

CSE 401 – Compilers

Survey of Code Optimizations

Hal Perkins

Winter 2015

Administrivia

- Codegen assignment:
 - Added: please put your boot.c file in src/boot.c
 - (so we can run your compiled code with your runtime)
 - Also: we updated boot.c to use the proper formatting code for 64-bit int put. Not crucial, but change if you want
 - No change: be sure that MiniJava.main still works as specified – not moved to some package somewhere
 - Check out cse401 project web page for x86-64 info and links

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, reduce energy consumption, etc.
- Target-independent optimizations best done on IL code
 - Remove redundant computations, eliminate dead code, etc.
- Target-dependent optimizations best done on target code
 - Tailor code sequences to particular machines
- “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; // was t1 * 4  
t3 = fp + t2;  
t4 = *(t3 + aoffset); // a[i]  
t5 = 2;  
t6 = t5 << 2; // was 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 << 2; // was t13 * 4  
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 values

```
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;
```

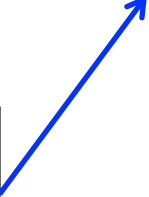
Dead store (or dead assignment) elimination:
remove assignments to provably unused variables

```
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;
```

Constant folding: statically
compute operations
with known constant values



```
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;
```

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

```
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] := ...
```

An example

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

... more constant folding,
which in turn enables ...

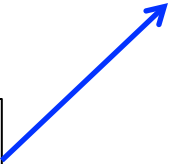


```
t1 = *(fp + ioffset); // i  
t2 = t1 << 2;  
t3 = fp + t2;  
t4 = *(t3 + aoffset); // a[i]  
t7 = -24; // was boffset (-32) + 8  
t8 = *(t7 + fp); // b[2]  
t9 = t4 + t8;  
*(fp + xoffset) = t9; // x = ...  
t10 = *(fp + xoffset); // x  
t12 = t10 - 5;  
t13 = *(fp + ioffset); // i  
t14 = t13 << 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 = -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;
```

Common subexpression
elimination – no need to
compute `*(fp+ioffset)` again
if we know it won't change



```
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 = *(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;
```

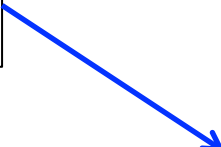
Copy propagation: replace assignment targets with their values (e.g., replace t13 with t1)

```
t1 = *(fp + ioffset); // i  
t2 = t1 << 2;  
t3 = fp + t2;  
t4 = *(t3 + aoffset); // a[i]  
t8 = *(fp - 24); // 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;
```

Common subexpression
elimination



```
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] := ...
```


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

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

Dead assignment
elimination



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

<pre>movq %r9,16(%rsp) movq 16(%rsp),%r12</pre>	<pre>movq %r9,16(%rsp) movq %r9,%r12</pre>
---	--

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

More Examples

<pre>subq \$8,%rax movq %r2,0(%rax) # %rax overwritten</pre>	<pre>movq %r2,-8(%rax)</pre>
<pre>movq 16(%rsp),%rax addq \$1,%rax movq %rax,16(%rsp) # %rax overwritten</pre>	<pre>incq 16(%rsp)</pre>

- One way to do complex instruction selection

Algebraic Simplification

- “constant folding”, “strength reduction”

– $z = 3 + 4;$ $\rightarrow z = 7$

– $z = x + 0;$ $\rightarrow z = x$

– $z = x * 1;$ $\rightarrow z = x$

– $z = x * 2;$ $\rightarrow z = x \ll 1$ or $z = x + x$

– $z = x * 8;$ $\rightarrow z = x \ll 3$

– $z = x / 8;$ $\rightarrow z = x \gg 3$ (only if $x \geq 0$ known)

– $z = (x + y) - y;$ $\rightarrow z = x$ (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:

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

```
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:

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

```
count = 10;
t1 = 10; // cp count
t2 = 5;
t3 = 10 * 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:

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

```
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

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

```
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

```
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

```
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;  
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

```
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;  
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

```
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;  
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

```
... 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

- 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; // CSE  
t6 = t5 * 4;  
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 = 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

```
... 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

- 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; // 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 “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

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

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

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

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

```
t11 = fp + ioffset; t13 = fp + aoffset;  
t12 = fp + joffset; t14 = fp + boffset  
*(fp + ioffset) = 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 *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

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

=> 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
- 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 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.)

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

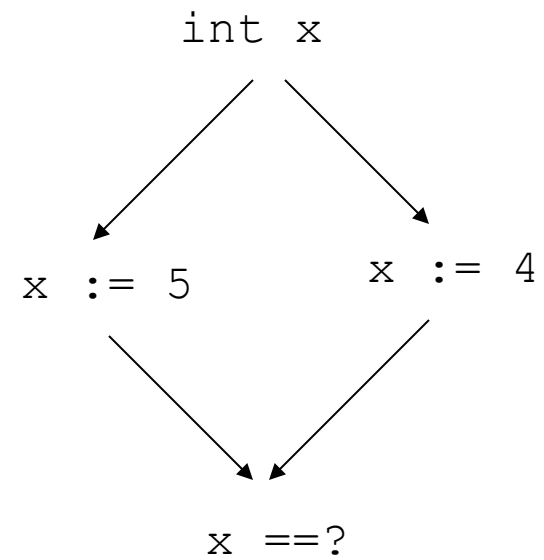
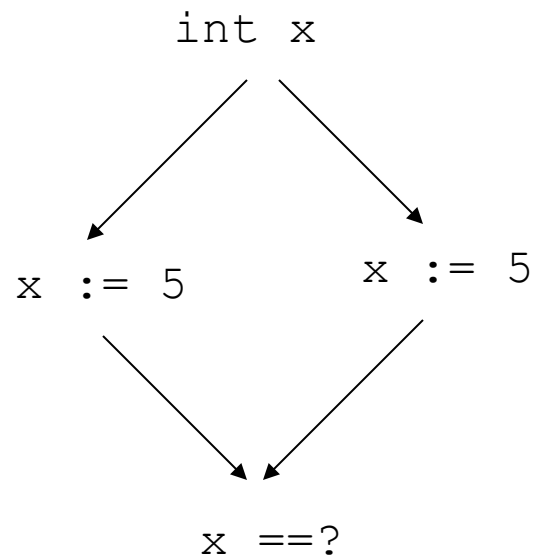
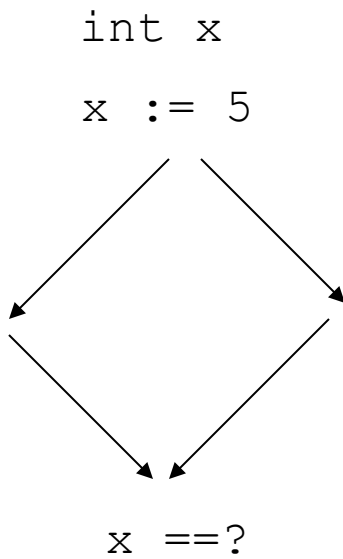
Example: Constant Propagation, Folding

- 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 an assignment of a constant to a variable, set variable as a constant with known value
 - 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

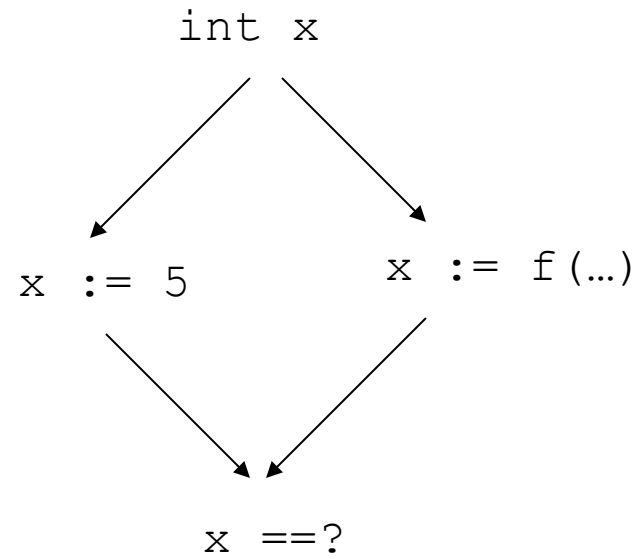
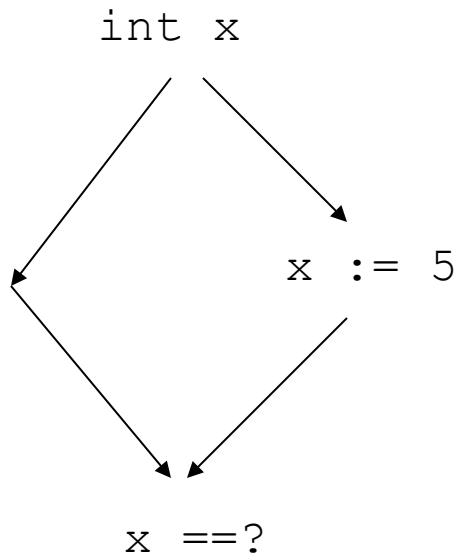
Merging data flow analysis info

- 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 variables:
 - if one is Undefined, keep the other
 - if both are the same constant, keep that constant
 - otherwise, degenerate to NonConstant (NC)

Example Merges



Example Merges

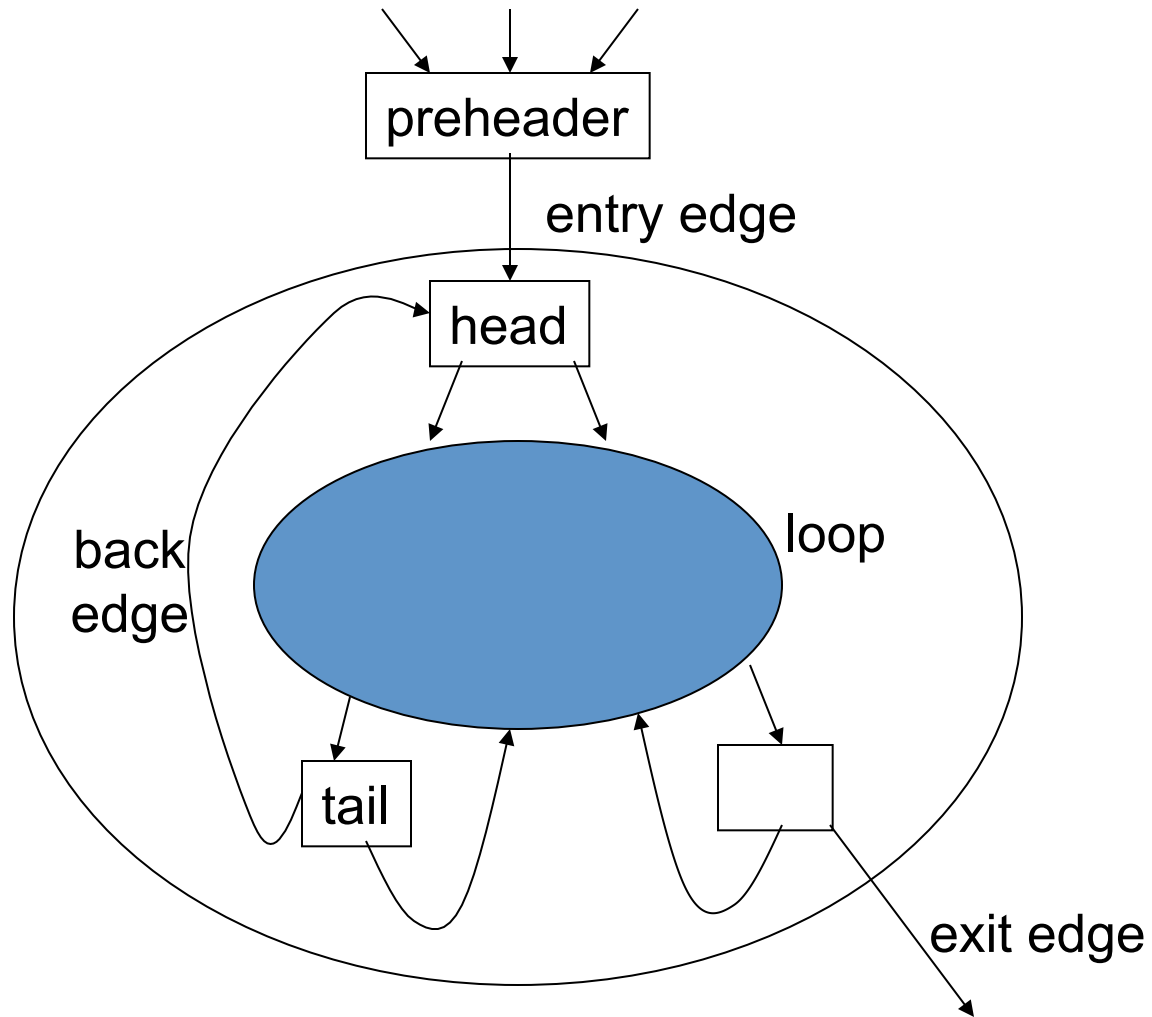


How to analyze loops

```
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

- Initially assume 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

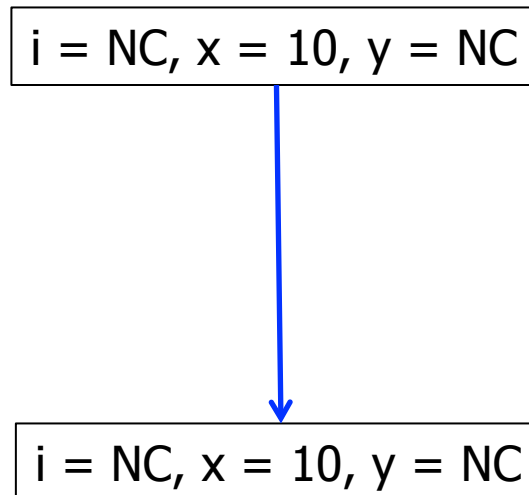
```
i = 0;  
x = 10;  
y = 20;  
while (...) {  
    // what's true here?  
    ...  
    i = i + 1;  
    y = 30; }  
// what's true here?  
... x ... i ... y ...
```

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 ...
```



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
- Will it terminate?
 - Yes, if there are only a finite number of times we can merge information before reaching worst-case info (e.g., NonConstant / NC)

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 which other references
 - One application: analyze array subscripts that depend on loop induction variables to determine which loop iterations depend on each other
 - Key analysis 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 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