Today: more optimizations

Loop optimizations: induction variables

New DF analysis: available expressions
  Common subexpression elimination
  Copy propagation

Loop optimization
  Induction variable optimization

Reduces work in loop
  Converts to cheaper work
Induction variable
Incremented by loop invariant-amount, in every iteration
Only assignment within loop

Families of induction variables
Multiple induction variables advance in lockstep
All based on some variable v called basic
Each induction variable described by triple <s,v,d>
<Start,Var,Delta>
var = s + v * d

i = 0
s = 0
11:
if (i >= n) goto L2
j = i * 4
k = j + a
x = M[k]
s = x + x
i = i + 1
goto 11

L2:
i is basic induction variable
j,k are derived induction vars
i,j,k make family
Finding basic variables is easy
1 assignment in loop
\[ x = x + y \text{ (y loop invariant)} \text{ or } x = x + c \]

\[
i = 0 \\
s = 0 \\
L1: \\
\text{if } (i \geq n) \text{ goto } L2 \\
j = i + 4 \\
k = j + 2 \\
x = M[k] \\
s = s + x \\
i = i + 1 \\
\text{goto } L1 \\
L2:
\]

Derived induction vars k are any that:
Have 1 assignment s of the form
\[ k = j \cdot b \text{ or } k = j + c \text{ or } k = j \cdot b + c \]
b,c are loop invariant
j is induction var
exactly one def of j reaches s

j can be basic induction var, in which case
description for k is:
\[ <0,j,b> \text{ or } <c,j,1> \text{ or } <c,j,b> \]

If only assignment
\[ k = j \cdot b \text{ or } k = j + c \text{ or } k = j \cdot b + c \]

j can be derived var based on induction variable i if
No def of i between def of j and k

If descr for j is \(<s,i,d>\), descr for k is:
\[ <s \cdot b, i \cdot d > \text{ or } <s + c, i, d > \text{ or } <s \cdot b + c, i, d \cdot b > \]
Once induction variables are identified, can serve as the basis for optimization

Analyze relationships between induction variables

Try to find transformations of that yield cheaper work

Consider source code

```
s = 0;
for (int i = 0; i < n; i++)
    s += a[i];
```

Lots of induction optimization opportunities…

basic induction variable i

```
i = 0
s = 0
L1:
    if (i >= n) goto L2
    j = i * 4
    k = j + a
    q = Mem[k]
    s = s + q
    i = i + 1
    goto L1
L2:
```
Replace multiplies with adds

Consider induction variable j:<0,i,4>
Uses multiplication i*4

Initialize j to i*4 before loop
Replace j=i*4 with j = j+4

-----------

basic induction variable i

\[ i = 0 \]
\[ s = 0 \]
\[ j = i*4 \]

L1:
\[ \text{if} \ (i \geq n) \ \text{goto L2} \]
\[ j = j + 4 \]  // was j = i*4
\[ k = j + a \]
\[ x = M[k] \]
\[ s = s + x \]
\[ i = i + 1 \]
\[ \text{goto L1} \]
L2:

-----------

Generalizing j:<a,i,b> and i:<0,i,c>
Create new variable x
Shared by all induction vars y : <a,i,B>
where B = b + o(y)
o(y) possibly distinct for each y
Append x=a+i*b to pre-header
After assignment i=i+c
insert x = x + b*c
Replace assignment of j
\[ j = x + o(j) \]  \( o(j) = 0 \)
Notice b*c is loop invariant (can be hoisted)
Variable x is useless in a loop if
is dead at all exits
only occurrence of x is assignment to x

Can remove useless variables

In previous loop (after transformation):
no usage of j, can remove

Induction variables coordinated if
They stay in sync
i.e. loop invariant relation between them at usage points

Caveat: usage points cannot occur between coordinating assignments
Some variables are almost useless
  Only extra usage is comparison to loop invariant
  Has coordinated variable
Can use coordinated variable instead of almost useless variable in previous loop
  i almost useless
  Only compared to n
  Can replace with comparison to k

// i = 0
s = 0
x = 0   // i*4
nx = n*4
L1:
  if (x >= nx) goto L2
  // j = i * 4
  k = x+a    // was k = j + a
  q = M[k]
  s = s + q
  // i = i + 1
  x = x +4
goto L1
L2:

Already considered two dataflow analyses:
  Liveness
  Reaching definitions
Optimization benefits from 1 more:
  Available expressions

What expressions already computed?
  Independent of where
  Must yield same value
Consider binary expression x#y
  arbitrary operator #
x#y available at node n iff on every path
to n:
  x#y computed at least once
  and
  x not assigned after last computation
  and
  y not assigned after last computation

Uses same equations as reaching defs
  in[n] = Π out(p)
  p in pred(n)
  out[n] = defs[n] U (in[n] - kill[n])

New meaning for defs and kill
defs is now set of expressions
  What expressions were computed
  Usually 1 expr/instr
kill now describes many things
  Assume assign to y
  kills all defs involving y

a = b + c    def (b+c), kill (a*)
d = e - c    def (e-c), kill (d*)
cmp a,d      def (a-b), kill (a*)
beq L1

e = 2        def (2), kill (e*)
x = a+b       def (a+b), kill (x*)
br L2
L1: y = a+b    def (b+c), kill (a*)
L2: what expressions available at L2?
What expressions available at L2?

Available expressions supports CSE
No need to recompute any available expressions
Two approaches to exploit:
Use existing variable holding value
Introduce new variable for value

We consider both possibilities
\[
a = b + c \\
d = e - c \\
\text{cmp} \ a, \ d \\
\text{beq} \ L1 \\
e = 2 \\
x = a+b \\
\text{br} \ L2 \\
L1: \ y = a + b \\
L2: \ f = a + b + c
\]

*b+c is available expression*

\[
a = b + c \\
d = e - c \\
\text{cmp} \ a, \ d \\
\text{beq} \ L1 \\
e = 2 \\
x = a+b \\
\text{br} \ L2 \\
L1: \ y = a + b \\
L2: \ f = a + b + c
\]

*value in a, replace + with move*

\[
a = b + c \\
d = e - c \\
\text{cmp} \ a, \ d \\
\text{beq} \ L1 \\
e = 2 \\
x = a+b \\
\text{br} \ L2 \\
L1: \ y = a + b \\
L2: \ f = a + b + c
\]

*a+b also available expression*
\[ a = b + c \quad \text{def} \ (b+c), \ \text{kill} \ (a^*), \ \text{out} \ (b+c) \]
\[ d = e - c \quad \text{def} \ (e-c), \ \text{kill} \ (d^*), \ \text{out}(b+c,e-c) \]
\[ \text{cmp} \ a,d \]
\[ \text{beq} \ L1 \]
\[ e = 2 \quad \text{def} \ (2), \ \text{kill} \ (e^*), \ \text{out} \ (b+c) \]
\[ x = a+b \quad \text{def} \ (a+b), \ \text{kill} \ (x^*), \ \text{out}(b+c,a+b) \]
\[ \text{br} \ L2 \]
\[ L1: \ y = a + b \quad \text{def} \ (b+c), \ \text{kill} \ (a^*), \ \text{out}(a+b,b+c,e-c) \]
\[ L2: \ f = ? \]
\[ \text{in} \ (a+b,b+c) \]
\[ \text{no single value holds} \ a+b \]

**Solution:** Insert new temp \( t \)

---

1: \( a=b+c \)
2: \( d=e+f \)
3: \( g=a^*d \)
4: \( w=b+c \)
5: \( x=e+f \)
6: \( y=w^x \)

Can replace \( b+c \) at 4 with \( a \)
Can replace \( e+f \) at 5 with \( d \)
What about \( w^x \) at 6?
1: \(a=b+c\)  
2: \(d=e+f\)  
3: \(g=a*d\)  
4: \(w=b+c\)  
5: \(x=e+f\)  
6: \(y=w*x\)

\(w*x\) at 6 is same as \(g\)  
w same as \(a\)  
x same as \(d\)

Replace vars with earlier equivalents  
copy propagation  
Maintain list of equivalent vars  
Use dataflow to propagate function wide  
\(a=b\) records \(a\) as equivalent for \(b\)  
Replace new with old equivalent  
Subsequent \(x=a^2\) becomes \(x=b^2\)  
Generalized move coalescing

Now track equivalents:  
1: \(a=b+c\)  
1a: \(t1=a\)  
2: \(d=e+f\)  
2a: \(t2=d\)  
3: \(g=a*d\)  
4: \(w=t1\ a\)  
5: \(x=t2\ d\)  
6: \(y=w*x\ a*d\)  
a*d in available expressions