Lecture #30

Code Optimization

Code Optimization

- Most complexity in modern compilers is in the optimizer
    - Also by far the largest phase
- Optimizations can be performed on
    - AST / DAG
        - Machine independent optimization
    - Assembly code
        - Target dependent optimization
    - On an intermediate language

Basic Blocks

- A basic block is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)

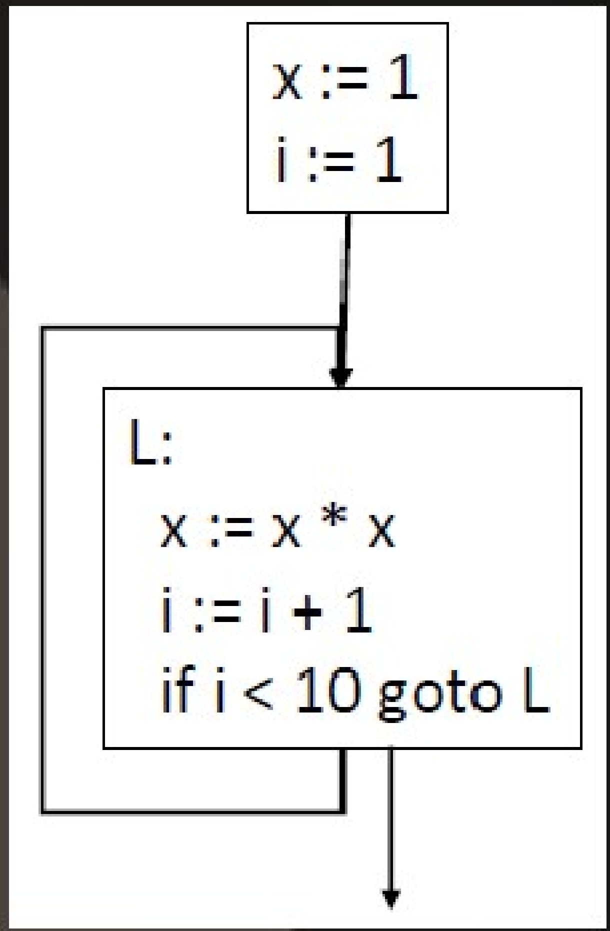
- Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment

- *The property of sequential control flow can be useful for many optimizations.*

Control Flow Graphs

- Control-Flow Graph (CFG)
  - Models the way that the code transfers control between blocks in the procedure.

    - *Node*: a single basic block
    - *Edge*: transfer of control between basic blocks.
    - All *return* nodes are terminal



Code Optimization

- Optimization seeks to improve a program's resource utilization
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Memory Usages
  - Disk Accesses
  - Power
- Optimization should not alter what the program computes
  - *The answer before and after optimization must remain same*

Code Optimization

- There are three granularities of optimizations
  - Local optimizations
    - Apply to a basic block in isolation
  - Global optimizations
    - Apply to a control-flow graph (method body) in isolation
  - Inter-procedural optimizations
    - Apply across method boundaries

- *Most compilers do local and global optimizations but not the third*

- Often a conscious decision is made not to implement the fanciest optimization known
  - Goal: Maximum benefit for minimum cost

Code Optimization

- There are three granularities of optimizations
  - Local optimizations
    - Apply to a basic block in isolation
  - Global optimizations
    - Apply to a control-flow graph (method body) in isolation
  - Inter-procedural optimizations
    - Apply across method boundaries

- *Most compilers do local and global optimizations but not the third*

- Often a conscious decision is made not to implement the fanciest optimization known
  - Goal: Maximum benefit for minimum cost

Local Optimization

- Constant Folding
  - Evaluation at compile-time of expressions whose operands are known to be constant
  - Example

a = 10 * 5 + 6 - b;

t0 = 10 ;
t1 = 5 ;
t2 = t0 * t1 ;
t3 = 6 ;
t4 = t2 + t3 ;
t5 = t4 – b;
a = t5 ;

t0 = 56 ;
t1 = t0 – b ;
a = t1 ;

Local Optimization

- Constant Propagation
  - If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable.
  - Example:

```
t0 = 12 ;
t1 = arr + t0 ;
t2 = *(t1) ;
```

```
li $t0, 12
lw $t1, 8($fp)
add $t2, $t1, $t0
lw $t3, 0($t2)
```

Constant propagation + rearrangement cuts no. of regs. and insns. from 4 to 2

```
t0 = *(arr + 12) ;
```

```
lw $t0, -8($fp)
lw $t1, 12($t0)
```

Local Optimization

- Algebraic simplification and Reassociation
  - Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions.
  - Reassociation refers to using properties such as associativity, commutativity and distributivity to rearrange an expression to enable other optimizations

Local Optimization

- Algebraic simplification and Reassociation
  - Simplification Examples:

x+0 = x	0+x = x	x*1 = x	1*x = x	0/x = 0	x-0 = x
b && true = b			b && false = false		
b || true = true			b || false = b		

  - Example: Re-arrangement + constant folding

```
b = 5 + a + 10 ;
```

→

```
t0 = 5 ;
t1 = t0 + a ;
t2 = t1 + 10 ;
b = t2 ;
```

→

```
t0 = 15 ;
t1 = a + t0 ;
b = t1 ;
```

Local Optimization

- Operator Strength Reduction
  - Replaces an operator by a "less expensive" one
  - Often performed as part of *loop-induction variable elimination*
  - Example:

```
while (i < 100)
{
    arr[i] = 0;
    i = i + 1;
}
```

⇨

```
t1 = i;
L0:
If  t1>100 Goto L1 ;
t2 = 4 * t1 ;
t3 = arr + t2 ;
*(t3) = 0 ;
t1 = t1 + 1 ;
Jump L0
L1:
```

⇨

```
t1 = arr ;
L0: If  i > 100  Goto L1 ;
* t1 = 0;
t1 = t1 + 4;
i = i + 1 ;
L1:
```

Local Optimization

- Copy Propagation
  - Similar to constant propagation, but generalized to non-constant values
  - For a = b, we can replace later occurrences of a with b (assuming there are no changes to either variable in-between)
  - Example

```
t2 = t1 ;
t3 = t2 * t1;
t4 = t3 ;
t5 = t3 * t2 ;
c = t5 + t4 ;
```

→

```
t3 = t1 * t1 ;
t5 = t3 * t1 ;
c = t5 + t3 ;
```