Introduction to Compilers
Tim Teitelbaum

Lecture 23: Introduction to Optimizations
14 March 08
Where We Are

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

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation

Correct program
In High IR

Errors

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

Correct program In High IR
Optimize

IR Lowering
Optimize

Program In Low IR
Errors
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

\[
\text{int square(x) \{ return x*x; \}}
\]
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:

\[
\begin{align*}
    n &= 10 \\
    c &= 2 \\
    \text{for (i=0; i<n; i++) } & \{ \ s = s + i*c; \ \}
\end{align*}
\]

• Replace n, c:

\[
\begin{align*}
    \text{for (i=0; i<10; i++) } & \{ \ s = s + i*2; \ \}
\end{align*}
\]

• 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:

  \[ x = 1.1 \times 2; \quad \Rightarrow \quad x = 2.2; \]

• Performed at every stage of compilation
  – Constants created by translations or optimizations

  \[
  \text{int } x = a[2] \quad \Rightarrow \quad t1 = 2 \times 4 \\
  t2 = a + t1 \\
  x = *t2
  \]
Algebraic Simplification

- More general form of constant folding: take advantage of usual simplification rules
  
  \[
  \begin{align*}
  a \times 1 & \Rightarrow a & a \times 0 & \Rightarrow 0 \\
  a \div 1 & \Rightarrow a & a + 0 & \Rightarrow a \\
  b \mathbin{||} \text{false} & \Rightarrow b & b \mathbin{&&} \text{true} & \Rightarrow b
  \end{align*}
  \]

- Repeatedly apply the above rules
  \[
  (y \times 1+0)/1 \Rightarrow y \times 1+0 \Rightarrow y \times 1 \Rightarrow y
  \]

- Must be careful with floating point!
Copy Propagation

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

\[
x = y; \\
\text{if } (x > 1) \quad \Rightarrow \quad \text{if } (y > 1) \\
s = x \times f(x - 1); \\
s = y \times f(y - 1);
\]

• What if there was an assignment $y = z$ before?
  – Transitiveley apply replacements
Common Subexpression Elimination

• If program computes same expression multiple time, can reuse the computed value

• Example:

\[
\begin{align*}
a &= b+c; \\
c &= b+c; \
\Rightarrow 
\end{align*}
\]

• Common subexpressions also occur in low-level code in address calculations for array accesses:

\[
a[i] = b[i] + 1;
\]
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)
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)

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

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

\[
\begin{align*}
x &= y+1; \\
y &= 1; \\
x &= 2*z;
\end{align*}
\Rightarrow
\begin{align*}
y &= 1; \\
x &= 2*z;
\end{align*}
\]

• Variable is *dead* if value is never used after definition

• Eliminate assignments to dead variables

• Other optimizations may create dead code
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
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:

\[
\text{for}(i=0; \ i<n; \ i++) \\
\quad a[i] = a[i] + (x*x)/(y*y);
\]

• Hoist the expression out of the loop:

\[
\text{c} = (x*x)/(y*y);
\text{for}(i=0; \ i<n; \ i++) \\
\quad a[i] = a[i] + c;
\]
Another Example

• Can also hoist statements out of loops
• Assume \( x \) not updated in the loop body:

\[
\begin{align*}
\ldots & \quad \ldots \\
\text{while (\ldots) } & \quad \text{while (\ldots) } \\
\quad y = x^2; & \quad y = x^2; \\
\Rightarrow & \quad \Rightarrow \\
\ldots & \quad \ldots \\
\ldots & \quad \ldots \\
\ldots & \quad \ldots \\
\ldots & \quad \ldots \\
\ldots & \quad \ldots \\
\end{align*}
\]

• …Is it safe?
Strength Reduction

- Replaces expensive operations (multiplies, divides) by cheap ones (adds, subtractions)
- Strength reduction more effective in loops

- Induction variable = loop variable whose value depends linearly on the iteration number
- Apply strength reduction to induction variables

```plaintext
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;
}
⇒
```
Strength Reduction

- Can apply strength reduction to computation other than induction variables:
  
  \[
  \begin{align*}
  x \times 2 & \Rightarrow x + x \\
  i \times 2^c & \Rightarrow i \ll c \\
  i / 2^c & \Rightarrow i \gg c
  \end{align*}
  \]
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

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

\[
\Rightarrow \\
s = 0; \ v = -4; \\
\text{for (} ; v < (4*n-4); \text{) } \{ \\
\quad v = v+4; \\
\quad s = s + v; \\
\}
\]
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 $\frac{3}{4}$ 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>Low IR</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function inlining</td>
<td>Loop-invariant code motion</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Function cloning</td>
<td>Common sub-expression elimination</td>
<td>Cache optimization</td>
</tr>
<tr>
<td>Constant folding</td>
<td>Strength reduction</td>
<td></td>
</tr>
<tr>
<td>Constant propagation</td>
<td>Constant folding &amp; propagation</td>
<td></td>
</tr>
<tr>
<td>Value numbering</td>
<td>Branch prediction/optimization</td>
<td></td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>Loop unrolling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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?