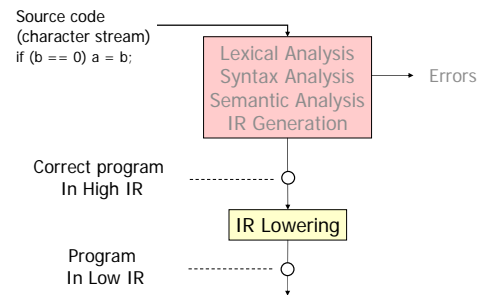


CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 23: Introduction to Optimizations
16 March 07

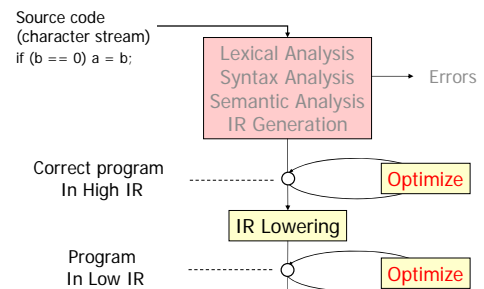
Where We Are



What Next?

- At this point we could generate assembly code from the low-level IR
- Better:
 - Optimize the program first
 - Then generate code
- If optimization performed at the IR level, then they apply to all target machines

Optimizations



What are Optimizations?

- **Optimizations** = code transformations that *improve* the program
- **Different kinds**
 - space optimizations: improve (reduce) memory use
 - time optimizations: improve (reduce) execution time
- Code transformations must be **safe!**
 - They must preserve the meaning of the program

Why Optimize?

- Programmers don't always write optimal code – can recognize ways to improve code (e.g., avoid recomputing same expression)
- High-level language may make some optimizations inconvenient or impossible to express
$$a[i][j] = a[i][j] + 1;$$
- High-level unoptimized code may be more readable: cleaner, modular
$$\text{int square}(x) \{ \text{return } x*x; \}$$

Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade off space versus time
- Example: loop unrolling
 - Increases code space, speeds up one loop
 - Frequently executed code with long loops: space/time tradeoff is generally a win
 - Infrequently executed code: may want to optimize code space at expense of time
- Want to optimize program hot spots

Many Possible Optimizations

- Many ways to optimize a program
- Some of the most common optimizations:
 - Function Inlining
 - Function Cloning
 - Constant folding
 - Constant propagation
 - Dead code elimination
 - Loop-invariant code motion
 - Common sub-expression elimination
 - Strength reduction
 - Branch prediction/optimization
 - Loop unrolling

Constant Propagation

- If value of variable is known to be a constant, replace use of variable with constant
- Example:

```
n = 10
c = 2
for (i=0; i<n; i++) { s = s + i*c; }
```
- Replace n, c:

```
for (i=0; i<10; i++) { s = s + i*2; }
```
- Each variable must be replaced only when it has known constant value:
 - Forward from a constant assignment
 - Until next assignment of the variable

Constant Folding

- Evaluate an expression if operands are known at compile time (i.e., they are constants)
 - Example:

```
x = 1.1 * 2;    =>    x = 2.2;
```
 - Performed at every stage of compilation
 - Constants created by translations or optimizations
- ```
int x = a[2] => t1 = 2*4
 t2 = a + t1
 x = *t2
```

## Algebraic Simplification

- More general form of constant folding: take advantage of usual simplification rules

```
a * 1 => a a * 0 => 0
a / 1 => a a + 0 => a
b || false => b b && true => b
```
- Repeatedly apply the above rules

```
(y*1+0)/1 => y*1+0 => y*1 => y
```
- Must be careful with floating point!

## Copy Propagation

- After assignment  $x = y$ , replace uses of  $x$  with  $y$
- Replace until  $x$  is assigned again

```
x = y;
if (x > 1)
 s = x * f(x - 1);
```

```
x = y;
if (y > 1)
 s = y * f(y - 1);
```
- What if there was an assignment  $y = z$  before?
  - Transitivity apply replacements

## Common Subexpression Elimination

- If program computes same expression multiple time, can reuse the computed value
- Example:  

```
a = b+c;
c = b+c;
d = b+c;
⇒
a = b+c;
c = a;
d = b+c;
```
- Common subexpressions also occur in low-level code in address calculations for array accesses:  

```
a[i] = b[i] + 1;
```

CS 412/413 Spring 2007

Introduction to Compilers

13

## Unreachable Code Elimination

- Eliminate code that is never executed
- Example:  

```
#define debug false
s = 1;
if (debug)
 print("state = ", s);
⇒
s = 1;
```
- Unreachable code may not be obvious in low IR (or in high-level languages with unstructured "goto" statements)

CS 412/413 Spring 2007

Introduction to Compilers

14

## Unreachable Code Elimination

- Unreachable code in while/if statements when:
  - Loop condition is always false (loop never executed)
  - Condition of an if statement is always true or always false (only one branch executed)

```
if (false) S ⇒ ;
if (true) S else S' ⇒ S
if (false) S else S' ⇒ S'
while (false) S ⇒ ;
while (2>3) S ⇒ ;
```

CS 412/413 Spring 2007

Introduction to Compilers

15

## Dead Code Elimination

- If effect of a statement is never observed, eliminate the statement

```
x = y+1;
y = 1;
x = 2*z;
⇒
y = 1;
x = 2*z;
```

- Variable is *dead* if value is never used after definition
- Eliminate assignments to dead variables
- Other optimizations may create dead code

CS 412/413 Spring 2007

Introduction to Compilers

16

## Loop Optimizations

- Program hot spots are usually loops (exceptions: OS kernels, compilers)
- Most execution time in most programs is spent in loops: 90/10 is typical
- Loop optimizations are important, effective, and numerous

CS 412/413 Spring 2007

Introduction to Compilers

17

## Loop-Invariant Code Motion

- If result of a statement or expression does not change during loop, and it has no externally-visible side-effect (!), can *hoist* its computation out of the loop
- Often useful for array element addressing computations – invariant code not visible at source level
- Requires analysis to identify loop-invariant expressions

CS 412/413 Spring 2007

Introduction to Compilers

18

## Code Motion Example

- Identify invariant expression:

```
for(i=0; i<n; i++)
 a[i] = a[i] + (x*x)/(y*y);
```

- Hoist the expression out of the loop:

```
c = (x*x)/(y*y);
for(i=0; i<n; i++)
 a[i] = a[i] + c;
```

CS 412/413 Spring 2007

Introduction to Compilers

19

## Another Example

- Can also hoist statements out of loops
- Assume x not updated in the loop body:

```
...
while (...) {
 y = x*x;
 ...
}
...
⇒
y = x*x;
while (...) {
 ...
}
...
```

- ... Is it safe?

CS 412/413 Spring 2007

Introduction to Compilers

20

## Strength Reduction

- Replaces expensive operations (multiplies, divides) by cheap ones (adds, subtracts)
- Strength reduction more effective in loops
- Induction variable** = loop variable whose value depends linearly on the iteration number
- Apply strength reduction to induction variables

```
s = 0;
for (i = 0; i < n; i++) {
 v = 4*i;
 s = s + v;
}
⇒
s = 0; v = -4;
for (i = 0; i < n; i++) {
 v = v + 4;
 s = s + v;
}
```

CS 412/413 Spring 2007

Introduction to Compilers

21

## Strength Reduction

- Can apply strength reduction to computation other than induction variables:

```
x * 2 ⇒ x + x
i * 2c ⇒ i << c
i / 2c ⇒ i >> c
```

CS 412/413 Spring 2007

Introduction to Compilers

22

## Induction Variable Elimination

- If there are multiple induction variables in a loop, can eliminate the ones that are used only in the test condition
- Need to rewrite test using the other induction variables
- Usually applied after strength reduction

```
s = 0; v = -4;
for (i = 0; i < n; i++) {
 v = v + 4;
 s = s + v;
}
⇒
s = 0; v = -4;
for (; v < (4*n+4);) {
 v = v + 4;
 s = s + v;
}
```

CS 412/413 Spring 2007

Introduction to Compilers

23

## Loop Unrolling

- Execute loop body multiple times at each iteration
- Example:
 

```
for (i = 0; i < n; i++) { S }
```
- Unroll loop four times:
 

```
for (i = 0; i < n-3; i+=4) { S; S; S; S; }
for (; i < n; i++) S;
```
- Gets rid of ¾ of conditional branches!
- Space-time tradeoff: program size increases

CS 412/413 Spring 2007

Introduction to Compilers

24

## Function Inlining

- Replace a function call with the body of the function:

```
int g(int x) { return f(x)-1; }
int f(int n) { int b=1; while (n--) { b = 2*b }; return b; }
```

```
int g(int x) { int r;
 int n = x;
 { int b = 1; while (n--) { b = 2*b }; r = b }
 return r - 1; }
```

- Can inline methods, but more difficult
- ... how about recursive procedures?

CS 412/413 Spring 2007

Introduction to Compilers

25

## Function Cloning

- Create specialized versions of functions that are called from different call sites with different arguments

```
void f(int x[], int n, int m) {
 for(int i=0; i<n; i++) { x[i] = x[i] + i*m; }
}
```

- For a call `f(a, 10, 1)`, create a specialized version of `f`:

```
void f1(int x[]) {
 for(int i=0; i<10; i++) { x[i] = x[i] + i; }
}
```

- For another call `f(b, p, 0)`, create another version `f2(...)`

CS 412/413 Spring 2007

Introduction to Compilers

26

## When to Apply Optimizations

High IR

Function inlining  
Function cloning  
Constant folding  
Constant propagation  
Value numbering  
Dead code elimination  
Loop-invariant code motion  
Common sub-expression elimination  
Strength reduction  
Constant folding & propagation  
Branch prediction/optimization  
Loop unrolling  
Register allocation  
Cache optimization

Low IR

Assembly

CS 412/413 Spring 2007

Introduction to Compilers

27

## Summary

- Many useful optimizations that can transform code to make it faster
- Whole is greater than sum of parts: optimizations should be applied together, sometimes more than once, at different levels
- Problem: when are optimizations are safe?

CS 412/413 Spring 2007

Introduction to Compilers

28