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

What are Optimizations?
- Optimizations = code transformations which 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
  
  ```c
  int square(x) { 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
  - Frequent execution 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
  - Constant folding & propagation
  - 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;  \Rightarrow  x = 2.2;
  
  x = 2 + 2 + 2
  \Rightarrow  x = 6
  ```

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
  b || false \Rightarrow b
  b && true \Rightarrow b
  ```

- Repeatedly apply the above rules
  
  ```
  (y*1+0)/1 \Rightarrow y*1+0 \Rightarrow y*1 \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;
  if (x > 1)
  \Rightarrow  if (y > 1)
  s = x * f(x - 1);
  \Rightarrow  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:
  \[
  a = b + c; \\
  c = b + c; \\
  d = b + c;
  \]
  \[
  \Rightarrow \quad a = b + c; \\
  c = a; \\
  d = b + c;
  \]

- Common subexpressions also occur in low-level code in address calculations for array accesses:
  \[
  a[i] = b[i] + 1;
  \]

Unreachable Code Elimination

- Eliminate code which is never executed

- Example:
  
  ```c
  #define debug false
  s = 1;
  if (debug)
    printf("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)

  ```c
  if (false) S \\
  if (true) S else S' \\
  if (false) S else S' \\
  while (false) S \\
  while (2>3) S
  ```

Dead Code Elimination

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

  ```c
  x = y+1; \\
  y = 1; \\
  x = 2*z; \\
  \Rightarrow \quad 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 (!), 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:
  \[
  \text{for}(i=0; i<n; i++) \;
  a[i] = a[i] + \frac{(x^*x)(y^*y)}{y};
  \]
- Hoist the expression out of the loop:
  \[
  c = \frac{(x^*x)(y^*y)}{y};
  \text{for}(i=0; i<n; i++) \;
  a[i] = a[i] + c;
  \]

Another Example

- Can also hoist statements out of loops
- Assume x not updated in the loop body:
  \[
  \text{...}
  \text{while} (...) \{ \quad \Rightarrow \quad y = x^*x;
  \text{...} \}
  \]
- ... 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 by the iteration number
- Apply strength reduction to induction variables
  \[
  s = 0; \; v = 0;
  \text{for } (i = 0; i < n; i++) \{ \quad \Rightarrow \quad v = v+4; \;
  s = s + v; \}
  \]

Strength Reduction

- Can apply strength reduction to computation other than induction variables:
  \[
  x \cdot 2 \quad \Rightarrow \quad x + x \\
  i \cdot 2^c \quad \Rightarrow \quad i << c \\
  i / 2^c \quad \Rightarrow \quad i >> c
  \]

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
  \[
  s = 0; \; v=0;
  \text{for } (i = 0; i < n; i++) \{ \quad \Rightarrow \quad v = v+4; \;
  s = s + v; \}
  \]

Loop Unrolling

- Execute loop body multiple times at each iteration
- Example:
  \[
  \text{for } (i = 0; i < n; i++) \{ \; S \}
  \]
- Unroll loop four times:
  \[
  \text{for } (i = 0; i < n-3; i++) \{ \; S; \; S; \; S; \}
  \]
- Gets rid of \(\frac{3}{4}\) 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, 1), create a specialized version of f:
  ```c
  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(...)

**When to Apply Optimizations**

<table>
<thead>
<tr>
<th>High IR</th>
<th>Function inlining</th>
<th>Function cloning</th>
<th>Constant folding</th>
<th>Constant propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low IR</td>
<td>Value numbering</td>
<td>Dead code elimination</td>
<td>Loop-invariant code motion</td>
<td>Common sub-expression elimination</td>
</tr>
<tr>
<td>Assembly</td>
<td>Strength reduction</td>
<td>Constant folding &amp; propagation</td>
<td>Branch prediction/optimization</td>
<td>Loop unrolling</td>
</tr>
<tr>
<td></td>
<td>Register allocation</td>
<td>Cache optimization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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?