CS412/413
Introduction to Compilers
Radu Rugina

Lecture 17: Introduction to Optimizations
27 Feb 02

Where We Are

Source code (character stream)
if (b == 0) a = b;

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation

Errors

Correct program
In High IR

Optimize

IR Lowering

Program
In Low IR

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

Source code (character stream)
if (b == 0) a = b;

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation

Errors

Correct program
In High IR

IR Lowering

Optimize

Program
In Low IR

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

  \[
  \text{int square}(x) \{ \text{return } x^2; \}
  \]
Where to Optimize?
- Usual goal: improve time performance
- Problem: many optimizations trade off space versus time
- Examples: 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 inlineing
  - Function cloning
  - Constant folding
  - Constant propagation
  - Dead code elimination
  - Loop invariant code motion
  - Common sub-expression elimination
  - Strength reduction
  - Constant folding & propagation
  - Branch prediction optimization
  - Loop unrolling
- Covered in more detail in Muchnick

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 \( n \cdot c \)
  \[ for(i=0; i<10; i++) \{ s = s + i^2; \} \]
- 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
  - \( int \ x = a[2] \)
  - \( t = \text{addr} \ a \)
  - \( t1 = 2*x4 \)
  - \( t = t + t1 \)
  - \( x = [t] \)

Algebraic Simplification
- More general form of constant folding: take advantage of usual simplification rules
  \[ a * 1 \Rightarrow a \]
  \[ a * 0 \Rightarrow 0 \]
  \[ a / 1 \Rightarrow a \]
  \[ a + 0 \Rightarrow a \]
  \[ \text{false} \Rightarrow b \]
  \[ b \text{ & true} \Rightarrow b \]
- Repeatedly apply the above rules
  \[ (y^2 + 0) / 1 \Rightarrow y^2 + 0 \Rightarrow y \]
- Must be careful with floating point!

Copy Propagation
- After assignment \( x = y \), replace uses of \( x \) with \( y \)
- Replace until \( x \) is assigned again
  \[ x = y; \]
  \[ \text{if} \ (x > 1) \Rightarrow x = y; \]
  \[ s = x * f(x - 1); \]
  \[ s = y * f(y - 1); \]
- What if there was an assignment \( y = z \) before?
  - Transitive apply replacements
Common Subexpression Elimination
- If program computes same expression multiple time, can reuse the computed value
- Example:
  
  \[
  a = b+c; \\
  c = b+c; \\
  d = b+c;
  \]
  
  \( \Rightarrow \ c = a; \)
  
  \( \Rightarrow \ d = b+c; \)
- Common subexpressions also occur in low-level code in address calculations for array accesses:
  
  \[ a[i] = b[i] + 1; \]

Unreachable Code Elimination
- Eliminate code which is never executed
- Example:
  
  \[
  \#define debug false
  \]
  
  \[
  s = 1; \\
  \text{if} \ (\text{debug}) \\
  \text{print}(\text{"state = ", s});
  \]
- Unreachable code may not be obvious in low IR (or in high-level languages with unstructured "goto" statements)

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)
  
  \[
  \begin{align*}
  \text{if} \ (\text{false}) \ S & \Rightarrow ; \\
  \text{if} \ (\text{true}) \ S \text{ else } S' & \Rightarrow S \\
  \text{if} \ (\text{false}) \ S \text{ else } S' & \Rightarrow S' \\
  \text{while} \ (\text{false}) \ S & \Rightarrow ; \\
  \text{while} \ (2>3) \ S & \Rightarrow ;
  \end{align*}
  \]

Dead Code Elimination
- If effect of a statement is never observed, eliminate the statement
  
  \[
  x = y+1; \\
  y = 1; \\
  x = 2*z; \]
- Variable is dead if never used after definition
- Eliminate assignments to dead variables
- Other optimizations may create dead code

Loops
- 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

Loop-invariant Code Motion
- If result of a statement or expression does not change during loop, and it has no externally-visible side-effect (1), 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
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;
  ```

Another Example

- Can also hoist statements out of loops
- Assume x, y not updated in the loop body:
  
  ```
  ... 
  while (b) {
  z = x*x;
  ... 
  }
  ```

- ... Is it safe?

Strength Reduction

- Replaces expensive operations (multiplies, divides) by cheap ones (adds, subtracts)
- Strength reduction more effective in loops
- Induction variable = loop variable whose value is determined only by the iteration number
- Apply strength reduction to induction variables
  
  ```
  s = 0; v = 0;
  for (i = 0; i < n; i++) {
  s = s + v;
  v = v + 4;
  }
  ```

Strength Reduction

- Can apply strength reduction to computation other than induction variables:
  
  ```
  x * 2  ⇒  x + x 
  i * 2^c ⇒ i << c 
  i / 2^c ⇒ i >> c
  ```

Induction Variable Elimination

- If there are multiple induction variables in a loop, can eliminate the ones which are used only in the test condition
- Need to rewrite test using the other induction variables
- Usually applied after strength reduction
  
  ```
  s = 0; v = 0;
  for (i = 0; i < n; i++) {
  s = s + v;
  v = v + 4;
  }
  ```

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 = 0; i < n; i+=4) { S; S; S; S; }
  ```

- Gets rid of ¾ of conditional branches!
- Space-time tradeoff: program size increases
Function Inlining

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

```c
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?

Function Cloning

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

```c
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, 2), create a specialized version off:

```c
void f(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(…)

When to Apply Optimizations

- High IR
  - Function inlining  
  - Function cloning
  - Constant folding
  - Constant propagation
  - Value numbering
  - Dead code elimination
  - Loop invariant code motion
  - Common subexpression elimination
  - Strength reduction
- Low IR
  - Constant folding & propagation
  - Branch prediction optimization
  - Loop unrolling
  - Register allocation
  - Cache optimization

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 safe?