CSE 401 – Compilers

Lecture 21: Optimization Overview

Michael Ringenburg

Winter 2013

Reminders

- Project Part 3 due tonight.
- Project Part 4 will be due in two weeks – March 15.
- Laure out of town next week – so she won't have office hours on Monday.

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 ("has the same externally visible behavior") but better sequences
  - Better can mean many things: faster, smaller, less memory, more energy-efficient, etc.
- Target-independent optimizations best done on IL code
  - Removing redundant computations, dead code, etc.
- Target-dependent optimizations best done on target code
  - Generating sequence that are more efficient on a particular machine
- "Optimize" overly optimistic: "usually improve" is generally more accurate
  - And "clever" programmers can outwit you!

An example

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

```
t1 = *(fp + ioffset);  // i
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset); // i
t14 = t13 * 4;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

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

Strength Reduction: shift often cheaper than multiply

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

Constant propagation: Replace variables with known constant value.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = 2 << 2; // was t5 << 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;  // was t10 - t11
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;
```

Dead Store (or Dead Assignment) Elimination: Remove assignments to provably unused variables.

```
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] := …
```

An example

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

Dead Store (or Dead Assignment) Elimination: Remove stores to provably unused variables.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t6 = 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] := …
```


An example

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

Constant Folding: Statically compute operations with only constant operands.

```
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] := …
```

An example

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

Constant Propagation, then Dead Store Elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t6 = 8;
t7 = fp + 8; // was 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] := …
```


An example

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

Constant Propagation, then Dead Store Elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = fp + 8;
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] := …
```

An example

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

Applying arithmetic identities: We know + is commutative & associative. boffset is typically a known compile-time constant (say, -30), so this enables …

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = boffset + 8;
t8 = *(t7 + fp);  // 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] := …
```


An example

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

… more constant folding. Which in turn enables …

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = -22; // was boffset(-30) + 8
t8 = *(t7 + fp);  // 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] := …
```

An example

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

More constant propagation and dead store elimination.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
~~t7 = -22;~~
t8 = *(fp - 22);  // 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;
```

More constant propagation and dead store elimination.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // 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] := …
```

An example

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

Common subexpression elimination: No need to compute *(fp+ioffset) twice if we know it won't change.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 – 5;
t13 = t1; // i   (was *(fp+ioffset))
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

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

Copy propagation: Replace assignment targets with their values. E.g., replace t13 with t1.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = t9; // x  (was *(fp+xoffset))
t12 = t10 – 5;
t13 = t1; // i
t14 = t1 << 2; // was t13 << 2
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

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

More copy propagation

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = t9; // x
t12 = t9 - 5;  // Was t10 - 5
t13 = t1; // i
t14 = t1 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

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

Common subexpression elimination.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = t9; // x
t12 = t9 - 5;
t13 = t1; // i
t14 = t2;  // was t1 << 2
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

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

Copy Propagation

```
t1 = *(fp + ioffset);   // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);   // a[i]
t8 = *(fp - 22);   // 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

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

Dead Assignment Elimination

```
t1 = *(fp + ioffset);   // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);   // a[i]
t8 = *(fp - 22);   // 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] := …
```

An example

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

Dead Assignment Elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t12 = t9 - 5;
t15 = fp + t2;
*(t15 + coffset) = 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]
t8 = *(fp - 22);  // 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 typically deals in "pseudo-registers" – can have as many as you want – and lets register allocator figure out optimal assignments of pseudo-registers to real registers.)

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, e.g., store and load with store and register move:

`movq %r9,12(%rsp)` `movq 12(%rsp),%r12`	`movq %r9,12(%rsp)` `movq %r9,%r12`

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

More Examples

`subq $4,%r1`
`movq %r2,0(%r1)`
`# %r1 overwritten`
	`movq %r2, -4(%r1)`

`movq 12(%rsp),%rax`
`addq $1,%rax`
`movq %rax,12(%rsp)`
`# %rax overwritten`
	`incq 12(%rsp)`

- One way to do complex instruction selection

Algebraic Simplification

- "constant folding": pre-calculate operation on constant
- "strength reduction": replace operation with a cheaper operation
- "simplification": applying algebraic identities
  - `z = 3 + 4;` → `z = 7;`
  - `z = x + 0;` → `z = x;`
  - `z = x * 1;` → `z = x;`
  - `z = x * 2;` → `z = x << 1;` or `z = x + x;`
  - `z = x * 8;` → `z = x << 3;`
  - `z = x / 8;` → `z = x >> 3;`
  - `z = (x + y) - y;` → `z = x;`
- Can be done at many levels, from peephole on up.
- Why do these examples happen?
  - Often created: Conversion to lower-level IR, Other optimizations, Code generation

Higher-level Example: Loop-based Strength Reduction

```
for (int i = 0; i < size; i++) {
   foo[i] = i;
}
```

```
for (int i = 0; i < size; i++) {
   *(foo + i * elementSize) = i;
}
```

```
t1 = 0;
for (int i = 0; i < size; i++) {
   *(foo + t1) = i;
   t1 = t1 + 8;
}
```

- Sometimes multiplication by the loop variable in a loop can be replaced by additions into a temporary accumulator
- Similarly, exponentiation can be replaced by multiplication.

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 a reasonable IR

Local Constant Propagation

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

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = count;`
	`t2 = 5;`
	`t3 = t1 * t2;`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4, t5);`
	`y = t6;`

Local Constant Propagation

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

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;    // CP count`
	`t2 = 5;`
	`t3 = 10 * 5;   // CP t1`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4, 3); // CP t5`
	`y = t6;`

Local Constant Propagation

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

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;    // CF 5 * 10`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4, 3);`
	`y = t6;`

Local Constant Propagation

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

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`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;`

Local Constant Propagation

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

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000;   // CF 50^3`
	`y = t6;`

Local Constant Propagation

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

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000;`
	`y = 125000;  // CP t6`

Local Dead Assignment Elimination

- If left side of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
  - Clean-up after previous optimizations, often
- Intermediate code after constant propagation:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`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 left side of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
  - Clean-up after previous optimizations, often
- Intermediate code after constant propagation:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	~~`t1 = 10;`~~
	~~`t2 = 5;`~~
	~~`t3 = 50;`~~
	~~`x = 50;`~~
	~~`t4 = 50;`~~
	~~`t5 = 3;`~~
	~~`t6 = 125000;`~~
	`y = 125000;`
	`x = 7;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk basic block, keeping track of available expressions

`... 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;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk 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;   // CSE`
	`t6 = t5 * 4;`
	`t7 = fp + t6;`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk 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

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk 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

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk 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;`
	`t7 = t3; // CSE`
	`t8 = *(t3 + boffset);//CP`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk 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;`~~ `// DAE`
	~~`t6 = t2;`~~ `// DAE`
	~~`t7 = t3;`~~ `// DAE`
	`t8 = *(t3 + boffset);`
	`t9 = t4 + t8;`

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 function-wide 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

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + b[j];
  z = z + (foo*bar)^2;
}
```

```
t1 = b[j];
t2 = (foo*bar)^2;
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + t1;
  z = z + t2;
}
```

Code Motion at IL

```
 *(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

```
  t11 = fp + ioffset; t13 = fp + aoffset;
  t12 = fp + joffset; t14 = fp + boffset;
  *t11 = 0;
label top;
  t0 = *t11;
  iffalse (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 an *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

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + x;  // a[i] is *(a + i*4)
}
```

  - => transformed to

```
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, and substituting actual arguments for formal parameters
- Source:

```
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:

```
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 basic blocks
  - edges represent (all possible) control flow
  - node with multiple successors = branch/switch
  - node with multiple predecessors = merge or join point
  - loop in graph = loop
- Data flow graph (DFG) capture flow of data, e.g. def/use chains:
  - nodes are def(inition)s and uses of data/variables
  - edges from defs to uses of (potentially) the same data
  - a def can reach multiple uses
  - a use can have multiple reaching defs (different control flow, possible aliasing, etc.)

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

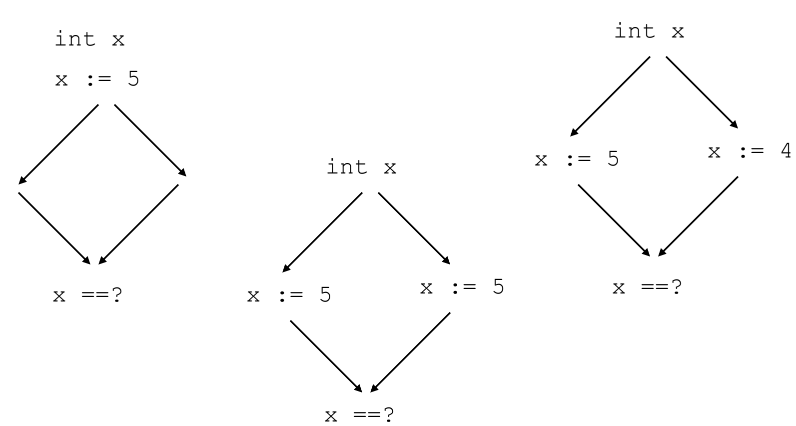
Example: Constant Propagation, Folding

- Can use either the CFG or the DFG
- CFG analysis info: table mapping each variable in scope to one of:
  - a particular constant
  - NonConstant
  - Undefined
- Transformation at each instruction:
  - If encounter an assignment of a constant to a variable, set variable as constant
  - if reference a variable that the table maps to a constant, then replace with that constant (constant propagation)
  - if r.h.s. expression involves only constants, and has no side-effects, then perform operation at compile-time and replace r.h.s. with constant result (constant folding)
- For best analysis, do constant folding as part of analysis, to learn all constants in one pass

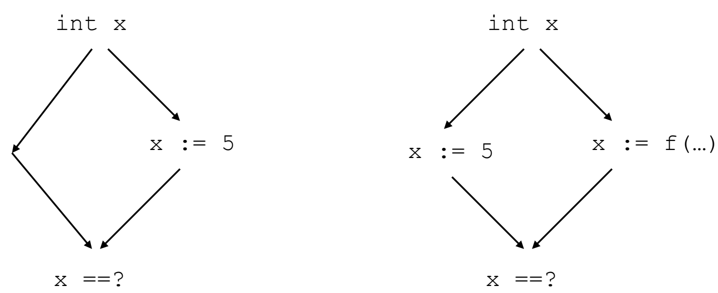
Merging data flow analysis info

- Constraint: merge results must be sound
  - if something is believed true after the merge, then it must be true no matter which path we took into the merge
  - only things true along all predecessors are true after the merge
- To merge two maps of constant information, build map by merging corresponding variable information
- To merge information about two variable
  - if one is Undefined, keep the other (uninitialized variables in many languages allowed to have any value)
  - if both are the **same** constant, keep that constant
  - otherwise, degenerate to NonConstant

Example Merges



Example Merges

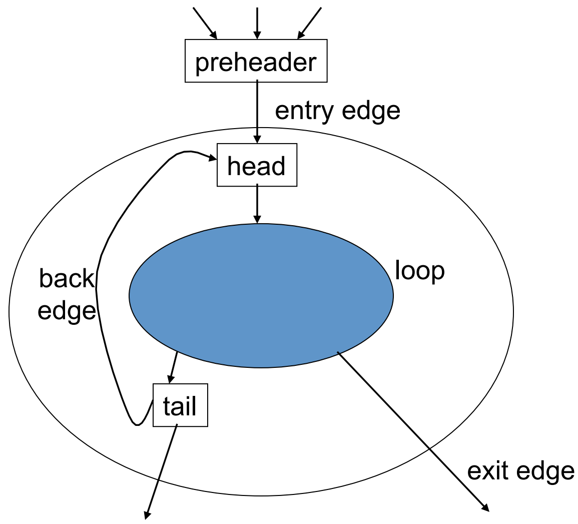


How to analyze loops

```
i = 0;
x = 10;
y = 20;
while (...) {
  // what's true here?
  ...
  i = i + 1;
  y = 30;
}
// what's true here?
... x ... i ... y ...
```

- Safe but imprecise: forget everything when we enter or exit a loop
- Precise but unsafe: keep everything when we enter or exit a loop
- Can we do better?

Loop Terminology



Optimistic Iterative Analysis

- Assuming information at loop head is same as information at loop entry
- Then analyze loop body, computing information at back edge
- Merge information at loop back edge and loop entry
- Test if merged information is same as original assumption
  - If so, then we're done
  - If not, then replace previous assumption with merged information,
  - and go back to analysis of loop body

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```


Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = 0, x = 10, y = 20

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = 0, x = 10, y = 20

i = 1, x = 10, y = 30

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = NC, x = 10, y = NC

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = NC, x = 10, y = NC

i = NC, x = 10, y = NC

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = NC, x = 10, y = NC

i = NC, x = 10, y = NC

Why does this work?

- Why are the results always conservative?
- Because if the algorithm stops, then
  - the loop head info is at least as conservative as both the loop entry info and the loop back edge info
  - the analysis within the loop body is conservative, given the assumption that the loop head info is conservative

More analyses

- Alias analysis
  - Detect when different references may or must refer to the same memory locations
- Escape analysis
  - Pointers that are live on exit from procedures
  - Pointed to data may "escape" to other procedures or threads
- Dependence analysis
  - Determining which references depend on other references
  - May analyze array subscripts that depend on loop induction variables, to determine which loop iterations depend on each other.
    - Important for loop parallelization/vectorization

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 a transformation that (hopefully) improves the program
  - Sometimes have "analysis-only" passes – produce info used by later passes
- Next week we'll look in a bit more depth at some analyses and transformations.