Lecture 26

Classical Optimizations

이재진

서울대학교 컴퓨터공학부

http://aces.snu.ac.kr

Optimization

- Hardly ever possible to guarantee optimality
- Not even always an improvement
- Optimizing for the average case
- Code optimization is an important phase in production compilers
- Get your code right first, then optimize it

Rules for Optimization

- In general, 80% percent of a program's execution time is spent executing 20% of the code
  - 90%/10% for performance-hungry programs
- Spend your time optimizing the important 10-20% of your program
- Optimize the common case even at the cost of making the uncommon case slower
- The best and most important way of optimizing a program is using good algorithms

Code Optimization

- Peephole optimization
  - Performed over a very small set of instructions
- Local code optimization
  - Code improvement with in a basic block
- Global code optimization
  - Improvements take into account what happens across basic blocks
  - Intra-procedural
- Program level
  - Inter-procedural
- An optimization must preserve the semantics of the original program

Constant Folding

- Evaluate expressions at compile-time
- Floating point/integer precision
  - Must agree with run-time precision
  - E.g., 1.0/3.0 != 0.3333
- Handling arithmetic exceptions
  - E.g., division by 0

```
    const float pi =
    3.1416;
    float v, r;
...
v = 4/3 * pi * r * r * r;
```

```
    const float pi =
    3.1416;
    float v, r;
...
v = 4.1888 * r * r * r;
```

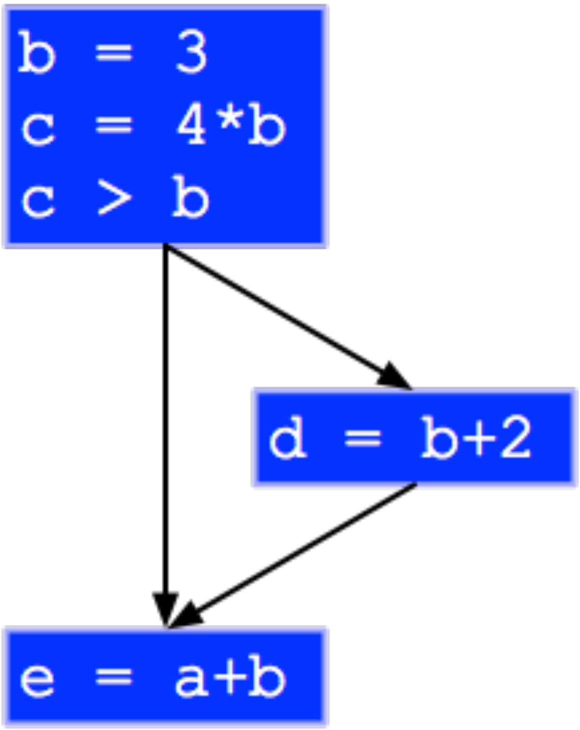
Constant Folding and Propagation

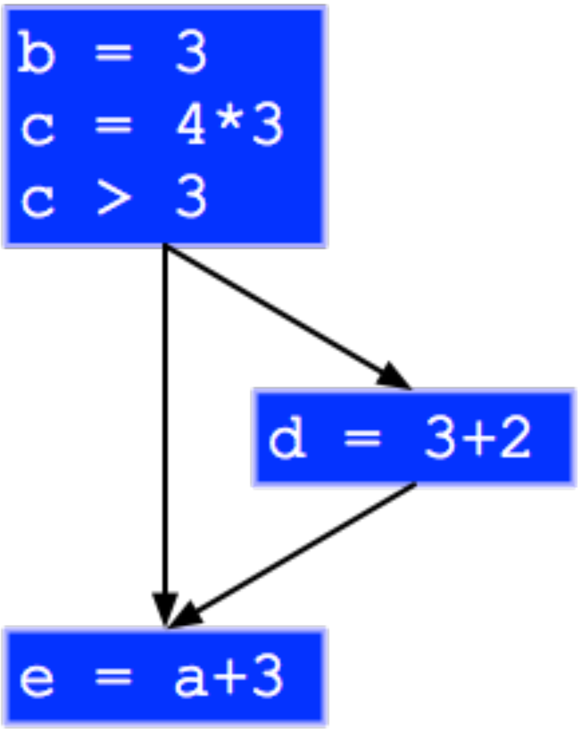
- Constant folding together with constant propagation
- Constant propagation
  - Given an assignment x = c for a variable x and a constant c, replace later uses of x with uses of c as long as intervening assignments have not changed the value of x
- Important for RISC architecture because all RISC architectures provide instructions that take a small integer constant as an operand (more efficient code)
- Saves registers

```
x = 24;
x = x + 24;
```

```
x = 64;
```

Constant Folding and Propagation (cont'd)





Constant Folding and Propagation (cont'd)

- In general, data flow analysis is required
- z is unknown below

```
x = 24;
y = x + 1;
while (x < z) {
    x = x + 2;
}
```

```
x = 24;
y = 25;
while (x < z) {
    x = x + 2;
}
```

Algebraic Simplifications

- More general form of constant folding
- Algebraic properties of operators or particular operator-operand combinations are exploited to simplify expressions

- For example,
  - $x + 0 \Rightarrow x$, $x - 0 \Rightarrow x$
  - $x * 1 \Rightarrow x$, $x / 1 \Rightarrow x$
  - $x * 0 \Rightarrow 0$
  - $0 - x \Rightarrow - x$
  - $x2 = x * x$
  - $2 * x = x + x$
  - …

Strength Reduction

- Replace expensive operations with less expensive operations

- For example,
  - x = x * 8 ⇒ x = x << 3

Algebraic Reassociation

- Using some specific algebraic properties (associativity, commutativity, and distributivity) to divide an expression into parts that are constant, loop-invariant, and variable

- For example,
  - $(i - j) + (i - j) + (i - j) + (i - j) = 4 * i – 4 * j$
    - What if $i = 230 = \text{0x40000000}$ and $j = 230 – 1 = \text{0x3fffffff}$

Unreachable Code Elimination

- Unreachable code is the code that can never be executed because there exists no control flow path to the code from the rest of the program

```
int foo( int x, int y )
{
    int z;

    return x * y;

    z = x * y;
}
```

Dead Code Elimination

- Removes dead code
- A variable is dead if it is not used on any path from the location in the code where it is defined to the exit point of the routine
  - Eliminates assignments to dead variables
  - Dead store elimination
- Dead code
  - Code whose results are never used in any useful computation
    - Results do not affect on the result of the program
- Useful computation
  - Output statements, input statements, control-flow statements, and their required statements
- Cleaner code

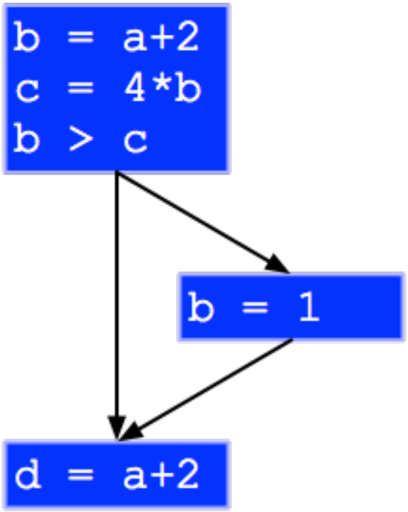
```
x = y + 4
y = 6
z = 2 * z
```

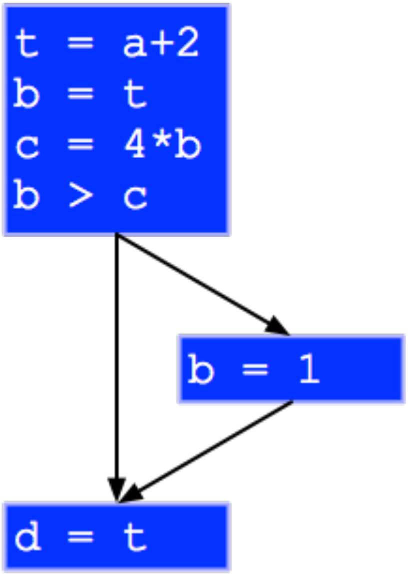
```

y = 6
z = 2 * z
```

Common Subexpression Elimination

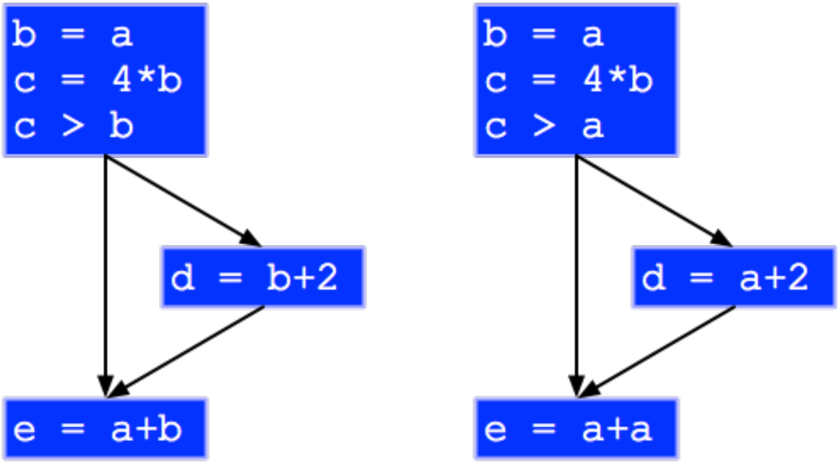
- Avoid evaluating the same expression more than once
- An occurrence of an expression in a program is a common subexpression if there is another occurrence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations
- Requires data-flow analysis





Copy Propagation

- Given an assignment x = y for some variables x and y, replaces later uses of x with uses of y, as long as intervening instructions have not changed the value of either x or y
- Requires data-flow analysis
- Enables other transformations



```
b = a
c = 4*b
c > b
```

```
d = b+2
```

```
e = a+b
```

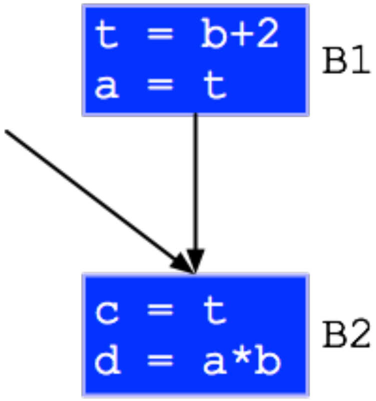
```
b = a
c = 4*b
c > a
```

```
d = a+2
```

```
e = a+a
```

Forward Expression Substitution

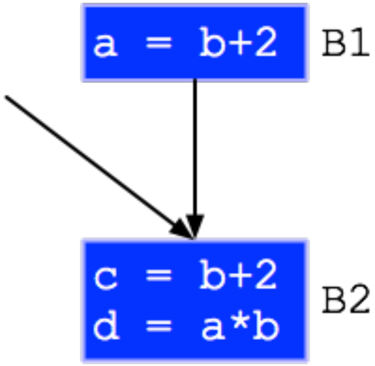
- Instead of replacing an expression evaluation by a copy operation, it replaces a copy by reevaluation of the expression
- The inverse of common-subexpression elimination
- Reduces register pressure



```
t = b+2
a = t
```

```
c = t
d = a*b
```

3 registers (B1→B2)



```
a = b+2
```

```
c = b+2
d = a*b
```

2 registers (B1→B2)

Loop-invariant Code

- Computations in a loop that produce the same value on every iteration of the loop
- An instruction is loop invariant if, for each of its operands,
  - The operand is constant,
  - All definitions that reach this use of the operand are located outside the loop, or
  - There is exactly one definition of the operand that reaches the instruction and that definition is an instruction inside the loop that is itself loop-invariant

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

Computing Loop Invariants

1. Mark as loop invariant those instructions whose operands are all either constant or have all their reaching definitions outside the loop
2. Mark as loop invariant all those instructions not previously so marked all of whose operands either are constant, have all their reaching definitions outside the loop, or have exactly one reaching definition, and that definition is an instruction in the loop marked invariant
3. Repeat step 2 above until no instruction is newly marked as invariant in an iteration

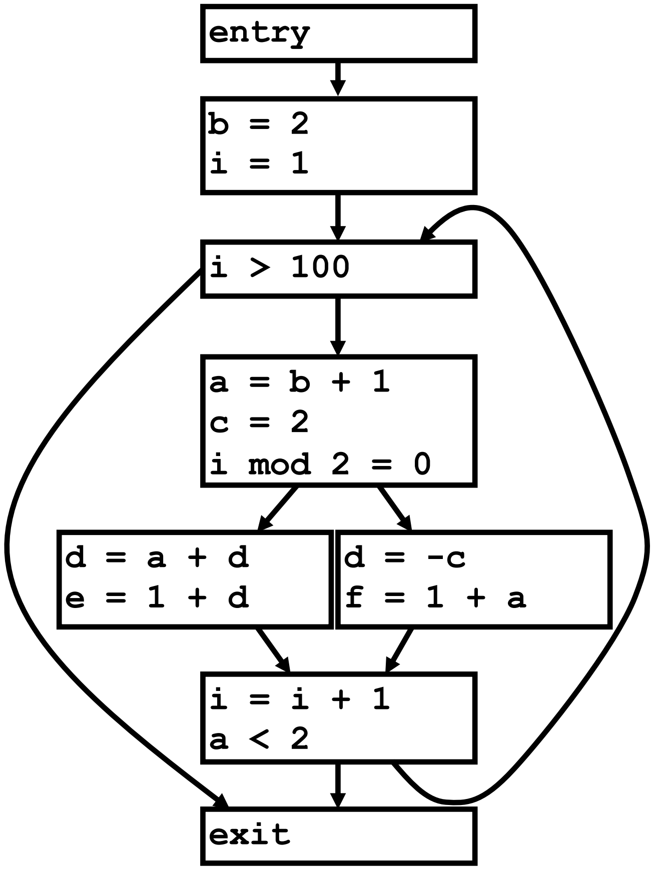
Loop-invariant Code Motion

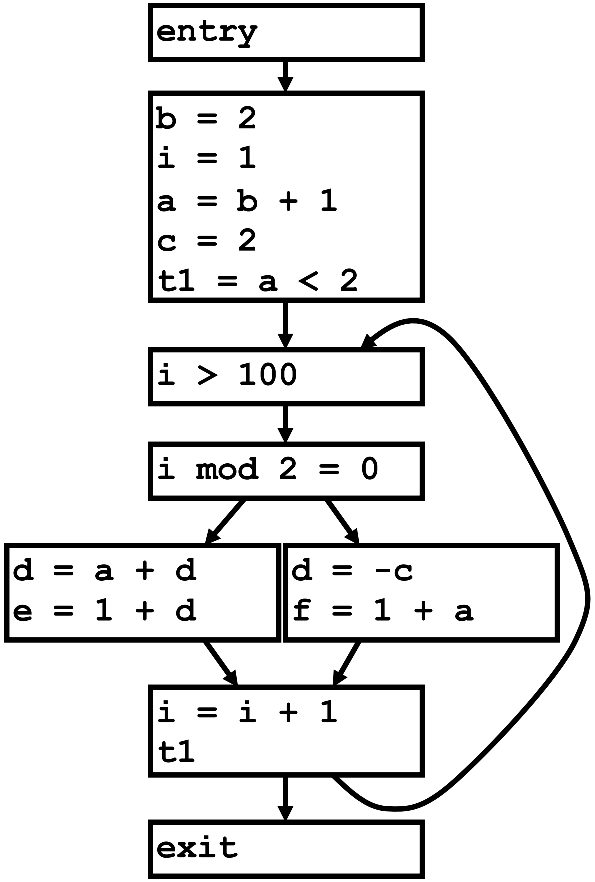
- Moves the loop-invariant code out of the loop

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

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

Loop-invariant Code Motion (contd.)





Function Inlining

- Also called procedure integration
- Replaces a function call with the body of the function
  - By instantiating the body with the actual parameters and copying the result to the call site
- Inlined functions should be small, or size of code might blow up
- Be careful with recursion
- Avoids overhead of function/procedure call

Function Inlining (cont’d)

```
int foo (int x)
{
    return (x + 22) * 3;
}
...
a = foo(y + 3);
b = foo(z * 5);
...
```

```
...
a = ((y + 3) + 22) * 3;
b = ((z * 5) + 22) * 3;
...
```

Function Cloning

- Create specialized code for a function for different calling parameters
  - Specialize functions such that certain optimizations are feasible

```
int foo (int x, int y, int z) {
    if (z < 3) return x * y;
    return x + y;
}
...
foo(a, b, 1);
...
foo(a, b, k);
...
foo(a, b, 1);
...
```

```
int foo (int x, int y, int z) {
    if (z < 3) return x * y;
    return x + y;
}

int foo_clone (int x, int y) {
    return x * y;
}
...
foo_clone(a, b);
...
foo(a, b, k);
...
foo_clone(a, b);
...
```

Tail Recursion Elimination

- Tail recursion
  - No computation follows recursive call
- Efficiently implemented
  - No stack is needed
  - Most of the frame of the current procedure is not needed any more, and it can be replaced by the frame of the tail call

```
int gcd (int a, int b) {
   if (a == b) return a;
   else if (a > b) return gcd (a - b, b);
   else return gcd (a, b – a);
}
```