autumn 2000



## Optimization

- Identify inefficiencies in target or intermediate code
- Replace with equivalent but "better" sequences
- Should really be "improvement" instead of "optimize"
- And remember, the source program is the first place to look for improvements in performance



## Example

```
x := a[i] + b[2];
c[i] := x - 5;
```

```
t1  := *(fp + ioffset)
t2  := t1 * 4
t3  := fp + t2
t4  := *(t3 + aoffset)
t5  := 2
t6  := t5 * 4
t7  := fp + t6
t8  := *(t7 + boffset)
t9  := t4 + t8
*(fp + xoffset) := t9
t10 := *(fp + xoffset)
t11 := 5
t12 := t10 - t11
t13 := *(fp + ioffset)
t14 := t3 * 4
t15 := fp + t14
*(t15 + coffset) := t15
```



## Kinds of optimizations

- Scope of study is central to what optimizations can be peformed
  - That is, a large scope means you can perform better optimizations, but it requires more complexity
- *Peephole*: look at adjacent instructions
- *Local*: look at straight-line sequences of instructions
- *Global (intraprocedural)*: look at whole procedure
- *Interprocedural*: look across procedures



## Peephole

- After codegen, look at adjacent instructions
  - Try to replace them with something better
- If you have
  - ```
    sw $8,12($fp)
    lw $12,12($fp)
    ```
- You can replace it with
  - ```
    sw $8,12($fp)
    mv $12,$8
    ```



## Peephole examples: 68k

A. If you have
  - ```
    sub sp,4,sp
    mov r1,0(sp)
    ```
- You can replace it with
  - `mov r1,-(sp)`

B. If you have
  - ```
    mov 12(fp),r1
    add r1,1,r1
    mov r1,12(fp)
    ```
- You can replace it with
  - `mov r1,-(sp)`

## A view

- You could consider peephole optimization, in some situations, as increasing the sophistication of the instruction selection algorithm



## Peephole optimization of jumps

- Eliminate jumps to jumps
- Eliminate jumps after conditional branches
- "adjacent" instructions in this case means "adjacent in the control flow" of the program

```
if a < b then
  if c < d then
    # do nothing
  else
    stmt1;
  end;
else
  stmt2;
end;
```



## Algebraic simplifications

*by peephole or codegen*

- "constant folding" and "strength reduction" are common names for this kind of optimization
  - `z := 3 + 4`
  - `z := x + 0`
    `z := x * 1`
  - `z := x * 2`
    `z := x * 8`
  - `z := x / 8`
  - `float x,y;`
    `z := (x + y) - y;`



## 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



## Example

```
const count : int = 10;
…
x := count * 5;
y := x ^ 3;
```

```
t1  := 10
t2  := 5
t3  := t1 * t2
x   := t3

t4  := x
t5  := 3
t6  := exp(t4,t5)
y   := t6
```

## 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;
x := input
```

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

Intermediate code after constant propagation



## Local common subexpression elimination

- Avoid repeating the same calculation
- Requires keeping track of available expressions



## Example

```
…a[i] + b[i]…
```

```
t1  := *(fp + ioffset)
t2  := t1 * 4
t3  := fp + t2
t4  := *(t3 + aoffset)

t5  := *(fp + ioffset)
t6  := t5 * 4
t7  := fp + t6
t8  := *(t7 + boffset)

t9  := t4 + t8
```



## Lecture++

- Intraprocedural optimizations
  - Code motion
  - Loop induction variable elimination
  - Global register allocation
- Interprocedural optimizations
  - Inlining
- After that…how to implement these optimizations