CSCI 742 - Compiler Construction

Lecture 31
Introduction to Optimizations
Instructor: Hossein Hojjat

April 11, 2018

What Next?

- At this point we can generate bytecode for a given program
- Next: how to generate better code through optimization
- Most complexity in modern compilers is in their optimizers
- This course covers some straightforward optimizations
- There is much more to learn!

"Advanced Compiler Design and Implementation" (Whale Book)
by Steven Muchnick

- 10 chapters ($\sim$ 400 pages) on optimization techniques
- Maybe an independent study? $\ddot\smile$

Goal of Optimization

- **Optimizations:** code transformations that improve the program
- Must not change meaning of program to behavior not allowed by source code
- Different kinds
- Space optimizations: reduce memory use
- Time optimizations: reduce execution time
- Power optimization: reduce power usage

Why Optimize?

- Programmers may write suboptimal code to make it clearer
- Many programmers cannot recognize ways to improve the efficiency

Example.

- Assume `a` is a field of a class
- `a[i][j] = a[i][j] + 1;` 18 bytecode instructions (gets the field `a` twice)
- `a[i][j]++;` 12 bytecode instructions (gets the field `a` once)

- High-level language may make some optimizations inconvenient or impossible to express

Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade off space versus time

Example.
Loop unrolling here reduces the number of iterations from 100 to 50

```
for (i = 0; i < 100; i++)
  f ();
```

```
for (i = 0; i < 100; i += 2) {
  f ();
  f ();
}
```

Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade off space versus time

- Loop unrolling increases code space, speeds up one loop
- Frequently-executed code with long loops:
  Preferably unroll the loop
  - Optimize code execution time at expense of space
- Infrequently-executed code:
  - Optimize code space at expense of execution time
  - Save instruction cache space
- Want to optimize program **hot** spots

Writing Fast Programs

- Design for locality and few operations
- Use the right algorithm and data structures
- Turn on optimization and use a profiler (e.g. JProfiler) to figure out hot spots
- Tweak source code until optimizer does "the right thing"
- Understanding optimizers helps!

Common Optimizations

- Constant Propagation
- Constant Folding
- Algebraic Simplification
- Unreachable Code Elimination
- Dead Code Elimination
- Function Inlining
- Copy Propagation
- Common Subexpression Elimination
- Loop-invariant Code Motion
- Strength Reduction

Constant Propagation

- If value of variable is known to be a constant, replace use of variable with constant
- Value of variable must be propagated forward from point of assignment

Example.

```
n = 10;
c = 5;
for (int i=0; i<n; i++) {
  s = s + i*c;
}
```

- Replace `n`, `c`

```
for (int i=0; i<10; i++) {
  s = s + i*5;
}
```

Constant Folding

- If operands are known at compile time, evaluate at compile time when possible

```
float x = 2.1 * 2;    ⇒    float x = 4.2;
```

- Useful at every stage of compilation
- Constant expressions are created by translation and by optimization

```
a = 7;
b = 2;
...
x = a - b;
while(x < 10){
...
}
```

Constant Propagation ⇒

```
a = 7;
b = 2;
...
x = 7 - 2;
while(x < 10){
...
}
```

Constant Folding ⇒

```
a = 7;
b = 2;
...
x = 5;
while(x < 10){
...
}
```

Constant Folding Control Structures

`if (true) S`	$\Rightarrow$	`S`
`if (false) S`	$\Rightarrow$	`{}`
`if (true) S else S'`	$\Rightarrow$	`S`
`if (false) S else S'`	$\Rightarrow$	`S'`
`while (false) S`	$\Rightarrow$	`{}`

Example.

`if (2 > 3) S` $\Rightarrow$ `if (false) S` $\Rightarrow$ `{}`

Algebraic Simplification

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

Example: Identities

`a * 1 ⇒ a` `a * 0 ⇒ 0` `a + 0 ⇒ a`

`b || false ⇒ b` `b && true ⇒ b`

`b || true ⇒ true` `b && false ⇒ false`

Example: Reassociation

Reassociate commutative expressions in an order that is better for e.g. constant folding

`(a + 2) + 2 ⇒ a + (2 + 2) ⇒ a + 4`

- Must be careful with floating point and with overflow
    - Algebraic rules may give wrong or less precise answers

Unreachable Code Elimination

- Remove code that will never be executed regardless of the values of variables at run time
- Reductions in code size improve cache, TLB performance

```
public int f() {
  return 0;
  int i = 0; // Unreachable code
}
```

- Unreachability is a control-flow property:

"May control ever arrive at this point?"

Dead Code Elimination

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

```
x = y - 1;
y = 5;
x = z + 1;
```

$\Rightarrow$

```
y = 5;
x = z + 1;
```

- Variable is **dead** if value is never used after definition
- Eliminate assignments to dead variables
- Other optimizations may create dead code
- Deadness is a data-flow property:

"May this data ever arrive anywhere?"

Function Inlining

- Replace a function call with the body of the function

```
int max( int a, int b ) {
  return a>b ? a : b;
}
int x = max(5,4);
```

$\Rightarrow$

```
int x = 5>4 ? 5 : 4;
```

- May need to rename variables to avoid **name capture**: same name happen to be in use at both the caller and inside the callee for different purposes
- How about recursive functions?

Copy Propagation

- Like constant propagation, instead of constant a variable is used
- After assignment `x = y`, replace subsequent uses of `x` with `y`
- Replace until `x` is assigned again
- May make `x` a dead variable, result in dead code

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

$\Rightarrow$

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

Common Subexpression Elimination

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

```
a = b+c;            a = b+c;
c = b+c;     ⇒      c = a;
d = b+c;            d = b+c;
```

- Common subexpressions also occur in code generation

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

- In a language like C need to compute memory offset for multi-dimensional arrays

```
a[i][j] = b[i][j]+1; // offset = i * #columns + j
```

Loop-invariant Code Motion

- If a statement or an expression does not change during loop, and has no externally-visible side effect, can move before loop

Example.

- Identify invariant expression:

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

- Move the expression out of the loop

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

Strength Reduction

- Replace expensive operations (*,/) by cheap ones (+,-) via dependent induction variable
- **Induction variable:** loop variable whose value is depends linearly on the iteration number

```
for (int i = 0; i < n; i++) {
  a[i*3] = i;
}
```

```
int j = 0;
for (int i = 0; i < n; i++) {
  a[j] = i;
  j = j + 3;
}
```