

CSE401: Optimization

David Notkin
Autumn 2000

Example

```
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
```

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

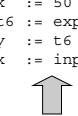
t4 := x
t5 := 3
t6 := exp(t4,t5)
y := t6
```

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()
```



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
```

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[j];
 z := z + 10000
end
```

## Intermediate code level

```
for i := 1 to 10 do
 a[i] := b[j];
end
```

```
*(fp+ioffset) := 1
_10:
 if *(fp+ioffset) > 10 goto _11
 t1 := *(fp+joffset)
 t2 := t1*4
 t3 := fp+t2
 t4 := *(t3+boffset)
 t5 := *(fp+ioffset)
 t6 := t5*4
 t7 := fp+t6
 t8 := *(fp+ioffset)
 t9 := t8+1
 *(fp+ioffset) := t9
 goto _10
_11:
```

## Loop induction variable elimination

- For-loop index is an *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

```
for i := 1 to 10 do for p := &a[1] to &a[10] do
 a[i] := a[i] + x; *p := *p + x;
end 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

```
procedure foo(n:int,x:int):int;
var sum: int, i:int;
begin
 sum := x;
 for i := 1 to n do
 sum := sum + i;
 end
 return sum;
end foo;
```