CSE P 501 – Compilers

Introduction to Optimization
Hal Perkins
Spring 2018

UW CSE P 501 Spring 2018
Agenda

• Survey some code “optimizations” (improvements)
  – Get a feel for what’s possible
• Some organizing concepts
  – Basic blocks
  – Control-flow and dataflow graph
  – Analysis vs. transformation
Optimizations

- Use added passes to identify inefficiencies in intermediate or target code
- Replace with equivalent but better sequences
  - Equivalent = “has same externally visible behavior”
  - Better can mean many things: faster, smaller, less power, etc.
- “Optimize” overly optimistic: “usually improve” is generally more accurate
  - And “clever” programmers can outwit you!
An example

\[
\begin{align*}
  x &= a[i] + b[2]; \\
  c[i] &= x - 5;
\end{align*}
\]

\[
\begin{align*}
  &\text{t1 = *(fp + ioffset); } \quad /\!\!/ \ i \\
  &\text{t2 = t1 * 4;} \\
  &\text{t3 = fp + t2;} \\
  &\text{t4 = *(t3 + aoffset); } \quad /\!\!/ \ a[i] \\
  &\text{t5 = 2;} \\
  &\text{t6 = t5 * 4;} \\
  &\text{t7 = fp + t6;} \\
  &\text{t8 = *(t7 + boffset); } \quad /\!\!/ \ b[2] \\
  &\text{t9 = t4 + t8;} \\
  &\text{*(fp + xoffset) = t9;} \quad /\!\!/ \ x = \ldots \\
  &\text{t10 = *(fp + xoffset); } \quad /\!\!/ \ x \\
  &\text{t11 = 5;} \\
  &\text{t12 = t10 - t11;} \\
  &\text{t13 = *(fp + ioffset); } \quad /\!\!/ \ i \\
  &\text{t14 = t13 * 4;} \\
  &\text{t15 = fp + t14;} \\
  &\text{*(t15 + coffset) = t12;} \quad /\!\!/ \ c[i] := \ldots
\end{align*}
\]
An example

\[
\begin{align*}
x &= a[i] + b[2]; \\
c[i] &= x - 5;
\end{align*}
\]

Strength reduction: shift often cheaper than multiply

\[
\begin{align*}
t_1 &= *(fp + ioffset); & \text{ // i} \\
t_2 &= t_1 \ll 2; & \text{ // was t}_1 \ast 4 \\
t_3 &= fp + t_2; \\
t_4 &= *(t_3 + aoffset); & \text{ // a}[i] \\
t_5 &= 2; \\
t_6 &= t_5 \ll 2; & \text{ // was t}_5 \ast 4 \\
t_7 &= fp + t_6; \\
t_8 &= *(t_7 + boffset); & \text{ // b}[2] \\
t_9 &= t_4 + t_8; \\
*(fp + xoffset) &= t_9; & \text{ // x = ...} \\
t_{10} &= *(fp + xoffset); & \text{ // x} \\
t_{11} &= 5; \\
t_{12} &= t_{10} - t_{11}; \\
t_{13} &= *(fp + ioffset); & \text{ // i} \\
t_{14} &= t_{13} \ll 2; & \text{ // was t}_{13} \ast 4 \\
t_{15} &= fp + t_{14}; \\
*(t_{15} + coffset) &= t_{12}; & \text{ // c}[i] := ... 
\end{align*}
\]
An example

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

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

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

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

Dead store (or dead assignment) elimination: remove assignments to provably unused variables.
An example

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

t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t6 = 8;    // was 2 << 2
\]

t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = ...
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset);  // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12;  // c[i] := ...

Constant folding: statically compute operations with known constant values
An example

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

\[ t1 = *(fp + ioffset); \quad // i \]
\[ t2 = t1 << 2; \]
\[ t3 = fp + t2; \]
\[ t4 = *(t3 + aoffset); \quad // a[i] \]
\[ t6 = 8; \]
\[ t7 = fp + 8; \quad // was fp + t6 \]
\[ t8 = *(t7 + boffset); \quad // b[2] \]
\[ t9 = t4 + t8; \]
\[ *(fp + xoffset) = t9; \quad // x = ... \]
\[ t10 = *(fp + xoffset); \quad // x \]
\[ t12 = t10 - 5; \]
\[ t13 = *(fp + ioffset); \quad // i \]
\[ t14 = t13 << 2; \]
\[ t15 = fp + t14; \]
\[ *(t15 + coffset) = t12; \quad // c[i] := ... \]
An example

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

```c
    t1 = *(fp + ioffset); // i
    t2 = t1 << 2;
    t3 = fp + t2;
    t4 = *(t3 + aoffset); // a[i]
    t7 = boffset + 8; // was fp + 8
    t8 = *(t7 + fp); // b[2] (was t7 + boffset)
    t9 = t4 + t8;
    *(fp + xoffset) = t9; // x = ...
    t10 = *(fp + xoffset); // x
    t12 = t10 - 5;
    t13 = *(fp + ioffset); // i
    t14 = t13 << 2;
    t15 = fp + t14;
    *(t15 + coffset) = t12; // c[i] := ...
```

Arithmetic identities: + is commutative & associative. boffset is typically a known, compile-time constant (say -32), so this enables...
An example

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

\[
t1 = *(fp + i offset); // i \\
t2 = t1 << 2; \\
t3 = fp + t2; \\
t4 = *(t3 + a offset); // a[i] \\
t7 = -24; // was b offset (-32) + 8 \\
t8 = *(t7 + fp); // b[2] \\
t9 = t4 + t8; \\
*(fp + x offset) = t9; // x = ... \\
t10 = *(fp + x offset); // x \\
t12 = t10 - 5; \\
t13 = *(fp + i offset); // i \\
t14 = t13 << 2; \\
t15 = fp + t14; \\
*(t15 + c offset) = t12; // c[i] := ...
\]
An example

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

\[
t1 = *(fp + ioffset); // i \\
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset); // a[i] \\
t7 = -24; \\
t8 = *(fp - 24); // b[2] (was t7+fp) \\
t9 = t4 + t8; \\
*(fp + xoffset) = t9; // x = ... \\
t10 = *(fp + xoffset); // x \\
t12 = t10 - 5; \\
t13 = *(fp + ioffset); // i \\
t14 = t13 << 2; \\
t15 = fp + t14; \\
*(t15 + coffset) = t12; // c[i] := ...
\]
An example

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

\[
t1 = *(fp + ioffset); \quad // \ i
\]
\[
t2 = t1 << 2;
\]
\[
t3 = fp + t2;
\]
\[
t4 = *(t3 + aoffset); \quad // \ a[i]
\]
\[
t8 = *(fp - 24); \quad // b[2]
\]
\[
t9 = t4 + t8;
\]
\[
*(fp + xoffset) = t9; \quad // x = ...
\]
\[
t10 = *(fp + xoffset); \quad // x
\]
\[
t12 = t10 - 5;
\]
\[
t13 = t1; \quad // i \ (was \ *(fp + ioffset))
\]
\[
t14 = t13 << 2;
\]
\[
t15 = fp + t14;
\]
\[
*(t15 + coffset) = t12; \quad // c[i] := ...
\]
An example

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

\[
\begin{align*}
  t1 & = *(fp + ioffset); \quad // i \\
  t2 & = t1 \ll 2; \\
  t3 & = fp + t2; \\
  t4 & = *(t3 + aoffset); \quad // a[i] \\
  t8 & = *(fp - 24); \quad \quad \quad // b[2] \\
  t9 & = t4 + t8; \\
  *(fp + xoffset) & = t9; \quad // x = \ldots \\
  t10 & = t9; \quad \quad \quad \quad \quad // x (was *(fp + xoffset)) \\
  t12 & = t10 - 5; \\
  t13 & = t1; \quad \quad \quad \quad \quad // i \\
  t14 & = t1 \ll 2; \quad // was t13 \ll 2 \\
  t15 & = fp + t14; \\
  *(t15 + coffset) & = t12; \quad // c[i] := \ldots
\end{align*}
\]
An example

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

```plaintext
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = ...  
t10 = t9;  // x
t12 = t10 - 5;
t13 = t1;  // i
t14 = t2;  // was t1 << 2

t15 = fp + t14;
*(t15 + coffset) = t12;  // c[i] := ...  
```

Common subexpression elimination
An example

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

\[ t1 = *(fp + ioffset); \quad \text{// i} \]
\[ t2 = t1 \ll 2; \]
\[ t3 = fp + t2; \]
\[ t4 = *(t3 + aoffset); \quad \text{// a[i]} \]
\[ t8 = *(fp - 24); \quad \text{// b[2]} \]
\[ t9 = t4 + t8; \]
\[ *(fp + xoffset) = t9; \quad \text{// x = ...} \]
\[ t10 = t9; \quad \text{// x} \]
\[ t12 = t9 - 5; \quad \text{// was t10 - 5} \]
\[ t13 = t1; \quad \text{// i} \]
\[ t14 = t2; \]
\[ t15 = fp + t14; \]
\[ *(t15 + coffset) = t12; \quad \text{// c[i] := ...} \]
An example

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

\[
t1 = *(fp + ioffset); // i \\
t2 = t1 << 2; \\
t3 = fp + t2; \\
t4 = *(t3 + aoffset); // a[i] \\
t8 = *(fp - 24); // b[2] \\
t9 = t4 + t8; \\
*(fp + xoffset) = t9; // x = ... \\
t10 = t9; // x \\
t12 = t9 - 5; \\
t13 = t1; // i \\
t14 = t2; \\
t15 = fp + t2; // was fp + t14 \\
*(t15 + coffset) = t12; // c[i] := ...
\]
An example

\begin{align*}
x & = a[i] + b[2]; \\
c[i] & = x - 5;
\end{align*}

\begin{align*}
t1 & = *(fp + ioffset); // i \\
t2 & = t1 << 2; \\
t3 & = fp + t2; \\
t4 & = *(t3 + aoffset); // a[i] \\
t8 & = *(fp - 24); // b[2] \\
t9 & = t4 + t8; \\
*(fp + xoffset) & = t9; // x = ... \\
t10 & = t9; // x \\
t12 & = t9 - 5; \\
t13 & = t1; // i \\
t14 & = t2; \\
t15 & = fp + t2; \\
*(t15 + coffset) & = t12; // c[i] := ...
\end{align*}
An example

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

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);      // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = ...
t12 = t9 - 5;
t15 = fp + t2;
*(t15 + coffset) = t12; // c[i] := ...
```

- Final: 3 loads (i, a[i], b[2]), 2 stores (x, c[i]), 5 register-only moves, 9 +/-, 1 shift
- Original: 5 loads, 2 stores, 10 register-only moves, 12 +/-, 3 *

- Optimizer note: we usually leave assignment of actual registers to later stage of the compiler and assume as many “pseudo registers” as we need here
Kinds of optimizations

• peephole: look at adjacent instructions
• local: look at individual *basic blocks*
  – straight-line sequence of statements
• intraprocedural: look at whole procedure
  – Commonly called “global”
• interprocedural: look across procedures
  – “whole program” analysis
  – gcc’s “link time optimization” is a version of this
• Larger scope => usually better optimization but more cost and complexity
  – Analysis is often less precise because of more possibilities
Peephole Optimization

- After target code generation, look at adjacent instructions (a “peephole” on the code stream)
  - try to replace adjacent instructions with something faster

\[
\begin{align*}
movq \%r9,16(\%rsp) \\
movq 16(\%rsp),\%r12
\end{align*}
\begin{align*}
movq \%r9,16(\%rsp) \\
movq \%r9,\%r12
\end{align*}
\]

- Jump chaining can also be considered a form of peephole optimization (removing jump to jump)
More Examples

| subq $8,%rax    | movq %r2,-8(%rax) |
| movq %r2,0(%rax) |                   |
|                | movq %r2,0(%rax)  |

- One way to do complex instruction selection
Algebraic Simplification

- "constant folding", "strength reduction"
  - $z = 3 + 4; \quad \rightarrow z = 7$
  - $z = x + 0; \quad \rightarrow z = x$
  - $z = x \times 1; \quad \rightarrow z = x$
  - $z = x \times 2; \quad \rightarrow z = x << 1$ or $z = x + x$
  - $z = x \times 8; \quad \rightarrow z = x << 3$
  - $z = x / 8; \quad \rightarrow z = x >> 3$ (only if $x \geq 0$ known)
  - $z = (x + y) - y; \quad \rightarrow z = x$ (maybe; not doubles, might change int overflow)

- Can be done at many levels from peephole on up
- Why do these examples happen?
  - Often created during conversion to lower-level IR, by other optimizations, code gen, etc.
Local Optimizations

- Analysis and optimizations within a basic block
- *Basic block*: straight-line sequence of statements
  - no control flow into or out of middle of sequence
- Better than peephole
- Not too hard to implement with reasonable IR

- Machine-independent, if done on IR
Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; unoptimized intermediate code:

```c
count = 10;
...  // count not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

```c
count = 10;
t1 = count;
t2 = 5;
t3 = t1 * t2;
x = t3;
t4 = x;
t5 = 3;
t6 = exp(t4, t5);
y = t6;
x = 7
```
Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; constant propagation:

```plaintext
count = 10;
... // count not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

```plaintext
count = 10;
t1 = 10;     // cp count
t2 = 5;
t3 = t1 * t2;  // cp t1
x = t3;
t4 = x;
t5 = 3;
t6 = exp(t4, 3);  // cp t5
y = t6;
x = 7
```
Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; constant folding:

```plaintext
count = 10;
...  // count not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

```plaintext
count = 10;
t1 = 10;
t2 = 5;
t3 = 50;  // 10*t2
x = t3;
t4 = x;
t5 = 3;
t6 = exp(t4, 3);
y = t6;
x = 7;
```
Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; repropagated intermediate code

```plaintext
count = 10;
... // count not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

```plaintext
count = 10;
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;   // cp t3
t4 = 50;   // cp x
t5 = 3;
t6 = exp(50,3); // cp t4
y = t6;
x = 7;
```
Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; refold intermediate code

```plaintext
count = 10;
...  // count not changed
x = count * 5;
y = x ^ 3;
x = 7;

count = 10;
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000; // cf 50^3
y = t6;
x = 7;
```
Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; repropagated intermediate code

```plaintext
count = 10;
... // count not changed
x = count * 5;
y = x ^ 3;
x = 7;
```
```plaintext
count = 10;
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000;
y = 125000; // cp t6
x = 7;
```
Local Dead Assignment Elimination

- If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
    Clean-up after previous optimizations, often

```plaintext
count = 10;
... // count not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

```plaintext
count = 10;
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000;
y = 125000;
x = 7;
```
Local Dead Assignment Elimination

- If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
    Clean-up after previous optimizations, often

```c
count = 10;
... // count not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

```c
count = 10;
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000;
y = 125000;
x = 7;
```
Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won’t have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

\[
\begin{array}{|c|}
\hline
... a[i] + b[i] ... \\
\hline
\end{array}
\begin{array}{|c|}
\hline
\text{t1} = *(\text{fp} + \text{ioffset}); \\
\text{t2} = \text{t1} * 4; \\
\text{t3} = \text{fp} + \text{t2}; \\
\text{t4} = *(\text{t3} + \text{aoffset}); \\
\text{t5} = *(\text{fp} + \text{ioffset}); \\
\text{t6} = \text{t5} * 4; \\
\text{t7} = \text{fp} + \text{t6}; \\
\text{t8} = *(\text{t7} + \text{boffset}); \\
\text{t9} = \text{t4} + \text{t8}; \\
\hline
\end{array}
\]
Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won’t have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

\[
\ldots a[i] + b[i] \ldots
\]

\[
\begin{align*}
t1 &= *(fp + ioffset); \\
t2 &= t1 * 4; \\
t3 &= fp + t2; \\
t4 &= *(t3 + aoffset); \\
t5 &= t1; \quad \text{// CSE} \\
t6 &= t5 * 4; \\
t7 &= fp + t6; \\
t8 &= *(t7 + boffset); \\
t9 &= t4 + t8; \\
\end{align*}
\]
Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won’t have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

```
... a[i] + b[i] ...
```

```
t1 = *(fp + ioffset);
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);
t5 = t1;
t6 = t1 * 4;  // CP
t7 = fp + t6;
t8 = *(t7 + boffset);
t9 = t4 + t8;
```
Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won’t have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

```
... a[i] + b[i] ...
```

```
t1 = *(fp + ioffset);
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);
t5 = t1;
t6 = t2;    // CSE
t7 = fp + t2; // CP
t8 = *(t7 + boffset);
t9 = t4 + t8;
```
Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won’t have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

\[
\ldots a[i] + b[i] \ldots
\]

\[
\begin{align*}
t_1 &= *\text{(fp + ioffset)}; \\
t_2 &= t_1 * 4; \\
t_3 &= \text{fp} + t_2; \\
t_4 &= *(t_3 + aoffset); \\
t_5 &= t_1; \\
t_6 &= t_2; \\
t_7 &= t_3; & // \text{CSE} \\
t_8 &= *(t_3 + boffset); & // \Phi \\
t_9 &= t_4 + t_8;
\end{align*}
\]
Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won’t have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

\[
\begin{array}{l}
\ldots a[i] + b[i] \ldots \\
\hline
\begin{align*}
t1 &= *(fp + ioffset); \\
t2 &= t1 * 4; \\
t3 &= fp + t2; \\
t4 &= *(t3 + aoffset); \\
t5 &= t1; // DAE \\
t6 &= t2; // DAE \\
t7 &= t3; // DAE \\
t8 &= *(t3 + boffset); \\
t9 &= t4 + t8;
\end{align*}
\end{array}
\]
Intraprocedural optimizations

• Enlarge scope of analysis to whole procedure
  – more opportunities for optimization
  – have to deal with branches, merges, and loops
• Can do constant propagation, common subexpression elimination, etc. at “global” level
• Can do new things, e.g. loop optimizations
• Optimizing compilers usually work at this level (-O2)
Code Motion

- Goal: move loop-invariant calculations out of loops
- Can do at source level or at intermediate code level

```c
for (i = 0; i < 10; i = i+1) {
    a[i] = a[i] + b[j];
    z = z + 10000;
}
```

```c
[ t20 = \( z \) ]
[ t21 = a[i] + b[j] ]
[ t22 = t20 + t21 ]

t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i = i+1) {
    a[i] = a[i] + t1;
    z = z + t2;
}
```

Q-40
Code Motion at IL

```
for (i = 0; i < 10; i = i+1) {
    a[i] = b[j];
}

*(fp + ioffset) = 0;
label top;
    t0 = *(fp + ioffset);
    iffalse (t0 < 10) goto done;
    t1 = *(fp + joffset);
    t2 = t1 * 4;
    t3 = fp + t2;
    t4 = *(t3 + boffset);
    t5 = *(fp + ioffset);
    t6 = t5 * 4;
    t7 = fp + t6;
    *(t7 + aoffset) = t4;
    t9 = *(fp + ioffset);
    t10 = t9 + 1;
    *(fp + ioffset) = t10;
    goto top;
label done;
```
Code Motion at IL

```c
for (i = 0; i < 10; i = i+1) {
    a[i] = b[j];
}

lll = fp + ioffset; t13 = fp + aoffset;
t12 = fp + joffset; t14 = fp + boffset
*(fp + ioffset) = 0;
label top;
    t0 = *t11;
    if (t0 < 10) goto done;
    t1 = *t12;
    t2 = t1 * 4;
    t3 = t14;
    t4 = *(t14 + t2);
    t5 = *t11;
    t6 = t5 * 4;
    t7 = t13;
    *(t13 + t6) = t4;
    t9 = *t11;
    t10 = t9 + 1;
    *t11 = t10;
    goto top;
label done;
```
Loop Induction Variable Elimination

- A special and common case of loop-based strength reduction
- For-loop index is *induction variable*
  - incremented each time around 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 loop
  - no expensive scaling in loop
  - can then do loop-invariant code motion

```c
for (i = 0; i < 10; i = i+1) {
    a[i] = a[i] + x;
}
```

=> transformed to

```c
for (p = &a[0]; p < &a[10]; p = p+4) {
    *p = *p + x;
}
```
Interprocedural Optimization

- Expand scope of analysis to procedures calling each other
- Can do local & intraprocedural optimizations at larger scope
- Can do new optimizations, e.g. inlining
Inlining: replace call with body

• Replace procedure call with body of called procedure
• Source:
  ```java
  final double pi = 3.1415927;
  double circle_area(double radius) {
    return pi * (radius * radius);
  }
  ...
  double r = 5.0;
  ...
  double a = circle_area(r);
  ```
• After inlining:
  ```java
  ...
  double r = 5.0;
  ...
  double a = pi * r * r;
  ```
• (Then what? Constant propagation/folding)
Data Structures for Optimizations

• Need to represent control and data flow
• Control flow graph (CFG) captures flow of control
  – nodes are IL statements, or whole basic blocks
  – edges represent (all possible) control flow
  – node with multiple successors = branch/switch
  – node with multiple predecessors = merge
  – loop in graph = loop
• Data flow graph (DFG) captures flow of data, e.g. def/use chains:
  – nodes are def(inition)s and uses
  – edge from def to use
  – a def can reach multiple uses
  – a use can have multiple reaching defs (different control flow paths, possible aliasing, etc.)
✓ • SSA: another widely used way of linking defs and uses
Analysis and Transformation

• Each optimization is made up of
  – some number of analyses
  – followed by a transformation

• Analyze CFG and/or DFG by propagating info forward or backward along CFG and/or DFG edges
  – merges in graph require combining info
  – loops in graph require iterative approximation

• Perform (improving) transformations based on info computed

• Analysis must be conservative/safe/sound so that transformations preserve program behavior
Summary

- Optimizations organized as collections of passes, each rewriting IL in place into (hopefully) better version
- Each pass does analysis to determine what is possible, followed by transformation(s) that (hopefully) improve the program
  - Sometimes “analysis-only” passes are helpful
  - Often redo analysis/transformations again to take advantage of possibilities revealed by previous changes
- Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write