Northeastern University

CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- Big thank you to UW CSE faculty member, Hal Perkins
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, ...)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, ...)

Agenda

- Optimization and transformation
    - Survey of some code “optimizations” (improvements)
    - Some organizing concepts
        - Basic blocks
        - Control-flow and dataflow graph
        - Analysis vs. transformation
        - A closer look at some common optimizing transformations

Reading:

Cooper and Torczon, chapters 4.1-4.4, 5.5, 6.2-6.5 and 7.1-7.4, 8.1-8.6

Dragon book, chapters 6.3-6.5, 7.1-7.7, 8.1-8.4, 9.1

Review: Optimizations

Review: Kinds of Optimizations

- Peephole - look at adjacent instructions
- Local - look at individual *basic blocks*
  - straight-line sequence of statements
- Intraprocedural - look at the 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

Review: Peephole Optimization

- Look at adjacent instructions (a “peephole” on the code stream)
  - try to replace adjacent instructions with something faster

`movq %r9,16(%rsp)` `movq 16(%rsp),%r12`	`movq %r9,16(%rsp)` `movq %r9,%r12`

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

Review: Algebraic Simplification

- “constant folding”, “strength reduction”
  - 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 (only if x>=0 known)
  - z = (x + y) - y; ➞ 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.

Review: 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

Review: 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`

Review: 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`

Review: 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;`

Review: 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;`

Review: 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;`

Review: 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 Intermediate Level

```
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 Intermediate Level

```
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)

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;
```

```
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] := …
```

Strength reduction: shift often cheaper than multiply

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;
```

Dead assignment 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 = 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;
```

```
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

Some Frequent Compiler Optimization Techniques

- Strength reduction – replace an "expensive" operation with an equivalent, but less expensive operation (e.g., multiplication → summation/shift)
- Constant propagation – substitute values of known constants at compile time
- Constant folding – recognize and evaluate a constant at compile time rather than run tie
- Dead assignment elimination – recognize assignments that never referenced, and remove them from the code
- Common subexpression elimination - find repetitions of same computations, and eliminate them if result won't changed
- **Code motion** - move loop-invariant calculations out of loops
- Inlining – replace some function calls with the body of the function (e.g., some getters)

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

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

Analysis and Transformation

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

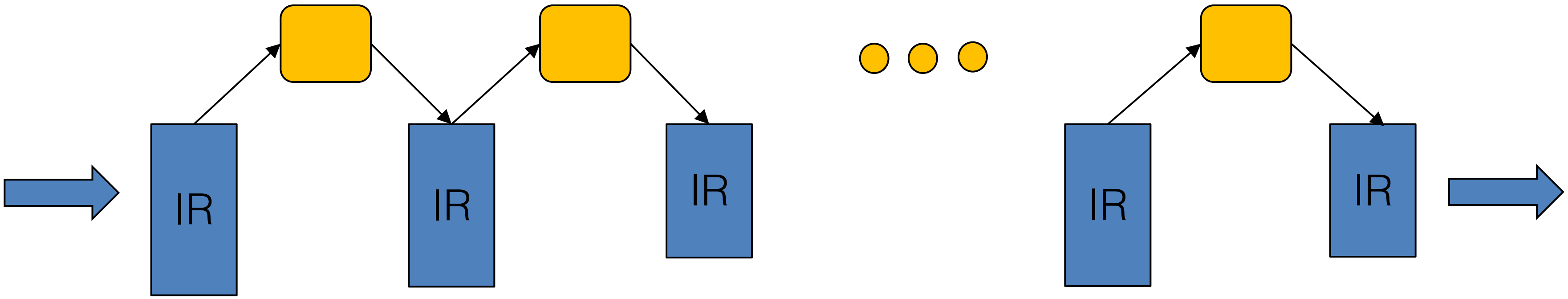
Role of Transformations

- Dataflow analysis discovers opportunities for code improvement
- Compiler rewrites the (IR) to make these improvements
  - Transformation may reveal additional opportunities for further optimization
  - May also block opportunities by obscuring information

Organizing Transformations in a Compiler

- Typically middle end consists of many phases
  - Analyze IR
  - Identify optimization
  - Rewrite IR to apply optimization
  - And repeat (50 phases in a commercial compiler is typical)
- Each individual optimization is supported by rigorous formal theory
- But no formal theory for what order or how often to apply them(!)
  - Some rules of thumb and best practices
  - May apply some transformations several times as different phases reveal opportunities for further improvement

Optimization 'Phases'



- Each optimization requires a 'pass' (linear scan) over the IR
- IR may sometimes shrink, sometimes expand
- Some optimizations may be repeated
- 'Best' ordering is heuristic
- Don't try to *beat* an optimizing compiler - you will lose!

- Note: not all programs are written by humans!
- Machine-generated code can pose a challenge for optimizers
  - eg: a single function with 10,000 statements, 1,000+ local variables, loops nested 15 deep, spaghetti of "GOTOs", etc

A Taxonomy

- Machine Independent Transformations
  - Mostly independent of target machine
    (e.g., loop unrolling will likely make it faster regardless of target)
  - "Mostly"? – e.g., vectorize only if target has SIMD ops
  - Worthwhile investment – applies to all targets
- Machine Dependent Transformations
  - Mostly concerned with instruction selection & scheduling, register allocation
  - Need to tune for different targets
  - Most of this in the back end, but some in the optimizer

Machine Independent Transformations

- Dead code elimination
  - unreachable or not actually used later
- Code motion
  - “hoist” loop-invariant code out of aloop
- Specialization
- Strength reduction
  - 2*x => x+x; @A+((i*numcols+j)*eltsize => p+=4
- Enable *other* transformations
- Eliminate redundant computations
  - Value numbering, GCSE

Machine Dependent Transformations

- Take advantage of special hardware
  - e.g., expose instruction-level parallelism (ILP)
  - e.g., use special instructions (VAX polyf; x86 sqrt, strings)
  - e.g., use SIMD instructions and registers
- Manage or hide latencies
  - e.g., tiling/blocking and loop interchange
  - Improves cache behavior – hugely important
- Deal with finite resources - # functional units
- Compilers generate for a vanilla machine, e.g., SSE2
  - But provide switches to tune (arch:AVX, arch:IA32)
  - JIT compiler knows its target architecture!

Optimizer Contracts

- Prime directive
  - No optimization will change observable program behavior!
  - This can be subtle. e.g.:
    - What is "observable"? (via IO? to another thread?)
    - Dead-Code-Eliminate a *throw*?
    - Language Reference Manual may be ambiguous/undefined/negotiable for edge cases
- Avoid harmful optimizations
  - If an optimization does not improve code significantly, don't do it: it harms throughput
  - If an optimization degrades code quality, don't do it

Is this *hoist* legal?

```
for (int i = start; i < finish; ++i) a[i] += 7;
```

```
    i = start
loop:
    if (i >= finish) goto done
    if (i < 0 || i >= a.length) throw OutOfBounds
    a[i] += 7
    goto loop
done:
```

```
    if (start < 0 || finish >= a.length) throw OutOfBounds
    i = start
loop:
    if (i >= finish) goto done
    a[i] += 7
    goto loop
done:
```

Another example: "volatile" pretty much kills all attempts to optimize

Dead Code Elimination

- If a compiler can prove that a computation has no external effect, it can be removed
  - Unreachable operations – always safe to remove
  - Useless operations – reachable, may be executed, but results not actually required
- Dead code often results from other transformations
  - Often want to do DCE several times

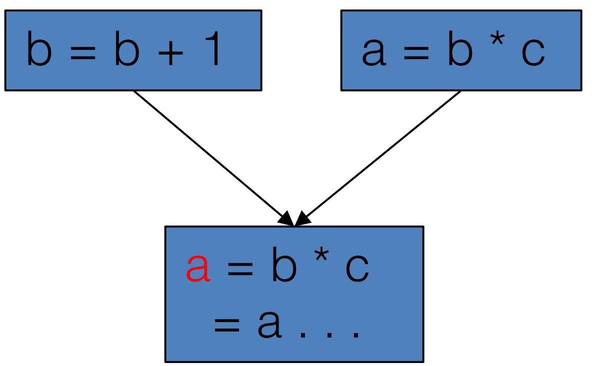
Dead Code Elimination

- Classic algorithm is similar to garbage collection
  - Pass I – Mark all useful operations
    - Instructions whose result does, or can, affect visible behavior:
      - Input or Output
      - Updates to object fields that might be used later
      - Instructions that may throw an exception (e.g.: array bounds check)
      - Calls to functions that might perform IO or affect visible behavior
      - (Remember, for many languages, compiler does not process entire program at one time – but a JIT compiler might be able to)
    - Mark all useful instructions
    - Repeat until no more changes
  - Pass II – delete all unmarked operations

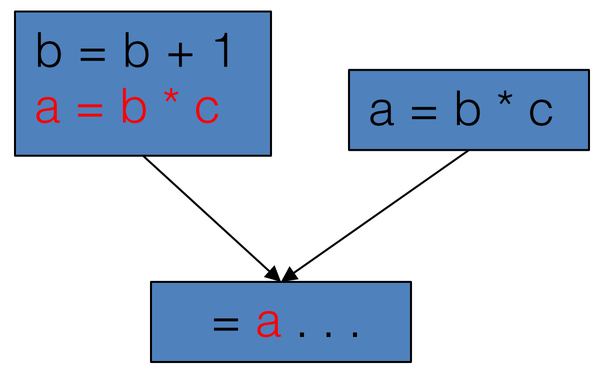
Code Motion

- Idea: move an operation to a location where it is executed less frequently
  - Classic situation: *hoist* loop-invariant code: execute once, rather than on every iteration

- Lazy code motion & *partial* redundancy



a must be re-calculated - wasteful if control took right-hand arm



Replicate, so a need not be re-calculated

Specialization I

- Idea: replace general operation in IR with more specific
  - Constant folding:
    - feet_per_minute = mph * feet_per_mile/minutes_per_hour
    - feet_per_minute = mph * 5280 / 60
    - feet_per_minute = mph * 88
  - Replacing multiplications and division by constants with shifts (when safe)
  - Peephole optimizations
    - movl $0,%eax => xorl %eax,%eax

Specialization:2 - Eliminate Tail Recursion

- Factorial - recursive
  int fac(n) = if (n <= 2) return 1; else return n * fac(n - 1);
- 'accumulating' Factorial - tail-recursive
  facaux(n, r) = if (n <= 2) return 1; else return facaux(n - 1, n*r)
  call facaux(n, 1)
- Optimize-away the call overhead; replace with simple jump
  facaux(n, r) = if (n <= 2) return 1;
  else n = n - 1; r = n*r; jump back to start of facaux
  - So replace recursive call with a loop and just one stack frame

- Issue?
  - Avoid stack overflow - good! - "observable" change?

Strength Reduction

- Classic example: Array references in a loop

```
for (k = 0; k < n; k++) a[k] = 0;
```

- Naive codegen for a[k] = 0 in loop body

```
movl $4,%eax                    // elemsize = 4 bytes
imull offset_k(%rbp),%eax       // k * elemsize
addl  offset_a(%rbp),%eax       // &a[0] + k * elemsize
mov  $0,(%eax)                  // a[k] = 0
```

- Better!

```
movl offset_a(%rbp),eax         // &a[0], once-off

movl $0,(%eax)                  // a[k] = 0
addl $4,%eax                    // eax = &a[k+1]
```

Note: *pointers* allow a user to do this directly in C or C++
Eg: for (p = a; p < a + n;) *p++ = 0;

Implementing Strength Reduction

- Idea: look for operations in a loop involving:
  - A value that does not change in the loop, the *region constant*, and
  - A value that varies systematically from iteration to iteration, the *induction variable*
- Create a new induction variable that directly computes the sequence of values produced by the original one; use an addition in each iteration to update the value

Other Common Transformations

- Inline substitution (procedure bodies)
- Cloning / Replicating
- Loop Unrolling
- Loop Unswitching

Inline Substitution - "inlining"

Class with trivial *getter*

```
class C {
  int x;
  int getx() { return x; }
}
```

Compiler *inlines* body of getx into f

```
class X {
  void f() {
    C c = new C();
    int total = c.x + 42;
  }
}
```

Method f calls getx

```
class X {
  void f() {
    C c = new C();
    int total = c.getx() + 42;
  }
}
```

- Eliminates call overhead
- Opens opportunities for more optimizations
- Can be applied to large method bodies too
- Aggressive optimizer will inline 2 or more deep
- Increases total code size (memory & cache issues)
- With care, is a huge win for OO code

Code Replication

Original

```
if (x < y) {
   p = x + y;
} else {
   p = z + 1;
}
q = p * 3;
w = y + x;
```

Replicated code

```
if (x < y) {
   p = x + y;
   q = p * 3;
   w = y + x;
} else {
   p = z + 1;
   q = p * 3;
   w = y + x;
}
```

- \+ : extra opportunities to optimize in larger basic blocks (eg: LVN)
- \- : increase total code size - may impact effectiveness of I-cache

Loop Unrolling

- Idea: replicate the loop body
  - More opportunity to optimize loop body
  - Increases chances for good schedules and instruction level parallelism
  - Reduces loop overhead (reduce test/jumps by 75%)
- Catches
  - must ensure unrolled code produces the same answer: "loop-carried dependency analysis"
  - code bloat
  - don't overwhelm registers

Loop Unroll Example

Original

```
for (i = 1, i <= n, i++) {
    a[i] = a[i] + b[i];
}
```

- Unroll 4x
- Need tidy-up loop for remainder

Unrolled

```
i = 1;
while (i + 3 <= n) {
  a[i]   = a[i]   + b[i];
  a[i+1] = a[i+1] + b[i+1];
  a[i+2] = a[i+2] + b[i+2];
  a[i+3] = a[i+3] + b[i+3];
  i += 4;
}

while (i <= n) {
    a[i] = a[i] + b[i];
    i++;
}
```

Loop Unswitching

- Idea: if the condition in an if-then-else is loop invariant, rewrite the loop by pulling the if-then-else out of the loop and generating a tailored copy of the loop for each half of the new conditional
  - After this transformation, both loops have simpler control flow – more chances for rest of compiler to do better

Loop Unswitch Example

Original

```
for (i = 1, i <= n, i++) {
     if (x > y) {
      a[i] = b[i]*x;
   } else {
      a[i] = b[i]*y;
   }
}
```

Unswitched

```
if (x > y) {
  for (i = 1; i <= n; i++) {
     a[i] = b[i]*x;
  }
} else {
  for (i = 1; i <= n; i++) {
     a[i] = b[i]*y;
  }
}
```

- IF condition does not change value in this code snippet
- No need to check x > y on every iteration
- Do the IF check once!

Summary

- Just a sampler
  - 100s of transformations in the literature
  - Will examine several in more detail, particularly involving loops
- Big part of engineering a compiler is:
  - decide which transformations to use
  - decide in what order
  - decide if & when to repeat each transformation
- Compilers offer options:
  - optimize for speed
  - optimize for codesize
  - optimize for specific target micro-architecture
  - optimize for power consumption(!)
- Competitive bench-marking will investigate many permutations

Value Numbering

Optimizations

- Big part of engineering a compiler is:
  – Deciding which transformations to use
  – Deciding in what order
  – Deciding if & when to repeat each transformation
- Compilers offer options to:
  – Optimize for speed
  – Optimize for codesize
  – Optimize for specific target micro-architecture
  – Optimize for power consumption(!)
- Competitive bench-marking will investigate many permutations

"But…"

- None of these improvements are truly "optimal"
  - Hard problems (in theory-of-computation sense)
  - Proofs of optimality assume artificial restrictions
- Best we can do is to improve things
  - Most (much?) (some?) of the time
  - Realistically: try to do better for common idioms both in the code and on the machine

Issues (1)

- Safety – transformation must not change program meaning
  - Must generate correct results
  - Can't generate spurious errors
  - Optimizations must be conservative
  - Large part of analysis goes towards proving safety
  - Can pay off to speculate (be optimistic) but then need to recover if reality is different

Issues (2)

- Profitability
  - If a transformation is possible, is it profitable?
  - Example: loop unrolling
    - Can increase amount of work done on each iteration, i.e., reduce loop overhead
    - Can eliminate duplicate operations done on separate iterations

Issues (3)

- Downside risks
  - Even if a transformation is generally worthwhile, need to think about potential problems
  - For example:
    - Transformation might need more temporaries, putting additional pressure on registers
    - Increased code size could cause cache misses, or, in bad cases, increase page working set

Example: Redundancy Elimination

- An expression x+y is *redundant* at a program point if and only if, along every path from the procedure's entry, it has been evaluated and its constituent subexpressions (x and y) have not been redefined
- If the compiler can prove the expression is redundant:
  – Can store the result of the earlier evaluation
  – Can replace the redundant computation with a reference to the earlier (stored) result

Value Numbering

- Technique for eliminating redundant expressions:
  - Assign an identifying number VN(n) to each expression
  - VN(x + y) = VN(j) if x+y and j have the same value
  - Use hashing over value numbers for efficiency
- Old idea (Balke 1968, Ershov 1954)
  - Invented for low-level, linear IRs
  - Equivalent methods exist for tree IRs, e.g., build a DAG

Local Value Numbering

- Algorithm
  - For each operation o = <op, o1,o2> in a block
    1. Get value numbers for operands from hash lookup
    2. Hash <op, VN(o1), VN(o2)> to get a value number for o (If op is commutative, sort VN(o1), VN(o2) first)
    3. If o already has a value number, replace o with a reference to the value
    4. If o1 and o2 are constant, evaluate o at compile time and replace with an immediate load
- If hashing behaves well, this runs in linear time

Example

Code

```
a = x + y
b = x + y
a = 17
c = x + y
```

Rewritten

Bug in Simple Example

- If we use the original names, we get in trouble when a name is reused
- Solutions
  - Be clever about which copy of the value to use (e.g., use c=b in last statement)
  - Create an extra temporary
  - Rename around it (best!)

Renaming

- Idea: give each value a unique name

  a_i^j means i^{th} definition of a with VN = j
- Somewhat complex notation, but meaning is clear
- This is the idea behind SSA (Static Single Assignment)
  - Popular modern IR – exposes many opportunities for optimizations

Example Revisited

Code

a = x + y

b = x + y

a = 17

c = x + y

Rewritten

Simple Extensions to Value Numbering

- Constant folding
  - Add a bit that records when a value is constant
  - Evaluate constant values at compile time
  - Replace op with load immediate
- Algebraic identities: x+0, x*1, x-x, …
  - Many special cases
    - Switch on op to narrow down checks needed
    - Replace result with input VN

Larger Scopes

- This algorithm works on straight-line blocks of code (basic blocks)
  - Best possible results for single basic blocks
  - Loses all information when control flows to another block
- To go further, we need to represent multiple blocks of code and the control flow between them

Optimization Categories (1)

- *Local methods*
  - Usually confined to basic blocks
  - Simplest to analyze and understand
  - Most precise information

Optimization Categories (2)

- *Superlocal methods*
  - Operate over *Extended Basic Blocks* (EBBs)
    - An EBB is a set of blocks b_1, b_2, …, b_n where b_1 has multiple predecessors and each of the remaining blocks b_i ($2 \le i \le n$) have only b_{i-1} as its unique predecessor
    - The EBB is entered only at b_1, but may have multiple exits
    - A single block b_i can be the head of multiple EBBs (these EBBs form a tree rooted at b_i)
  - Use information discovered in earlier blocks to improve code in successors

Optimization Categories (3)

- *Regional methods*
  - Operate over scopes larger than an EBB but smaller than an entire procedure/ function/method
  - Typical example: loop body
  - Difference from superlocal methods is that there may be merge points in the graph (i.e., a block with two or more predecessors)
    - Facts true at merge point are facts known to be true on all possible paths to that point

Optimization Categories (4)

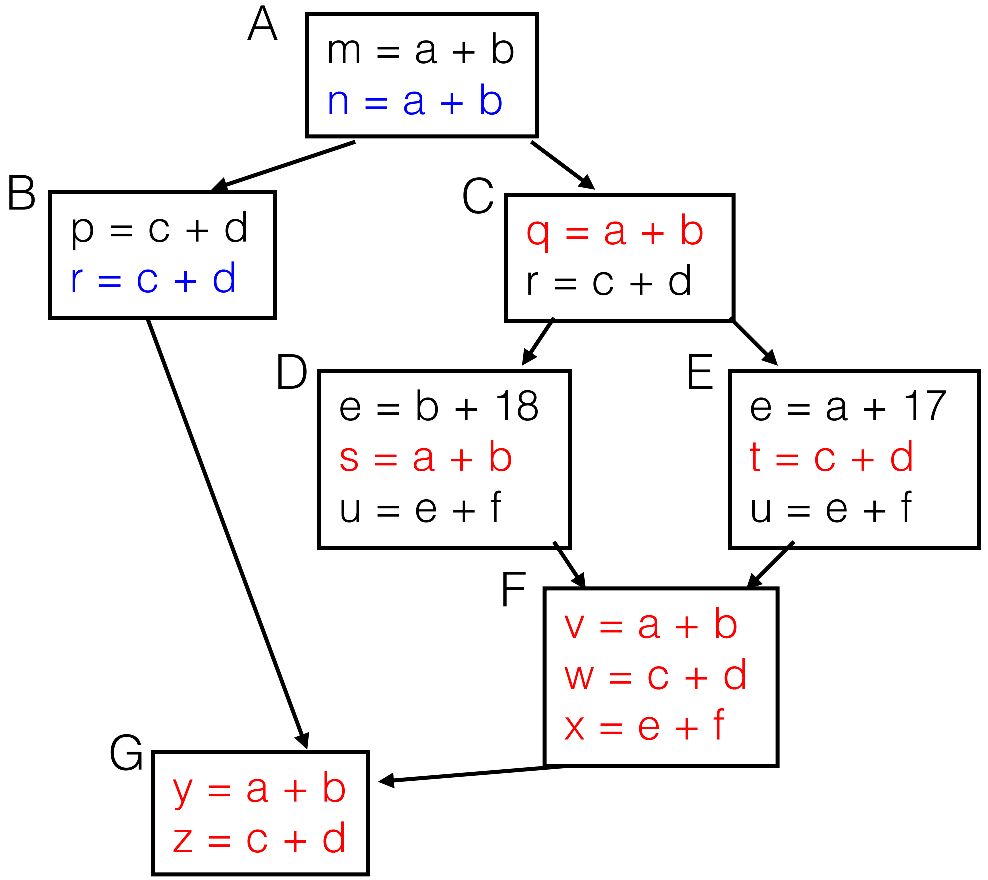
- *Global methods*
  - Operate over entire procedures
  - Sometimes called *intraprocedural* methods
  - Motivation is that local optimizations sometimes have bad consequences in larger context
  - Procedure/method/function is a natural unit for analysis, separate compilation, etc.
  - Almost always need global *data-flow* analysis information for these

Optimization Categories (5)

- *Whole-program methods*
  - Operate over more than one procedure
  - Sometimes called *interprocedural* methods
  - Challenges: name scoping and parameter binding issues at procedure boundaries
  - Classic examples: inline method substitution, interprocedural constant propagation
  - Common in aggressive JIT compilers and optimizing compilers for object-oriented languages

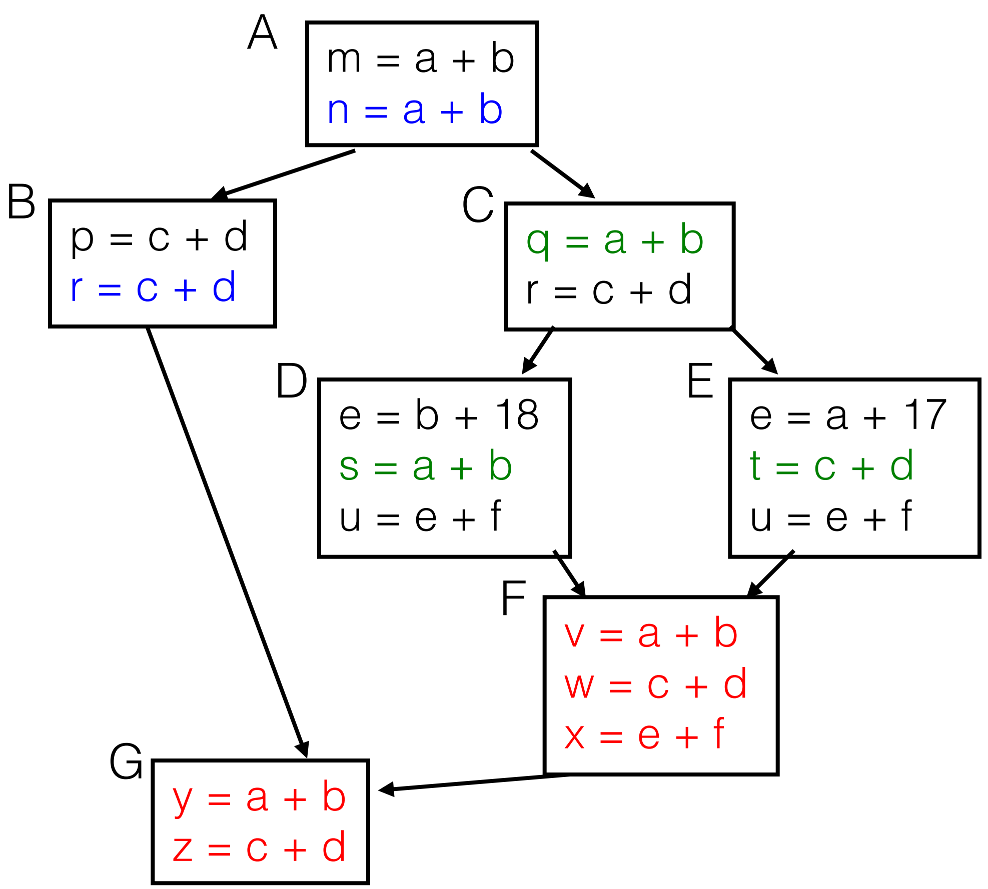
Value Numbering Revisited

- Local Value Numbering
  - 1 block at a time
  - Strong local results
  - No cross-block effects
- Missed opportunities



Superlocal Value Numbering

- Idea: apply local method to EBBs
  - {A,B}, {A,C,D}, {A,C,E}
- Final info from A is initial info for B, C; final info from C is initial for D, E
- Gets reuse from ancestors
- Avoid reanalyzing A, C
- Doesn't help with F, G



SSA Name Space

- Two Principles
  - Each name is defined by exactly one operation
  - Each operand refers to exactly one definition
- Need to deal with merge points
  - Add Φ functions at merge points to reconcile names
  - Use subscripts on variable names for uniqueness

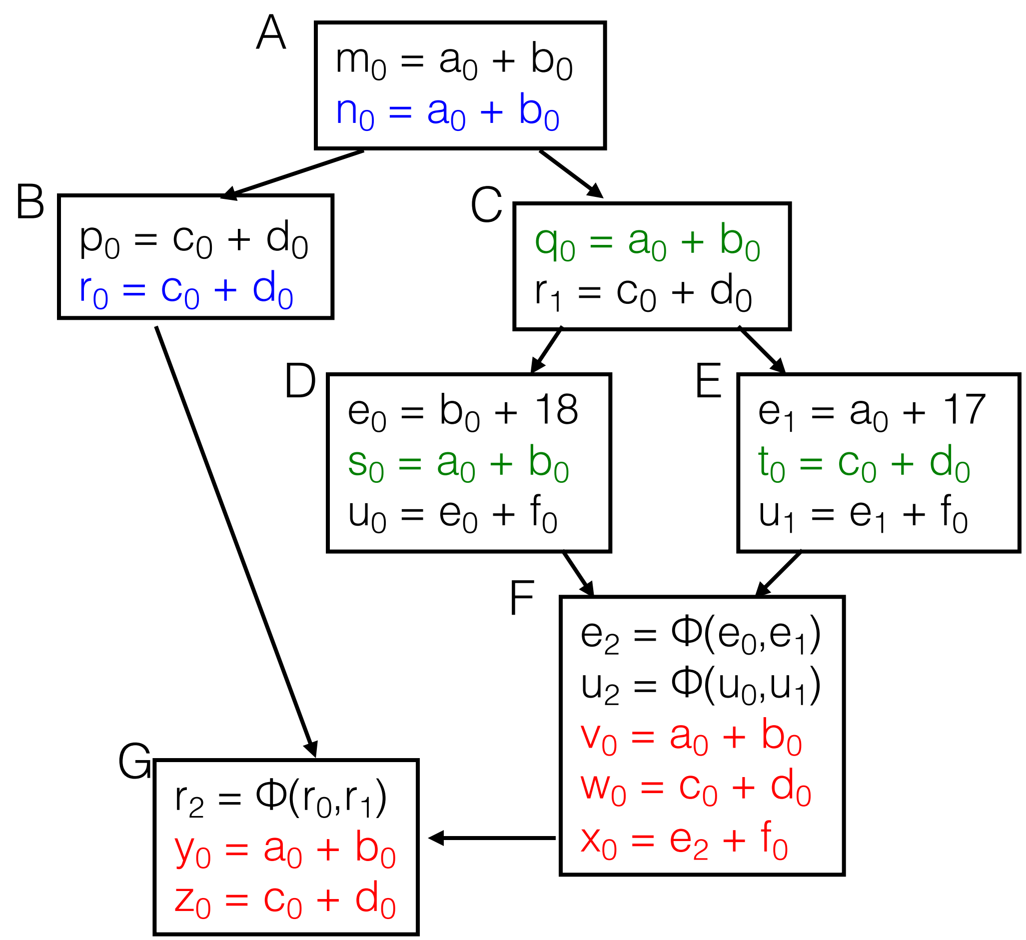
SSA Name Space (from before)

Code	Rewritten
$a_0^3 = x_0^1 + y_0^2$	$a_0^3 = x_0^1 + y_0^2$
$b_0^3 = x_0^1 + y_0^2$	$b_0^3 = a_0^3$
$a_1^4 = 17$	$a_1^4 = 17$
$c_0^3 = x_0^1 + y_0^2$	$c_0^3 = a_0^3$

- Unique name for each definition
- Name ⇔ VN
- a_0^3 is available to assign to c_0^3

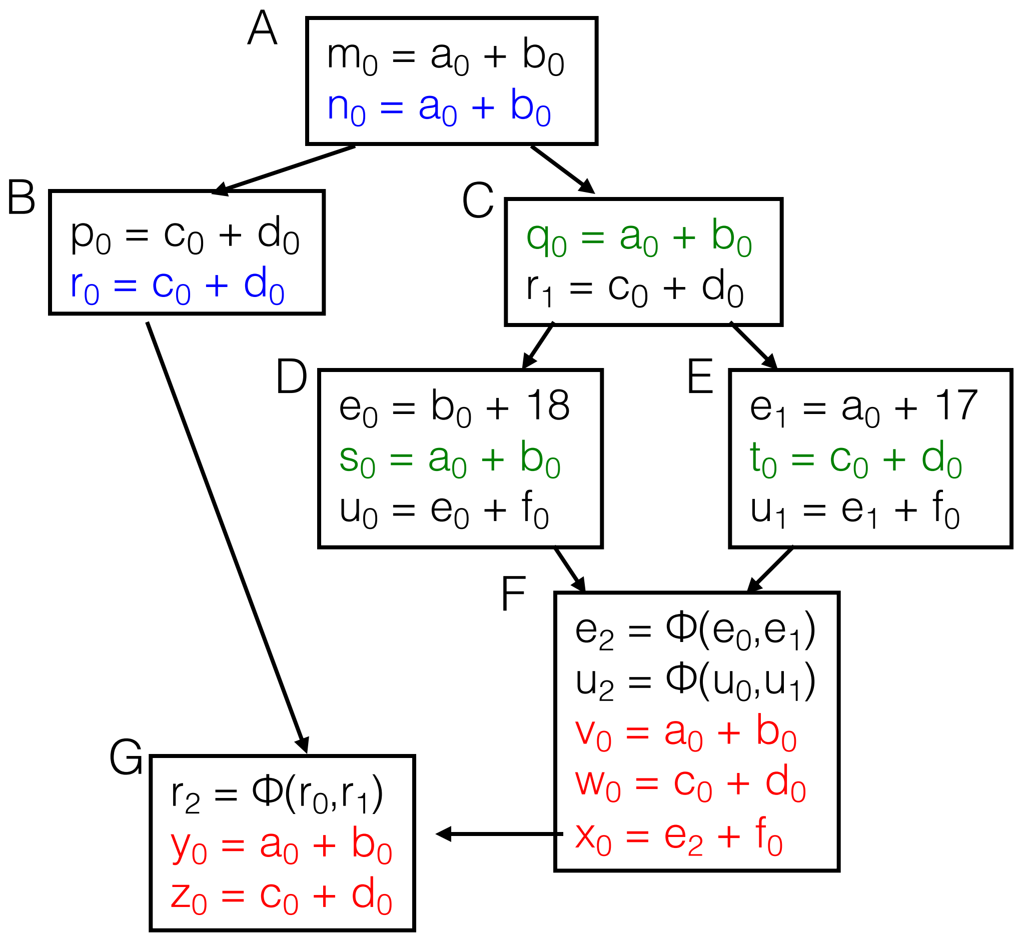
Superlocal Value Numbering with All Bells & Whistles

- Finds more redundancies
- Little extra cost
- Still does nothing for F and G



Larger Scopes

- Still have not helped F and G
- Problem: multiple predecessors
- Must decide what facts hold in F and in G
  - For G, combine B & F?
  - Merging states is expensive
  - Fall back on what we know



Dominators

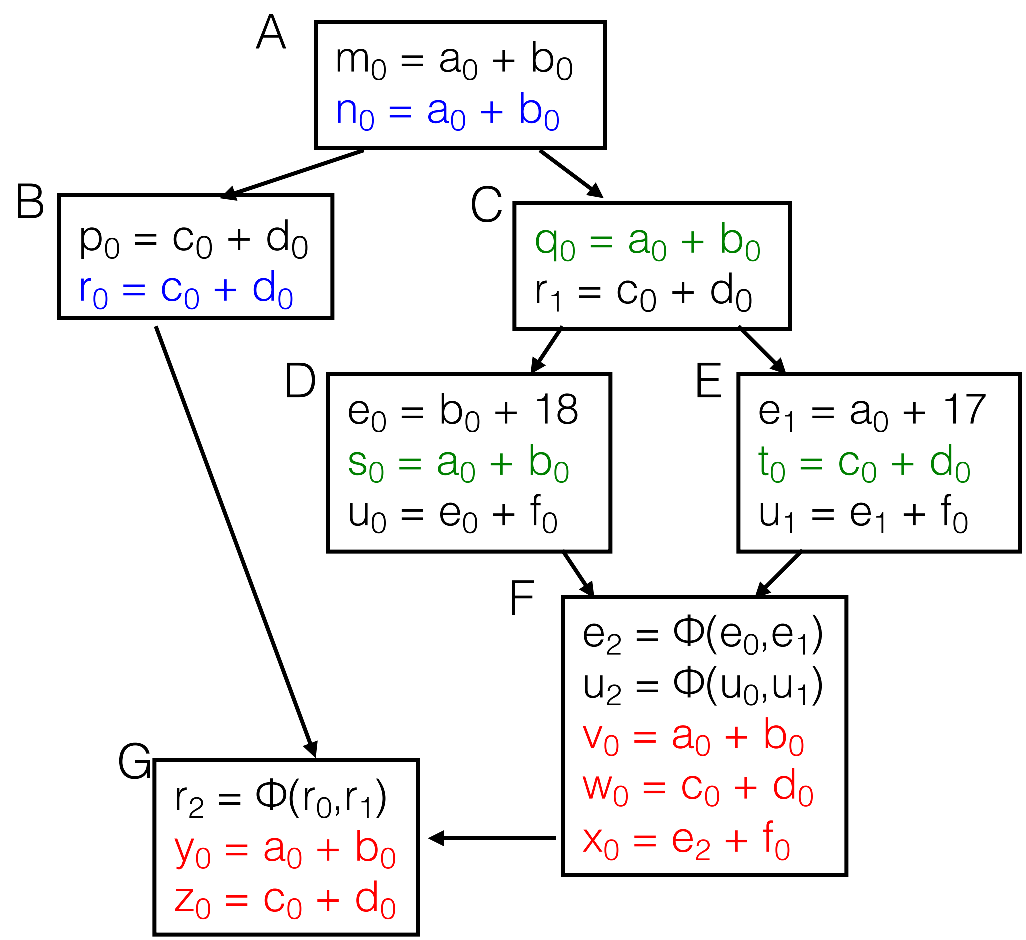
- Definition
  - x *dominates* y if and only if every path from the entry of the control-flow graph to y includes x
- By definition, x dominates x
- Associate a Dom set with each node
  - $|\ Dom(x)\ | \geq 1$
- Many uses in analysis and transformation
  - Finding loops, building SSA form, code motion

Immediate Dominators

- For any node x, there is a y in Dom(x) closest to x
- This is the *immediate dominator* of x
  - Notation: IDom(x)

Dominator Sets

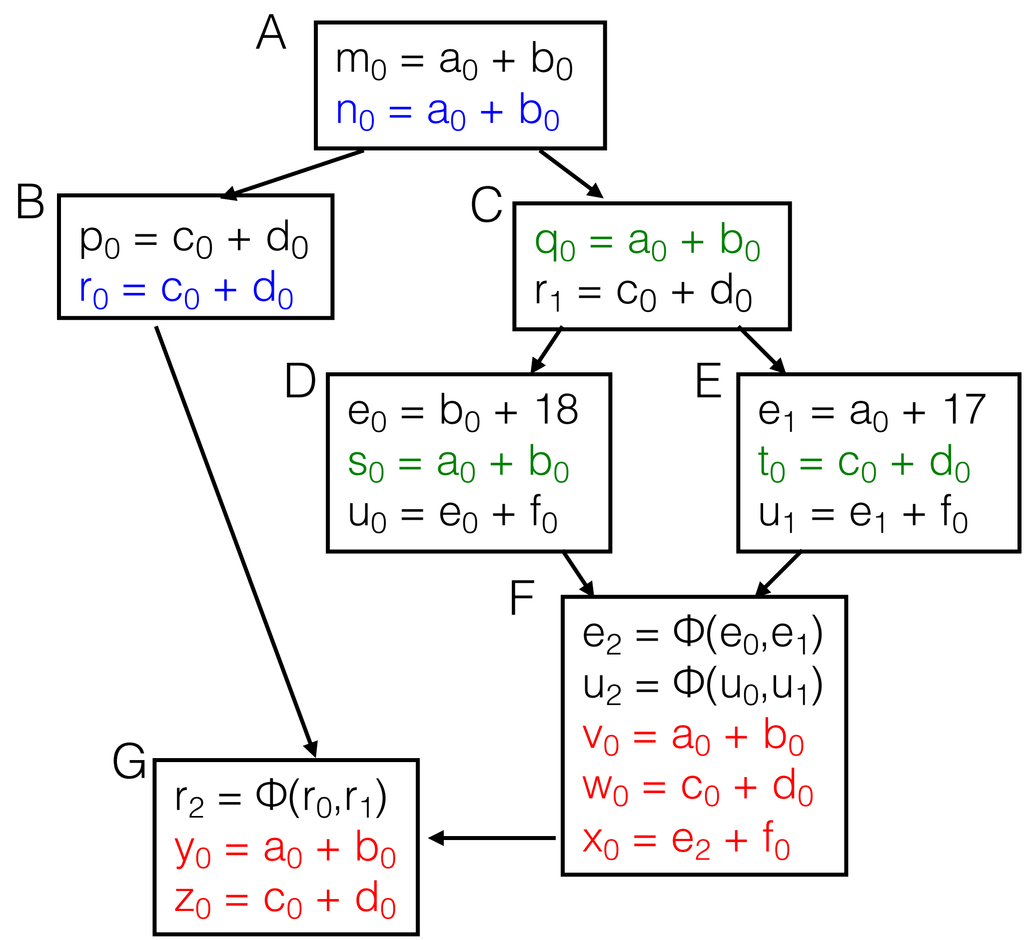
Block Dom IDom



Note that the IDOM relation defines a tree!

Dominator Value Numbering

- Still looking for a way to handle F and G
- Idea: Use info from IDom(x) to start analysis of x
  - Use C for F and A for G
- Dominator VN Technique (DVNT)

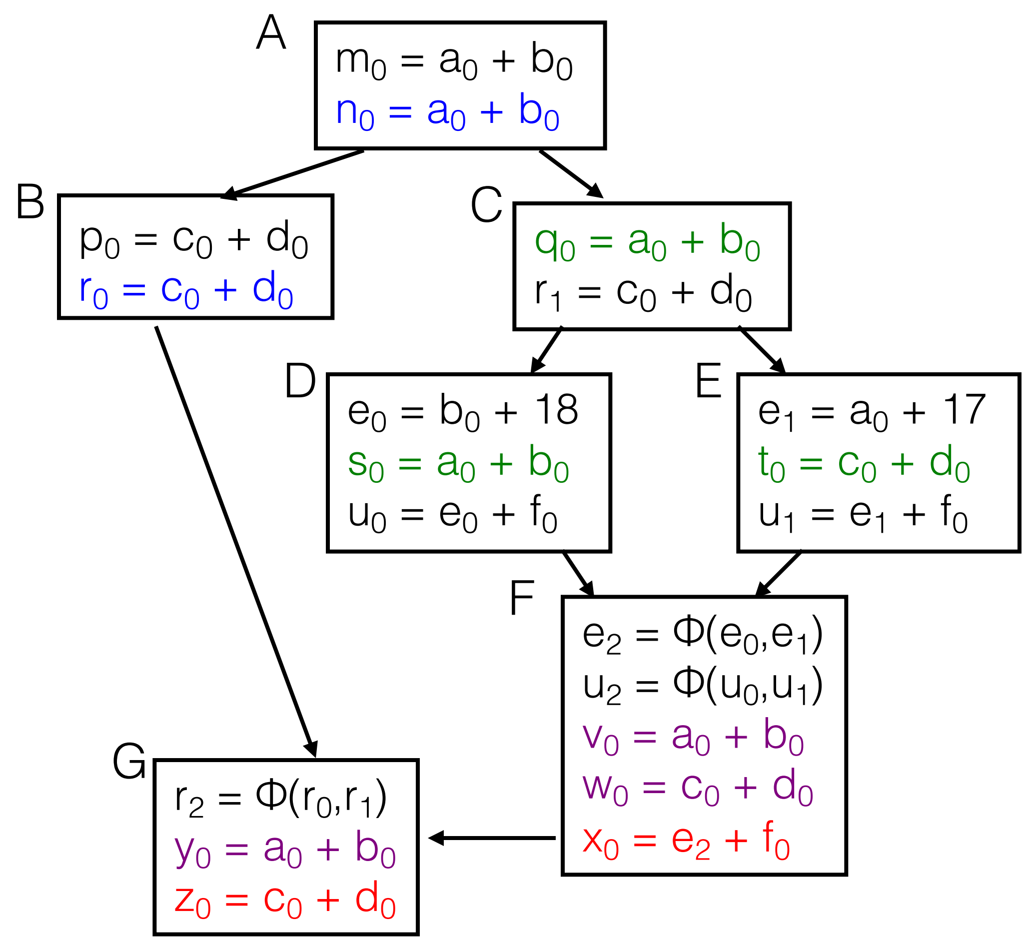


DVNT Algorithm

- Use superlocal algorithm on extended basic blocks
  - Use scoped hash tables & SSA name space as before
- Start each node with table from its IDOM
- No values flow along back edges (i.e., loops)
- Constant folding, algebraic identities as before

Dominator Value Numbering

- Advantages
  - Finds more redundancy
  - Little extra cost
- Shortcomings
  - Misses some opportunities (common calculations in ancestors that are not IDOMs)
  - Doesn't handle loops or other back edges



The Story So Far…

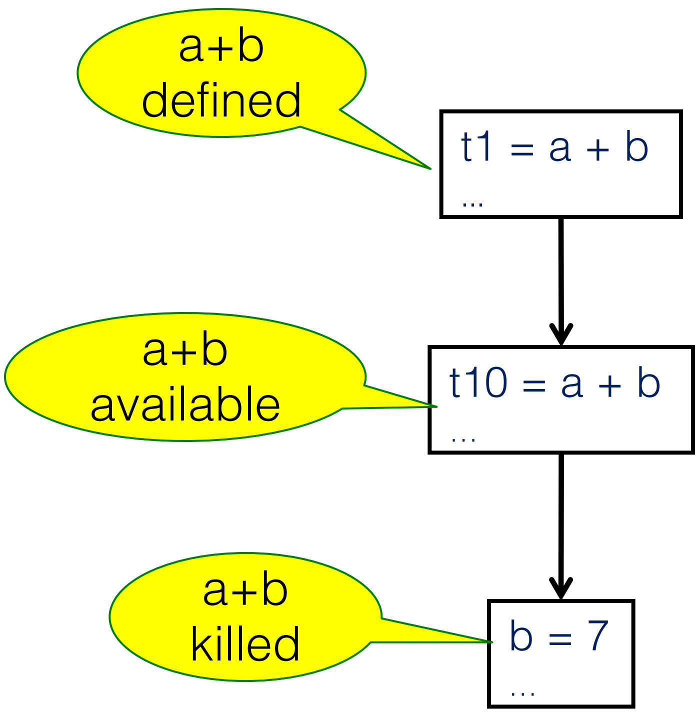
- Local algorithm
- Superlocal extension
  - Some local methods extend cleanly to superlocal scopes
- Dominator VN Technique (DVNT)
- All of these propagate along forward edges
- None are global

Available Expressions

- Goal: use dataflow analysis to find common sub-expressions whose range spans basic blocks
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – use a copy operation

"Available" and Other Terms

- An expression *e* is *defined* at point *p* in the CFG if its value is computed at *p*
  - Sometimes called *definition site*
- An expression *e* is *killed* at point *p* if one of its operands is defined at *p*
  - Sometimes called *kill site*
- An expression *e* is *available* at point *p* if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*



Available Expression Sets

- To compute available expressions, for each block *b*, define
  - AVAIL(b) – the set of expressions available on entry to *b*
  - NKILL(b) – the set of expressions not killed in *b*
    - i.e., all expressions in the program *except* for those killed in *b*
  - DEF(b) – the set of expressions defined in *b* and not subsequently killed in *b*

Computing Available Expressions

- AVAIL(b) is the set

  $\text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$

  - preds(b) is the set of b's predecessors in the CFG
  - The set of expressions available on entry to *b* is the set of expressions that were available at the end of *every* predecessor basic block *x*
  - The expressions available on exit from block *b* are those defined in *b* or available on entry to *b* and not killed in *b*
- This gives a system of simultaneous equations – a dataflow problem

Name Space Issues

- In previous value-numbering algorithms, we used a SSA-like renaming to keep track of versions
- In global dataflow problems, we use the original namespace
  - we require a+b have the same value along *all* paths to its use
  - If a or b is updated along *any* path to its use, then a+b has the "wrong" value
  - so original names are exactly what we want
- The KILL information captures when a value is no longer available

Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(*b*) and NKILL(*b*)
    - This only needs to be done once for each block *b* and depends only on the statements in *b*
  - Iteratively calculate AVAIL(*b*) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm

Computing DEF and NKILL (1)

- For each block *b* with operations o_1, o_2, …, o_k

```
KILLED = ∅      // killed variables, not expressions
DEF(b) = ∅
for i = k to 1   // note: working back to front
   assume oi is "x = y + z"
   if (y ∉ KILLED and z ∉ KILLED)
      add "y + z" to DEF(b)
   add x to KILLED
…
```

Computing DEF and NKILL (2)

- After computing DEF and KILLED for a block *b*, compute set of all expressions in the program not killed in *b*

NKILL(*b*) = { all expressions }

for each expression *e*

    for each variable $v \in$ e

        if $v \in$ KILLED then

            NKILL(*b*) = NKILL(*b*) - *e*

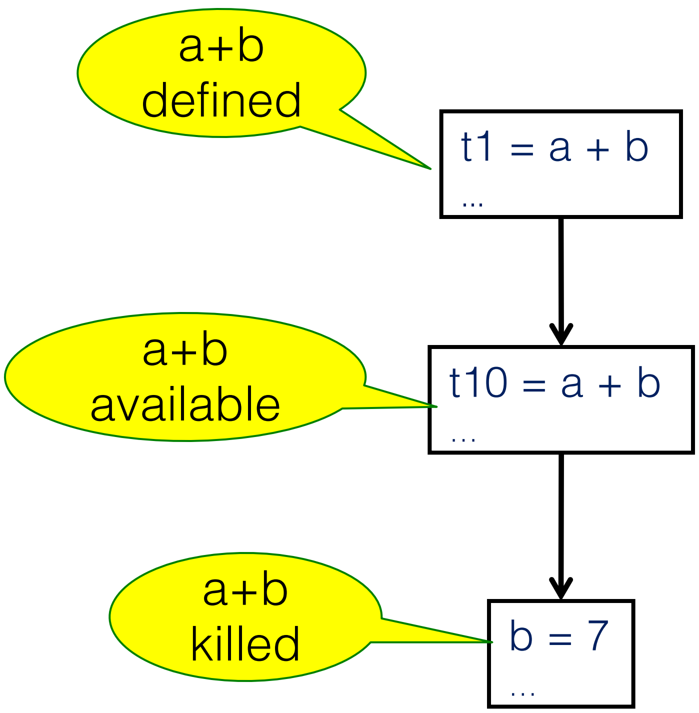
Data Flow Analysis

Available Expressions

- Goal: use dataflow analysis to find common sub-expressions whose range spans basic blocks
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – use a copy operation

"Available" and Other Terms

- An expression *e* is *defined* at point *p* in the CFG if its value is computed at *p*
  - Sometimes called *definition site*
- An expression *e* is *killed* at point *p* if one of its operands is defined at *p*
  - Sometimes called *kill site*
- An expression *e* is *available* at point *p* if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*



Available Expression Sets

- To compute available expressions, for each block *b*, define
  - AVAIL(b) – the set of expressions available on entry to *b*
  - NKILL(b) – the set of expressions not killed in *b*
    - i.e., all expressions in the program *except* for those killed in *b*
  - DEF(b) – the set of expressions defined in *b* and not subsequently killed in *b*

Computing Available Expressions

- AVAIL(b) is the set

  $\text{AVAIL}(b) = \cap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$

  – preds(b) is the set of b's predecessors in the CFG
  – The set of expressions available on entry to *b* is the set of expressions that were available at the end of *every* predecessor basic block *x*
  – The expressions available on exit from block *b* are those defined in *b* or available on entry to *b* and not killed in *b*
- This gives a system of simultaneous equations – a dataflow problem

Name Space Issues

- In previous value-numbering algorithms, we used a SSA-like renaming to keep track of versions
- In global dataflow problems, we use the original namespace
  - we require a+b have the same value along *all* paths to its use
  - If a or b is updated along *any* path to its use, then a+b has the "wrong" value
  - so original names are exactly what we want
- The KILL information captures when a value is no longer available

Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(*b*) and NKILL(*b*)
    - This only needs to be done once for each block *b* and depends only on the statements in *b*
  - Iteratively calculate AVAIL(*b*) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm

Computing DEF and NKILL (1)

- For each block *b* with operations o_1, o_2, …, o_k

  KILLED = ∅ // killed *variables*, not expressions

  DEF(b) = ∅

  for i = k to 1 // note: working back to front

  assume o_i is "x = y + z"

  if (y ∉ KILLED and z ∉ KILLED)

  add "y + z" to DEF(b)

  add x to KILLED

  …

Computing DEF and NKILL (2)

- After computing DEF and KILLED for a block *b*, compute set of all expressions in the program not killed in *b*

NKILL(*b*) = { all expressions }

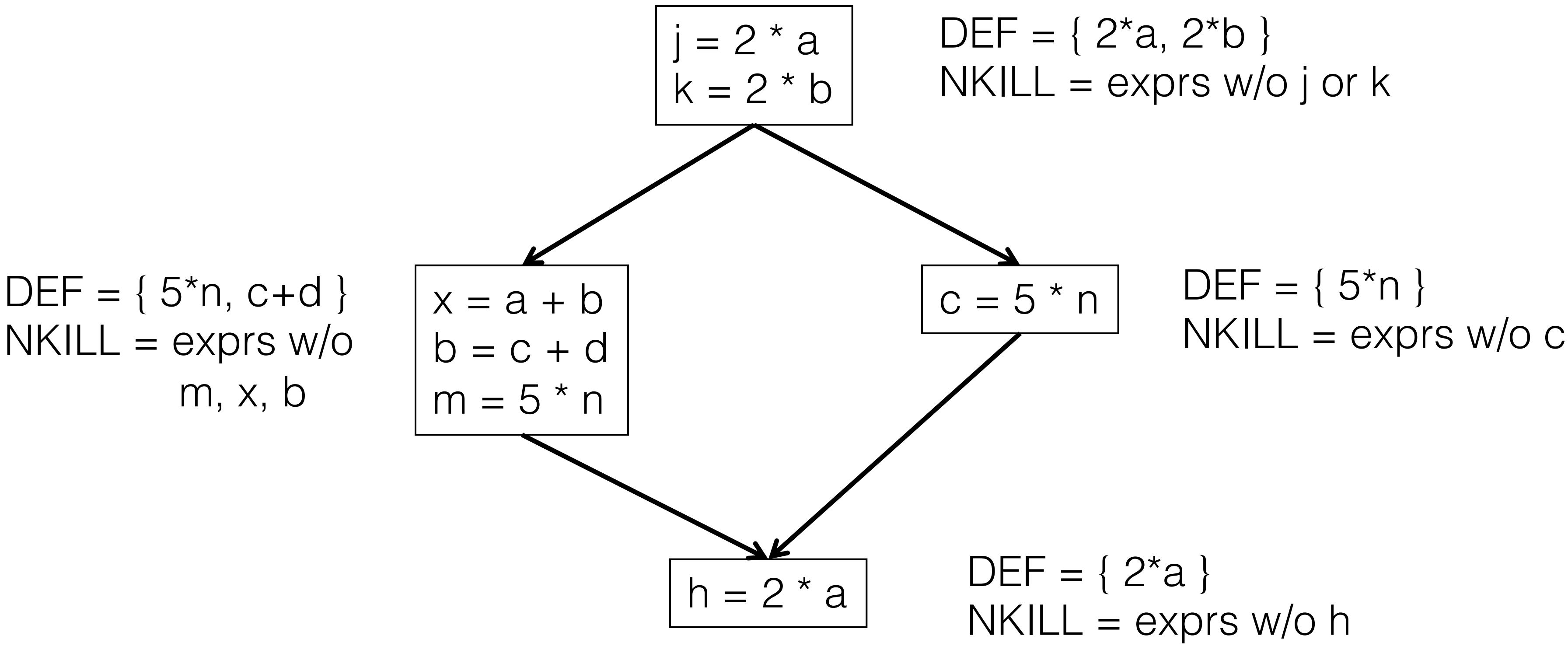
for each expression *e*

  for each variable $v \in$ e

    if $v \in$ KILLED then

      NKILL(*b*) = NKILL(*b*) - *e*

Example: Compute DEF and NKILL



Computing Available Expressions

Once DEF(b) and NKILL(b) are computed for all blocks b

Worklist = { all blocks b_i }

while (Worklist ≠ ∅)

remove a block b from Worklist

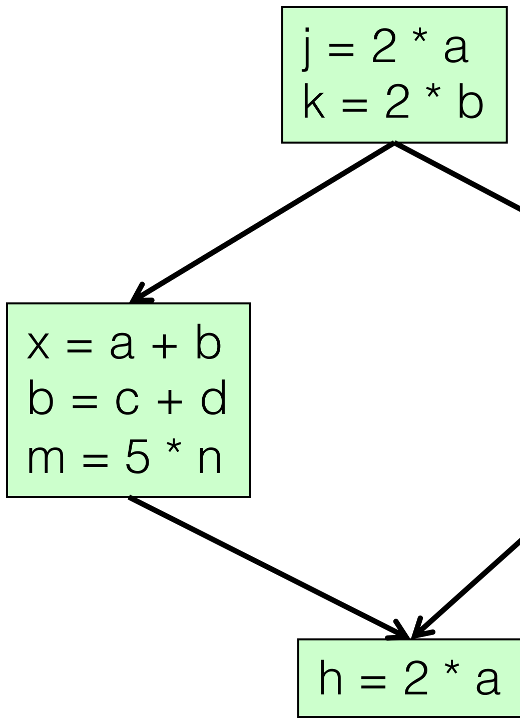
recompute AVAIL(b)

if AVAIL(b) changed

Worklist = Worklist ∪ successors(b)

Example: Find Available Expressions

$$AVAIL(b) = \cap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$



DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

DEF = { 5*n }
NKILL = exprs w/o c

DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

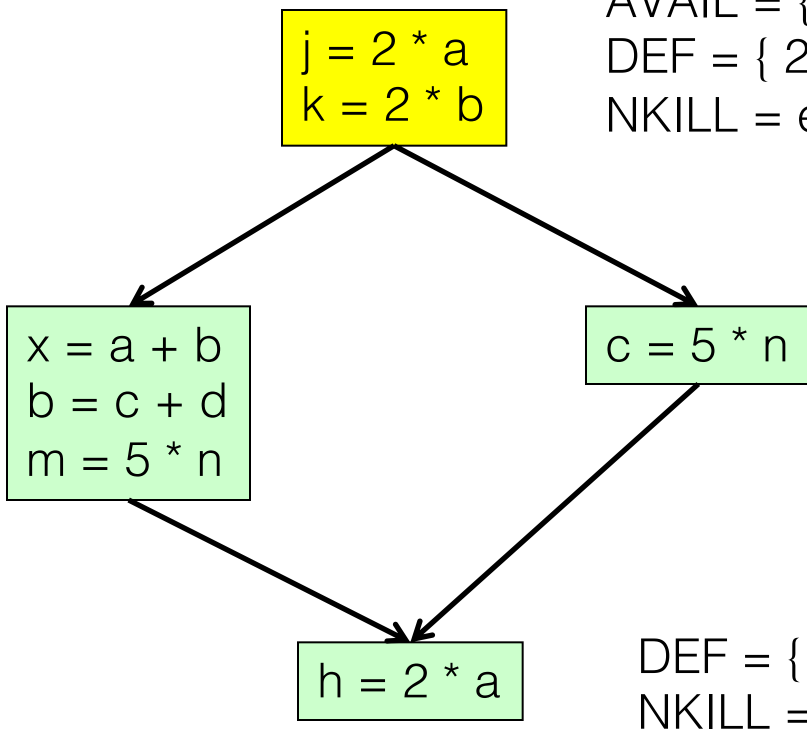
= processing

Example: Find Available Expressions

$$AVAIL(b) = \cap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b



DEF = { 5*n }
NKILL = exprs w/o c

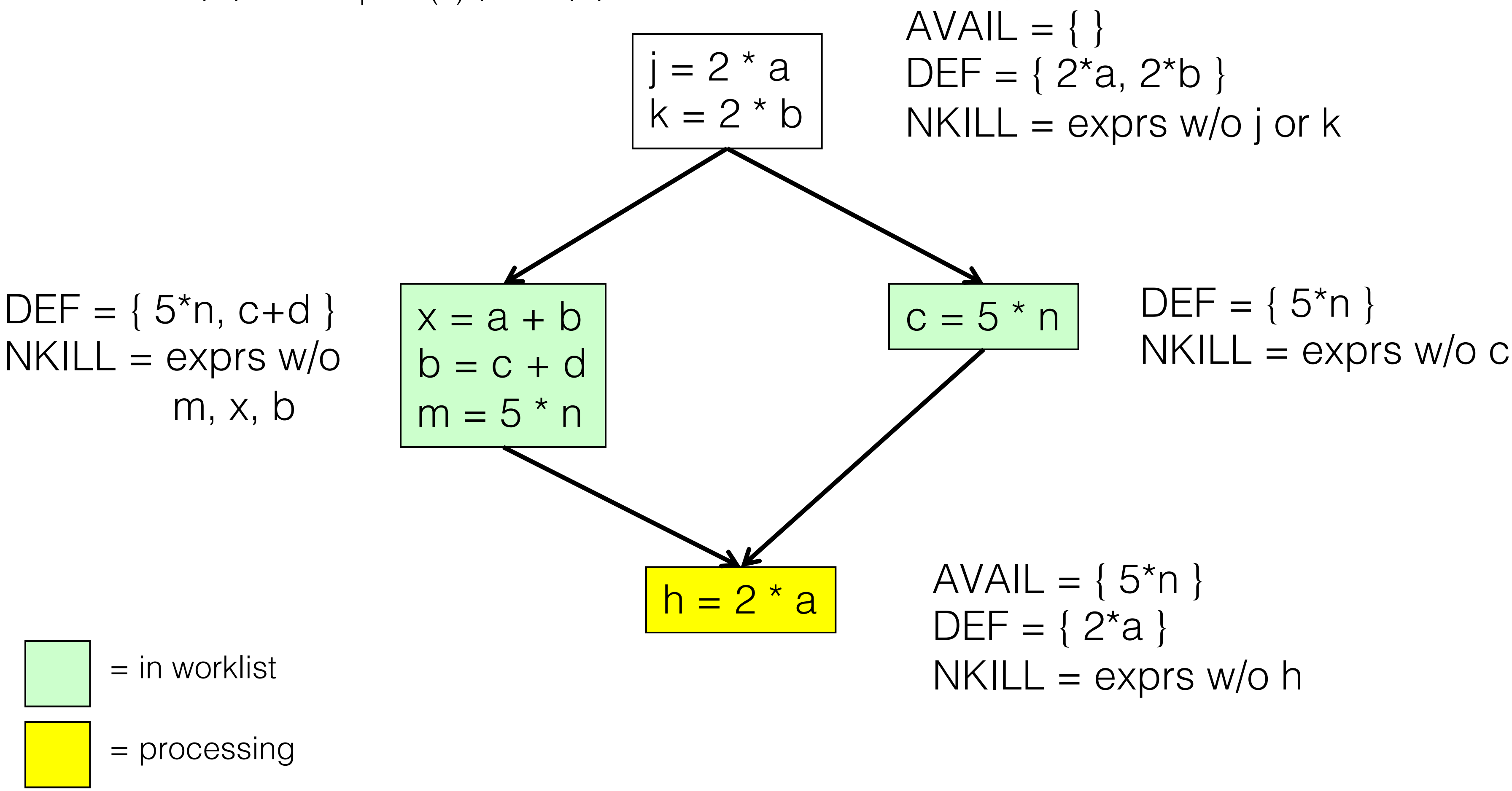
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Example: Find Available Expressions

$$\text{AVAIL}(b) = \cap_{x \in preds(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$$

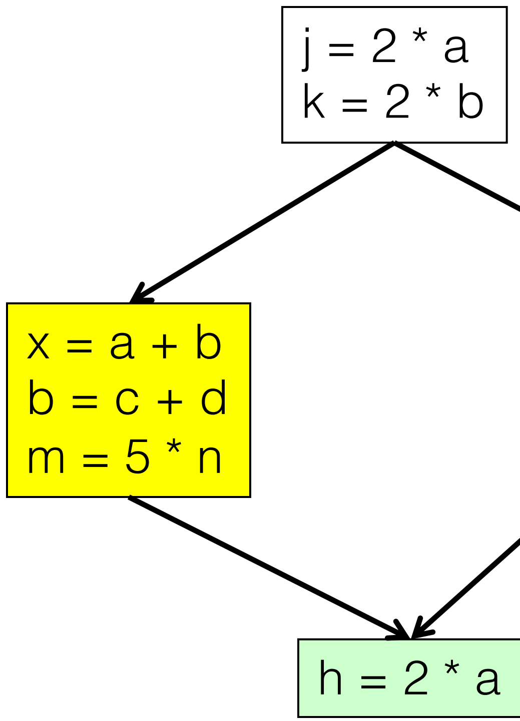


Example: Find Available Expressions

$$AVAIL(b) = \cap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b



DEF = { 5*n }
NKILL = exprs w/o c

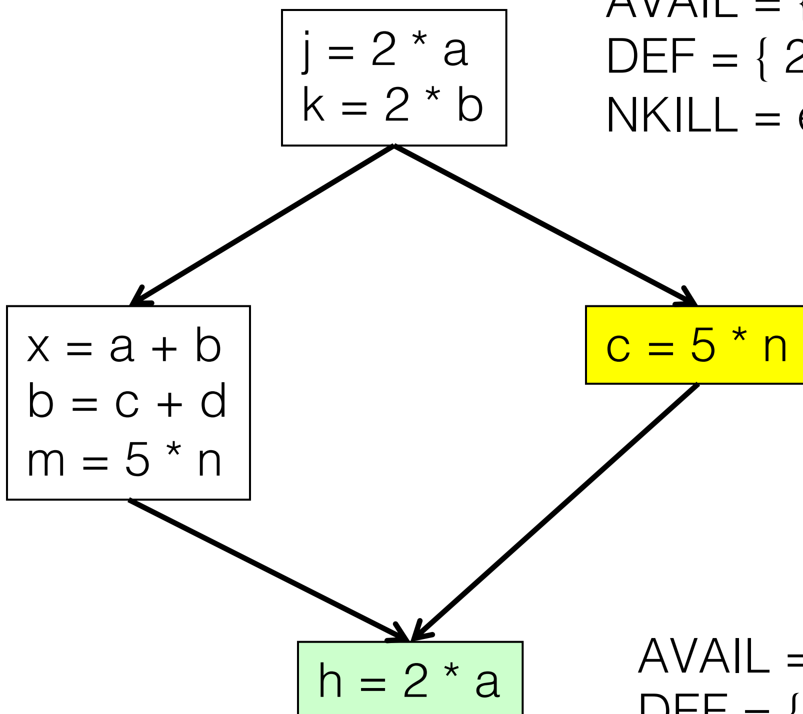
AVAIL = { 5*n }
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Example: Find Available Expressions

$$AVAIL(b) = \bigcap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$



AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

AVAIL = { 5*n }
DEF = { 2*a }
NKILL = exprs w/o h

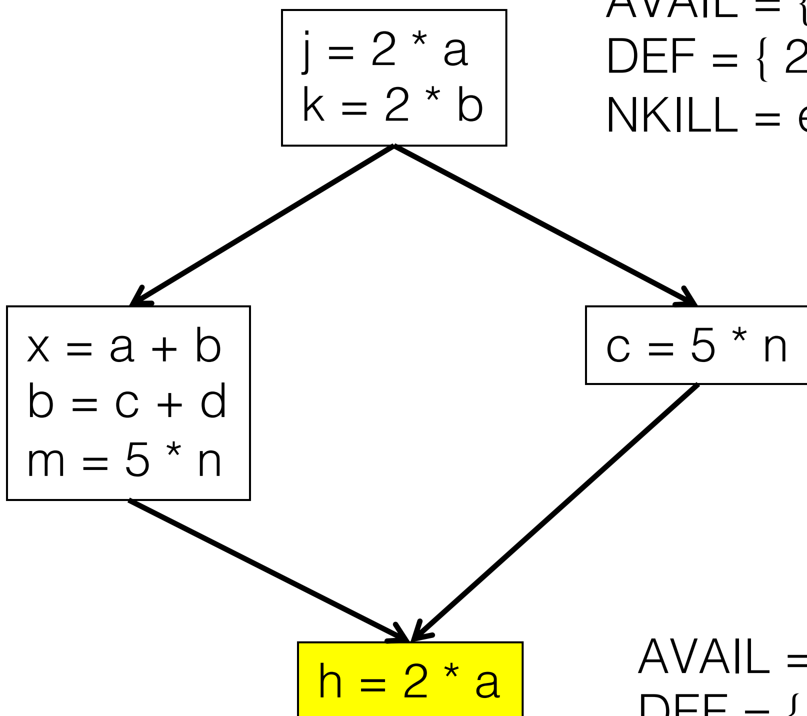
= in worklist

= processing

Example: Find Available Expressions

$$AVAIL(b) = \cap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k



AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

AVAIL = { 5*n, 2*a }
DEF = { 2*a }
NKILL = exprs w/o h

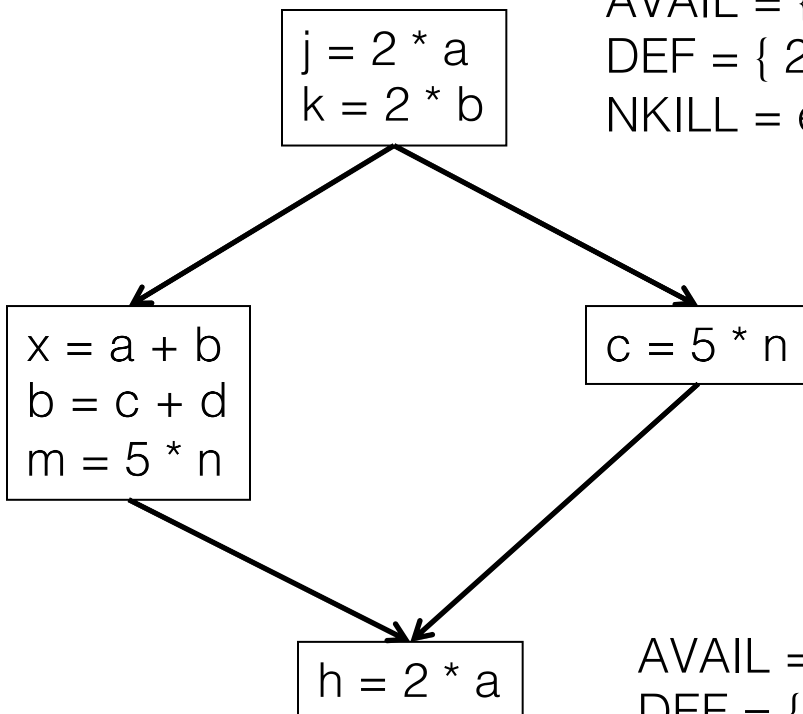
= in worklist

= processing

Example: Find Available Expressions

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k



AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

AVAIL = { 5*n, 2*a }
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

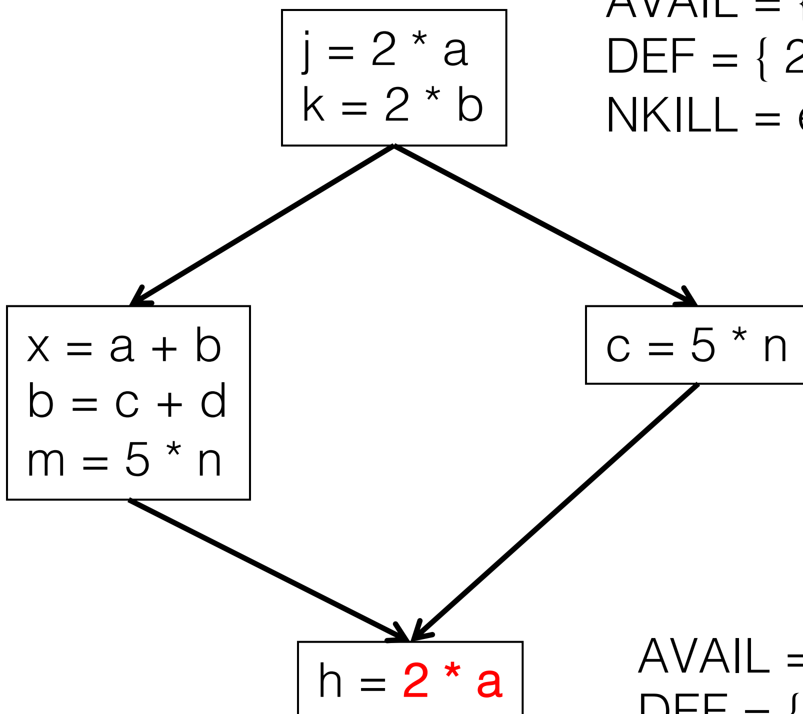
= processing

And the common subexpression is???

Example: Find Available Expressions

$AVAIL(b) = \cap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k



AVAIL = { 2*a, 2*b }
DEF = { 5*n, c+d }
NKILL = exprs w/o m, x, b

AVAIL = { 2*a, 2*b }
DEF = { 5*n }
NKILL = exprs w/o c

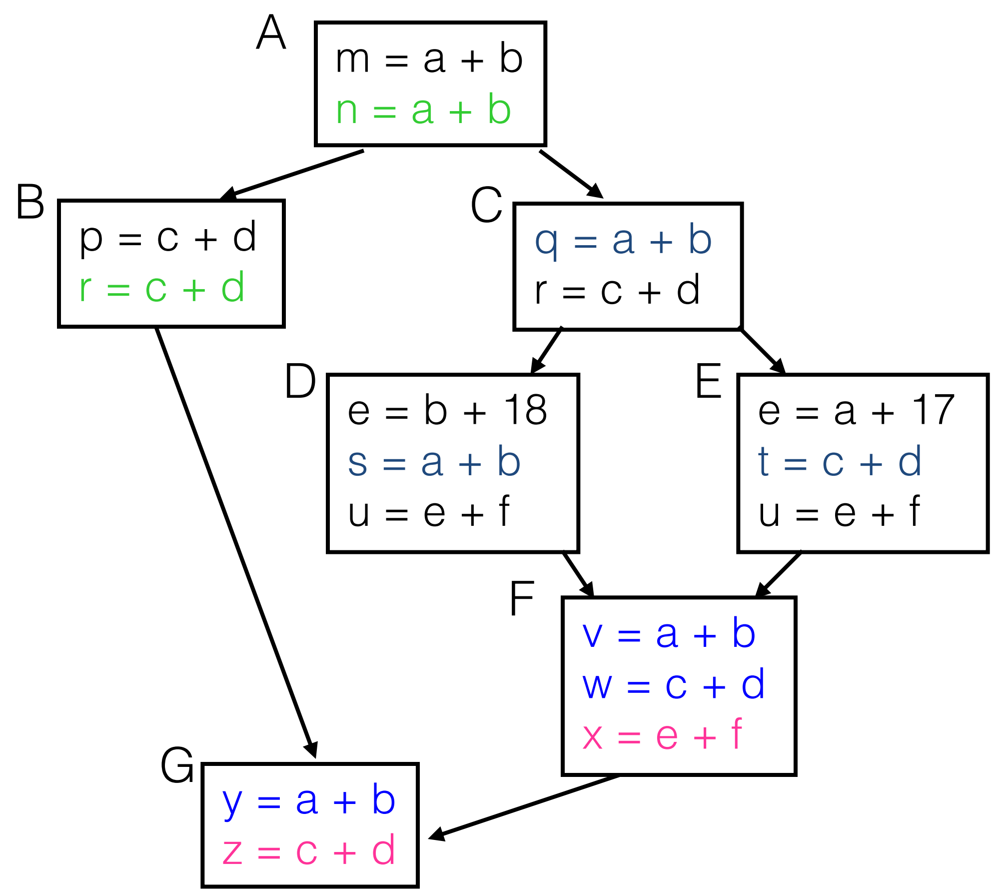
AVAIL = { 5*n, 2*a }
DEF = { 2*a }
NKILL = exprs w/o h

= in worklist

= processing

Comparing Algorithms

- LVN – Local Value Numbering
- SVN – Superlocal Value Numbering
- DVN – DominatoT-based Value Numbering
- GRE – Global Redundancy Elimination



Comparing Algorithms (2)

- LVN => SVN => DVN form a strict hierarchy – later algorithms find a superset of previous information
- Global RE finds a somewhat different set
  - Discovers e+f in F (computed in both D and E)
  - Misses identical values if they have different names (e.g.,
    a+b and c+d when a=c and b=d)
    - Value Numbering catches this

Scope of Analysis

- Larger context (EBBs, regions, global, interprocedural) sometimes helps
  - More opportunities for optimizations
- But not always
  - Introduces uncertainties about flow of control
  - Usually only allows weaker analysis
  - Sometimes has unwanted side effects
    - Can create additional pressure on registers, for example

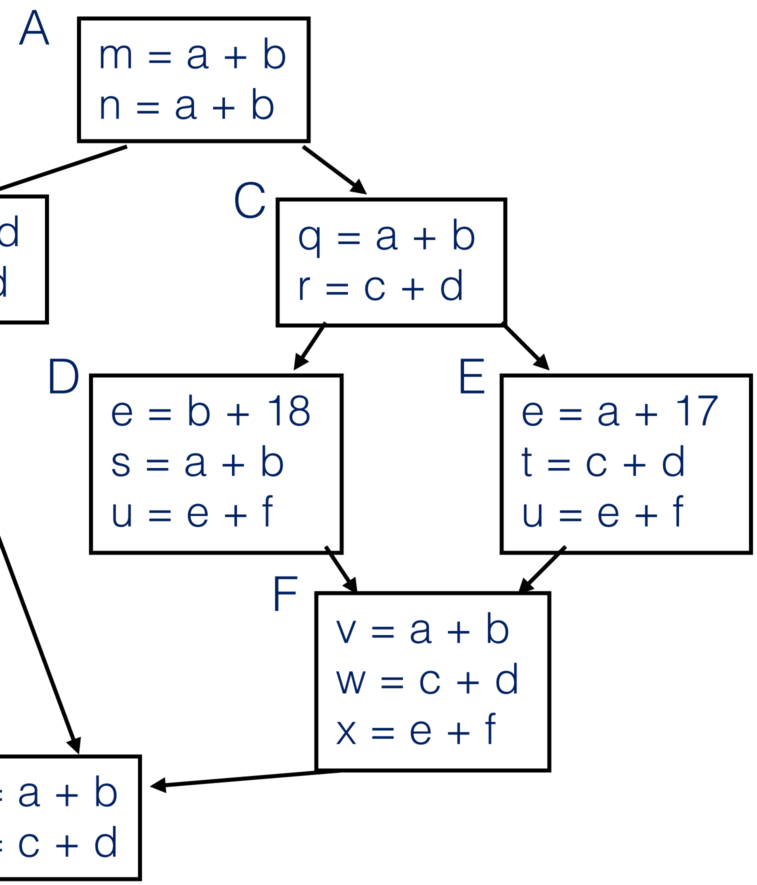
Code Replication

- Sometimes replicating code increases opportunities – modify the code to create larger regions with simple control flow
- Two examples
  - Cloning
  - Inline substitution

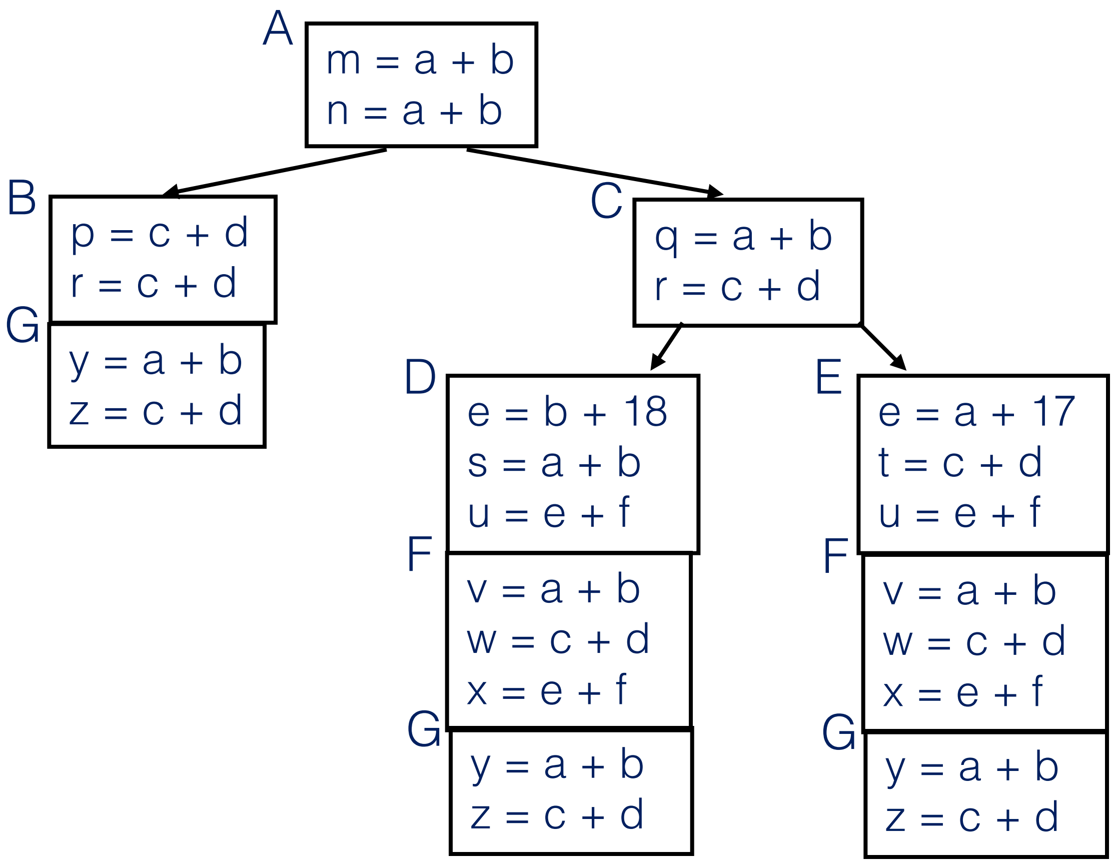
Cloning

- Idea: duplicate blocks with multiple predecessors
- Tradeoff
  - More local optimization possibilities – larger blocks, fewer branches
  - But: larger code size, may slow down if it interacts badly with cache

Original VN Example



Example with Cloning



Inline Substitution

- Problem: an optimizer has to treat a procedure call as if it (could have) modified all globally reachable data
  - Plus there is the basic expense of calling the procedure
- Inline Substitution: replace each call site with a copy of the called function body

Inline Substitution Issues

- Pro
  - More effective optimization – better local context and don't need to invalidate local assumptions
  - Eliminate overhead of normal function call
- Con
  - Potential code bloat
  - Need to manage recompilation when either caller or callee changes

Dataflow Analysis

- Available expressions are an example of a *dataflow analysis* problem
- Many similar problems can be expressed in a similar framework
- Only the first part of the story – once we've discovered facts, we then need to use them to improve code

Characterizing Dataflow Analysis

- All of these algorithms involve sets of facts about each basic block b

  IN(b) – facts true on entry to b

  OUT(b) – facts true on exit from b

  GEN(b) – facts created and not killed in b

  KILL(b) – facts killed in b
- These are related by the equation

  $$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

  – Solve this iteratively for all blocks

  – Sometimes information propagates forward; sometimes backward

Dataflow Analysis (1)

- A collection of techniques for compile-time reasoning about run-time values
- Almost always involves building a graph
  - Trivial for basic blocks
  - Control-flow graph or derivative for global problems
  - Call graph or derivative for whole-program problems

Dataflow Analysis (2)

- Usually formulated as a set of *simultaneous equations* (dataflow problem)
  - Sets attached to nodes and edges
  - Need a lattice (or semilattice) to describe values
    - In particular, has an appropriate operator to combine values and an appropriate "bottom" or minimal value

Dataflow Analysis (3)

- Desired solution is usually a *meet over all paths* (MOP) solution
  - "What is true on every path from entry"
  - "What can happen on any path from entry"
  - Usually relates to safety of optimization

Dataflow Analysis (4)

- Limitations
  - Precision – “up to symbolic execution”
    - Assumes all paths taken
  - Sometimes cannot afford to compute full solution
  - Arrays – classic analysis treats each array as a single fact
  - Pointers – difficult, expensive to analyze
    - Imprecision rapidly adds up
    - But gotta do it to effectively optimize things like C/C++
- For scalar values we can quickly solve simple problems

Example: Live Variable Analysis

- A variable *v* is *live* at point *p* iff there is *any* path from *p* to a use of *v* along which *v* is not redefined
- Some uses:
  - Register allocation – only live variables need a register
  - Eliminating useless stores – if variable not live at store, then stored variable will never be used
  - Detecting uses of uninitialized variables – if live at declaration (before initialization) then it might be used uninitialized
  - Improve SSA construction – only need Φ-function for variables that are live in a block (later)

Liveness Analysis Sets

- For each block b, define
  - use[*b*] = variable used in *b* before any def
  - def[*b*] = variable defined in *b* & not killed
  - in[*b*] = variables live on entry to *b*
  - out[*b*] = variables live on exit from *b*

Equations for Live Variables

- Given the preceding definitions, we have

  $in[b] = use[b] \cup (out[b] - def[b])$

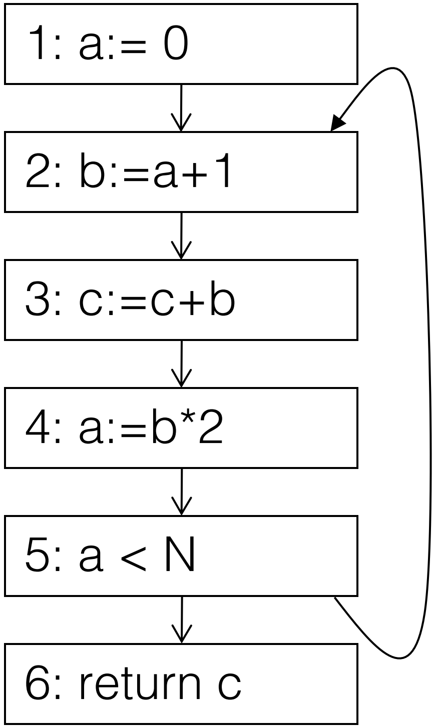
  $out[b] = \cup_{s \in succ[b]} in[s]$

- Algorithm
  - Set $in[b] = out[b] = \emptyset$
  - Update in, out until no change

Example (1 stmt per block)

- Code

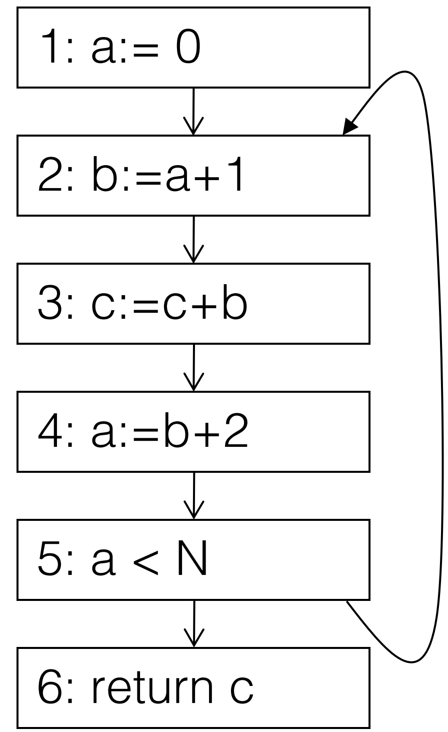
```
      a := 0
L:    b := a+1
      c := c+b
      a := b*2
      if a < N goto L
      return c
```



$$in[b] = use[b] \cup (out[b] - def[b])$$
$$out[b] = \cup_{s \in succ[b]} in[s]$$

Calculation

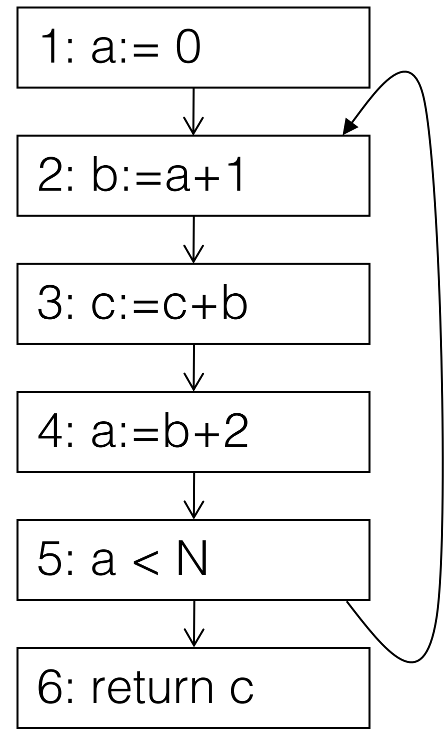
			I		II		III	
block	use	def	out	in	out	in	out	in
6								
5								
4								
3								
2								
1								



$$in[b] = use[b] \cup (out[b] - def[b])$$
$$out[b] = \cup_{s \in succ[b]} in[s]$$

Calculation

			I		II		III	
block	use	def	out	in	out	in	out	in
6	c	--	--	c	--	c		
5	a	--	c	a,c	a,c	a,c		
4	b	a	a,c	b,c	a,c	b,c		
3	b,c	c	b,c	b,c	b,c	b,c		
2	a	b	b,c	a,c	b,c	a,c		
1	--	a	a,c	c	a,c	c		



$$in[b] = use[b] \cup (out[b] - def[b])$$
$$out[b] = \cup_{s \in succ[b]} in[s]$$

Equations for Live Variables v2

- Many problems have more than one formulation. For example, Live Variables…
- Sets
  - USED(b) – variables used in b before being defined in b
  - NOTDEF(b) – variables not defined in b
  - LIVE(b) – variables live on *exit* from b
- Equation

$$LIVE(b) = \cup_{s \in succ(b)} USED(s) \cup (LIVE(s) \cap NOTDEF(s))$$

Efficiency of Dataflow Analysis

- The algorithms eventually terminate, but the expected time needed can be reduced by picking a good order to visit nodes in the CFG
  - Forward problems – reverse postorder
  - Backward problems – postorder

Example: Reaching Definitions

- A definition *d* of some variable *v* *reaches* operation *i* iff *i* reads the value of *v* and there is a path from *d* to *i* that does not define *v*
- Uses
  - Find all of the possible definition points for a variable in an expression

Equations for Reaching Definitions

- Sets
  - DEFOUT(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
  - SURVIVED(b) – set of all definitions not obscured by a definition in b
  - REACHES(b) – set of definitions that reach b
- Equation

$$\text{REACHES}(b) = \cup_{p \in preds(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))$$

Example: Very Busy Expressions

- An expression *e* is considered *very busy* at some point *p* if *e* is evaluated and used along every path that leaves *p*, and evaluating *e* at *p* would produce the same result as evaluating it at the original locations
- Uses
  - Code hoisting – move *e* to *p* (reduces code size; no effect on execution time)

Equations for Very Busy Expressions

- Sets
  - USED(b) – expressions used in b before they are killed
  - KILLED(b) – expressions redefined in b before they are used
  - VERYBUSY(b) – expressions very busy on exit from b
- Equation

$$\text{VERYBUSY}(b) = \bigcap_{s \in succ(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s))$$

Using Dataflow Information

- A few examples of possible transformations…

Classic Common-Subexpression Elimination (CSE)

- In a statement s: t := x op y, if x op y is *available* at s then it need not be recomputed
- Analysis: compute *reaching expressions* i.e., statements n: v := x op y such that the path from n to s does not compute x op y or define x or y

Classic CSE Transformation

- If x op y is defined at n and reaches s
  - Create new temporary w
  - Rewrite n: v := x op y as

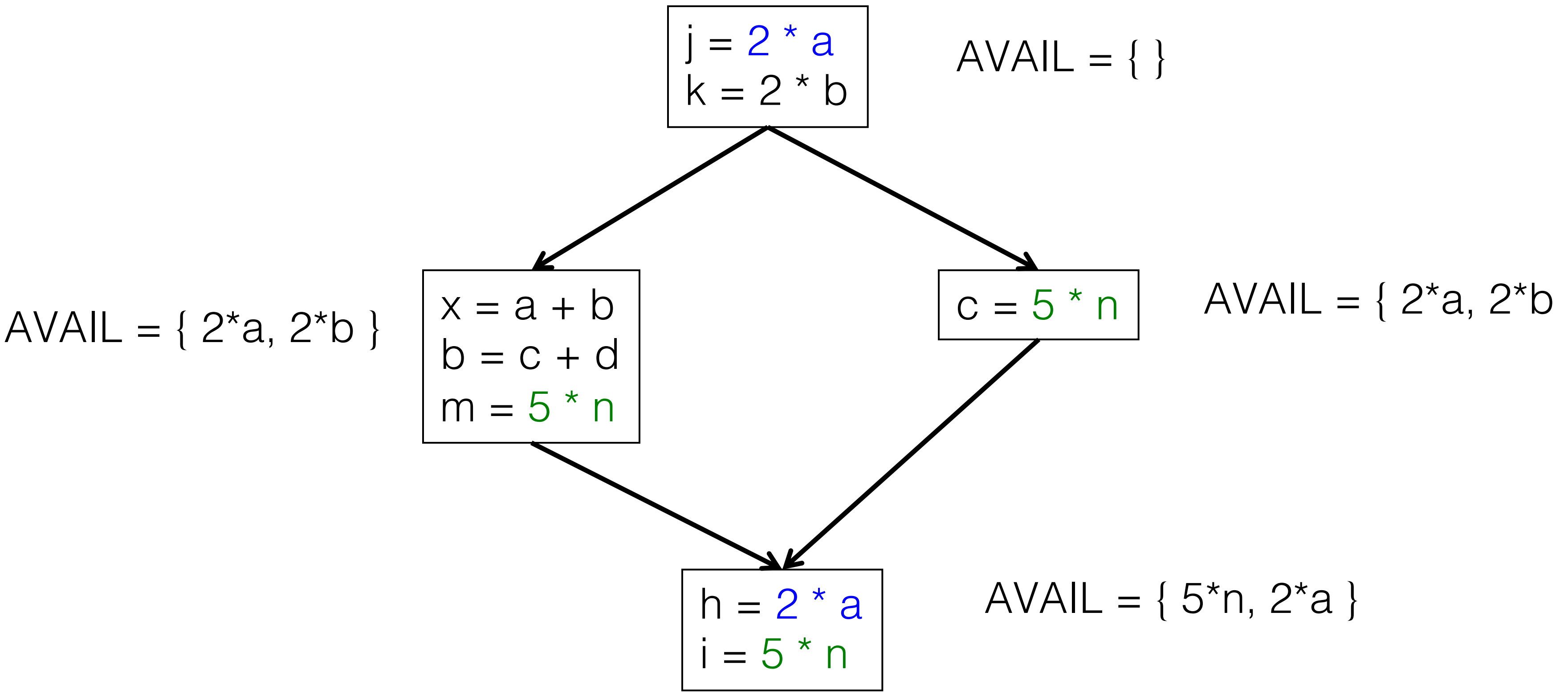
    n: w := x op y

    n': v := w
  - Modify statement s to be

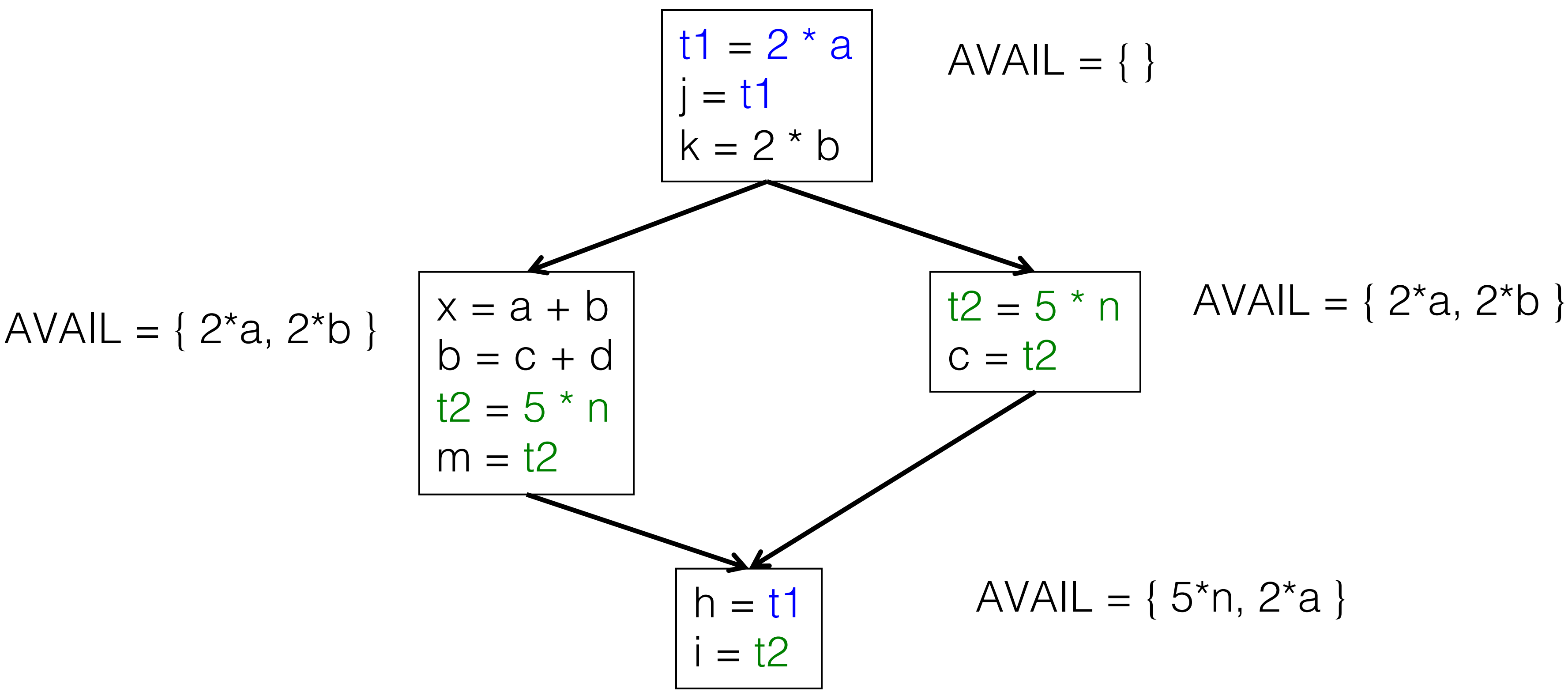
    s: t := w

  - (Rely on copy propagation to remove extra assignments that are not really needed)

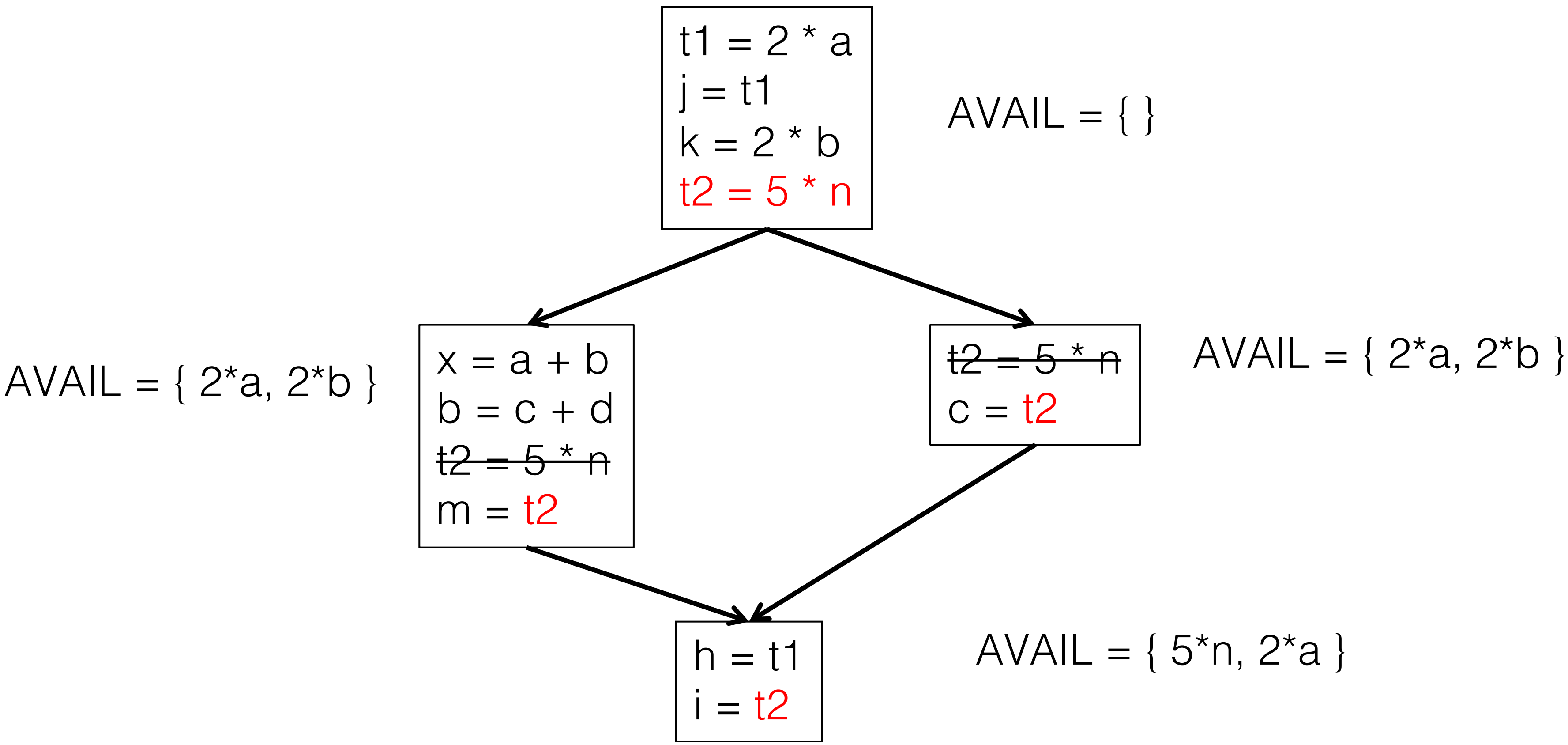
Revisiting Example (w/slight addition)



Revisiting Example (w/slight addition)



Then Apply Very Busy…



Constant Propagation

- Suppose we have
  - Statement d: t := c, where c is constant
  - Statement n that uses t
- If d reaches n and no other definitions of t reach n, then rewrite n to use c instead of t

Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: t := z
  - Statement n uses t
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
  - Recall that this can help remove dead assignments

Copy Propagation Tradeoffs

- Downside is that this can increase the lifetime of variable z and increase need for registers or memory traffic
- But it can expose other optimizations, e.g.,

```
a := y + z
u := y
c := u + z        // copy propagation makes this y + z
```

  – After copy propagation we can recognize the common subexpression

Dead Code Elimination

- If we have an instruction

  s: a := b op c

  and a is not live-out after s, then s can be eliminated

  – Provided it has no implicit side effects that are visible (output, exceptions, etc.)

    - If b or c are function calls, they have to be assumed to have unknown side effects unless the compiler can prove otherwise

Aliases

- A variable or memory location may have multiple names or *aliases*
  - Call-by-reference parameters
  - Variables whose address is taken (&x)
  - Expressions that dereference pointers (p.x, *p)
  - Expressions involving subscripts (a[i])
  - Variables in nested scopesa

Aliases vs Optimizations

- Example:

  p.x := 5; q.x := 7; a := p.x;

  - Does reaching definition analysis show that the definition of p.x reaches a?
  - (Or: do p and q refer to the same variable/object?)
  - (Or: *can* p and q refer to the same thing?)

Aliases vs Optimizations

- Example

```
void f(int *p, int *q) {
  *p = 1; *q = 2;
  return *p;
}
```

  – How do we account for the possibility that p and q might refer to the same thing?
  – Safe approximation: since it's possible, assume it is true (but rules out a lot)
    - C programmers can use "restrict" to indicate no other pointer is an alias for this one

Types and Aliases (1)

- In Java, ML, MiniJava, and others, if two variables have incompatible types they cannot be names for the same location
  - Also helps that programmer cannot create arbitrary pointers to storage in these languages

Types and Aliases (2)

- Strategy: Divide memory locations into *alias classes* based on type information (every type, array, record field is a class)
- Implication: need to propagate type information from the semantics pass to optimizer
  - Not normally true of a minimally typed IR
- Items in different alias classes cannot refer to each other

Aliases and Flow Analysis

- Idea: Base alias classes on points where a value is created
  - Every new/malloc and each local or global variable whose address is taken is an alias class
  - Pointers can refer to values in multiple alias classes (so each memory reference is to a set of alias classes)
  - Use to calculate “may alias” information (e.g., p “may alias” q at program point s)

Using “may-alias” information

- Treat each alias class as a “variable” in dataflow analysis problems
- Example: framework for available expressions
  - Given statement s: M[a]:=b,

    gen[s] = { }

    kill[s] = { M[x] | a may alias x at s }

May-Alias Analysis

- Without alias analysis, #2 kills M[t] since x and t might be related
- If analysis determines that "x may-alias t" is false, M[t] is still available at #3; can eliminate the common subexpression and use copy propagation

- Code

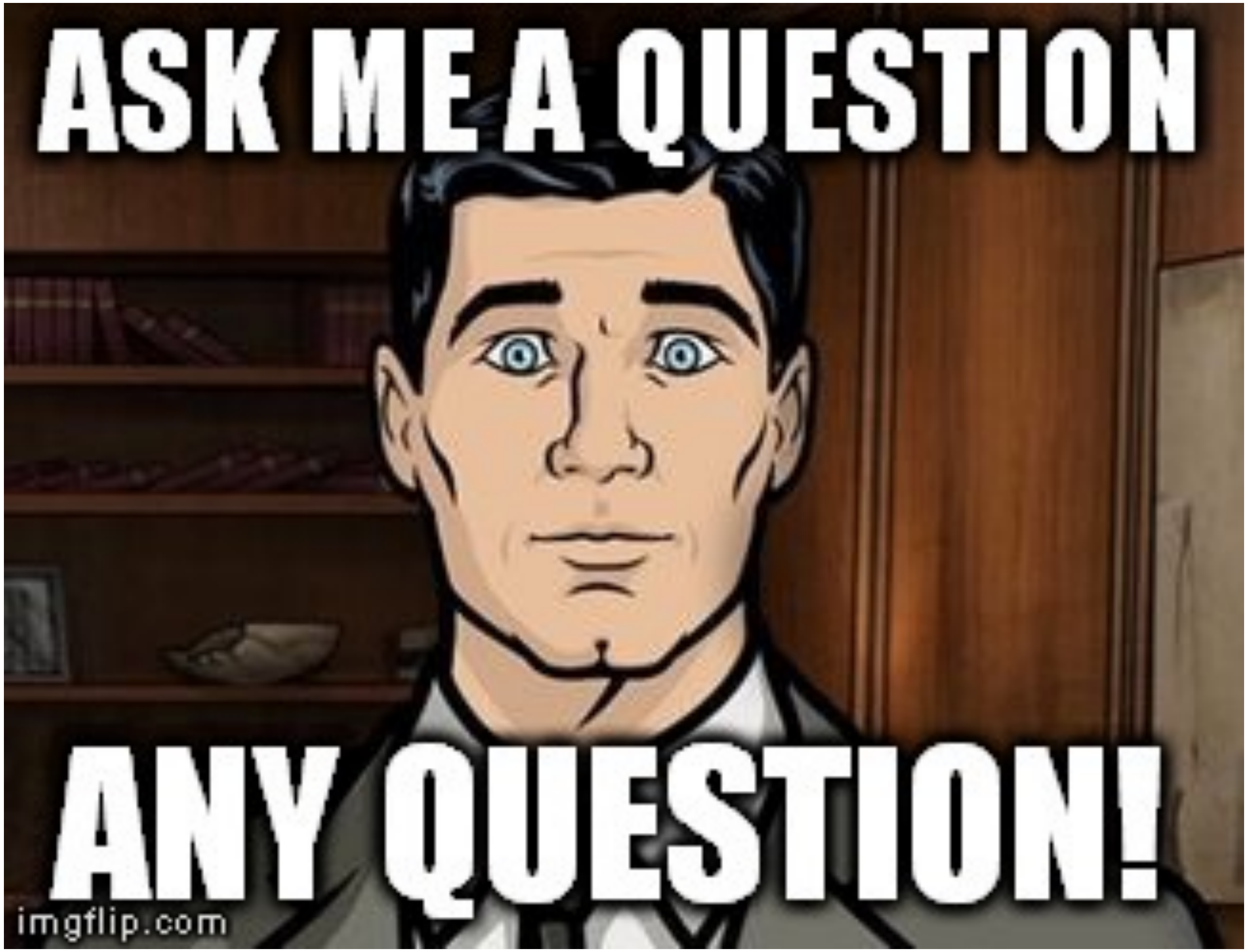
```
1:  u := M[t]
2:  M[x] := r
3:  w := M[t]
4:  b := u+w
```

Coming Attractions

- Dataflow analysis is the core of classical optimizations
  - Although not the only possible story
- Still to explore:
  - Discovering and optimizing loops
  - SSA – Static Single Assignment form

SSA Name Space

- Two Principles
  - Each name is defined by exactly one operation
  - Each operand refers to exactly one definition
- Need to deal with merge points
  - Add Φ functions at merge points to reconcile names
  - Use subscripts on variable names for uniqueness



[Meme credit: imgflip.com]