Other Optimizations

March 7, 2001

CS 132: Compiler Design

Constant Folding

- Compile-time evaluation of expressions with constant values.

```
x ← 3+5          →   x ← 8
x ← sqrt(4.0)    →   x ← 2.0
```

- Results:
  – Eliminates run-time operations

Constant Folding

- Warnings:
  – Need to make sure that the transformation preserves meaning
  – If computation results in error, need to generate code that has the error
    - The code `x ← 3/0` shouldn't cause compiler to terminate with a divide-by-zero error.
    - Need to detect overflow errors if in the language spec.
  – Must be accurate
    - Compiler's floating-point arithmetic must match the processor's floating-point arithmetic.

Constant Propagation

- Replaces uses of a variable whose value is known with the value itself.

```
i ← 8            →   i ← 8
y ← *(x+i)           y ← *(x+8)
```

- Results:
  – Moves constants to where they are used.
  – May eliminate variables (fewer registers required)
  – Permits better code on RISC machines

Copy Propagation

- Attempts to eliminate assignments of the form x ← y.

```
x ← y
z ← 2*x
w ← 2*y
```

→

```
x ← y
z ← 2*y
w ← 2*y
```

- Results:
  - May eliminate variables, run-time MOVEs
  - May expose opportunities for other optimizations

Dead Code Elimination

- Delete code whose results are never used

```
x ← y
z ← 2*y
w ← 2*y
```

→

```
z ← 2*y
w ← 2*y
```

(assuming x not otherwise used)

```
    goto L1
    z ← 2*x
L1: w ← 2*y
```

→

```
    goto L1
L1: w ← 2*y
```

- Results:
  - Shrinks code size, may reduce instruction count

Dead Code Elimination

- Warnings:
  - Must be sure that instruction being eliminated will either never be executed, or has no side-effects
    - e.g., generally can't delete function calls even if their return value never used.
    - Side-effects may include setting condition codes

Inlining

- Replace calls to procedures with copies of the body.

```
let
  fun f(x,y) =
    ...x...y...
in
  f(3,17)
end
```

→

```
let
  fun f(x,y) =
    ...x...y...
in
  ...3...17...
end
```

Inlining

- Results:
  - Avoids overhead of function call
  - May expose many more optimizations in function body since the arguments are now known
- Warnings:
  - Potential for code-blowup (and I-cache misses)
    - Automatic inlining usually based on heuristics involving procedure size, number of calls to procedure, frequency of procedure calls, constancy of arguments, ...
  - May need to rename local variables to avoid name clashes

Specialization

- Generate code for a procedure specialized for certain inputs.

```
let
  fun f(x,y) =
    ...x...y...
in
  f(3,z1);
  f(3,z2)
end
```

→

```
let
  fun f(x,y) =
    ...x...y...
  fun f3(y) =
    ...3...y...
in
  f3(z1);
  f3(z2)
end
```

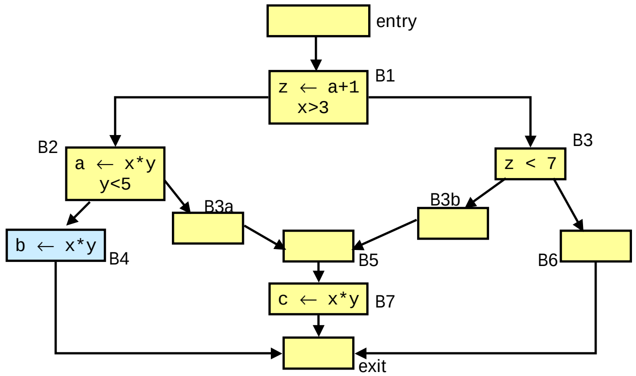
Specialization

- Results:
  - Better optimization of procedure's body
  - Fewer arguments passed at run-time
- Warnings:
  - Like inlining, subject to code blow-up

Common Subexpression Elimination (CSE)

- Avoids repeating computations known to have been already computed.
- Results
  - Removes computations from code
- Warnings
  - Cannot avoid repeating computations that might have important side-effects (exceptions, assignments, condition codes)

Common Subexpression Elimination (CSE)



Code Motion

- Many optimizations may change the order in which computations occur
  - For example, partial redundancy elimination
- Important to know that this doesn't change the code!
  - Swapped statements cannot both have visible side-effects

```
val n1 = x+y+1
val n2 = x/y
```

  - Swapped statements must be known not to interfere

```
z ← x*y
y ← y+1
```

```
x ← *p
*q ← y
```

(may require alias analysis)

Loop Invariant Removal

- Find computations that produce the same value on every occurrence of a loop, and move them out of the loop.

```
for (i=0; i<100; i++) {
   for(j=0; j<100; j++) {
      w = i*(n+2);
      a[i][j] =
         100*n + 10*w + j;
   };
};
```

→

```
t1 = n+2
t2 = 100*n
for (i=0; i<100; i++) {
   w = i*t1;
   t3 = t2 + 10*w;
   for(j=0; j<100; j++) {
      a[i][j] = t3 + j;
   };
};
```

Induction Variables

- A variable `i` is a *basic induction variable* in a loop if the only definitions of in the loop are of the form
  `i ← i+c`   or   `i ← i-c`
  where c is loop-invariant.
- A *derived induction variable* is a variable whose value is a linear function of an induction variable
  - e.g., `j ← a*i+b`.

Strength Reduction

- Replace more expensive operations with cheaper operations.
  – e.g., multiplication by addition

```
    i ← 0

L1: j ← 4*i
    k ← a+j
    x ← *k
    sum ← sum+x
    i ← i+1
    if i<10 goto L1
```

→

```
    i ← 0
    j ← -4
L1: j ← j+4
    k ← a+j
    x ← *k
    sum ← sum+x
    i ← i+1
    if i<10 goto L1
```

Induction Variable Elimination

- Minimize instances of linearly dependent variables

```
    i ← 0
    j ← -4
L1: j ← j+4
    k ← a+j
    x ← *k
    sum ← sum+x
    i ← i+1
    if i<10 goto L1
```

→

```
    t1 ← a-4
    t2 ← t1+40
L1: t1 ← t1+4
    x ← *t1
    sum ← sum+x
    if t1<t2 goto L1
```

(assuming `i`, `j`, `k` aren't used afterwards)

Loop Unrolling

- Duplicate bodies of loops

```
L1: x ← A[i]
    s ← s+x
    i ← i+4
    if i<n goto L1
```

$\not\rightarrow$

```
L1: x ← A[i]
    s ← s+x
    i ← i+4
    if i>=n goto L2
    x ← A[i]
    s ← s+x
    i ← i+4
    if i<n goto L1
L2:
```

Not like this, though

Loop Unrolling

- Duplicate bodies of loops

```
L1: x ← A[i]
    s ← s+x
    i ← i+4
    if i<n goto L1
```

→

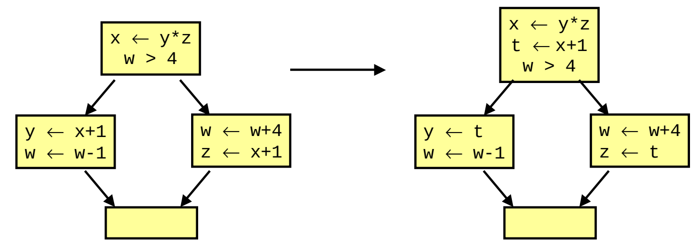
```
    if i >=n-8 goto L2
L1: x ← A[i]
    s ← s+x
    x ← A[i+4]
    s ← s+x
    i ← i+8
    if i < n-8 goto L1
L2: x ← A[i]
    s ← s+x
    i ← i+4
    if i<n goto L2
```

Loop Unrolling

- Results
  - Fewer loop termination tests
  - Fewer jumps executed
  - More potential parallelism (ILP)
- Warnings:
  - Increases code size
- Usually, loops are unrolled 2--4 times, occasionally up to 8.

Hoisting

- Find computations guaranteed to be executed, and move them as early as possible.



Hoisting

- Results:
  - More opportunities to do CSE
    - E.g., if a conditional evaluates the same expression in both branches it may be lifted out
  - May shrink code size, but little direct effect on time
- Warnings:
  - Tends to expand live ranges of variables, increasing register pressure.

Loop Fusion

- Merge multiple loops into one loop

```
for (i=0; i<n; i++) {
   b[i] = a[i] + 1.0;
};
for (j=0; j<n; j++) {
   c[j] = a[j] / 2.0;
};
```

→

```
for (i=0; i<n; i++) {
   b[i] = a[i] + 1.0;
   c[i] = a[i] / 2.0;
};
```

Loop Fusion

- Results:
  - Decreased loop overhead
  - In this case, an opportunity for CSE.
- Warnings:
  - Interleaves the execution of loop bodies; must know there are no bad dependencies.

Ordering Optimizations

- There is no "optimal" order for optimizations.
- One optimization may generate more opportunities for other optimizations
  - Either by happenstance or by design
  - Often simplifies optimizer to not worry about creating dead code or extra variable copies.
    - If we're going to run dead code elimination later, why worry?
- Order (and number of passes) a compiler heuristic
  - See Muchnick handout for suggestions