CS 160 Compilers

Lecture 15: Optimization

Yu Feng

Fall 2021

Slide designed by Prof. Alex Aiken, with modifications

Optimization

- Optimization is ou[illegible]piler phase
- Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase
- First, we need to discuss intermediate languages

Why IR?

- When should we perform optimizations?
  - On AST
    - Pro: Machine independent
    - Con: Too high level
  - On assembly language
    - Pro: Exposes optimization opportunities
    - Con: Machine dependent
    - Con: Must reimplement optimizations when retargetting
  - On an intermediate language
    - Pro: Machine independent
    - Pro: Exposes optimization opportunities

Intermediate Languages

- Intermediate langu[illegible]h-level assembly
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    - E.g., push translates to several assembly instructions
    - Most opcodes correspond directly to assembly opcodes

Three Address IR

- Each instruction is o

$$x := y \text{ op } z$$

$$x := \text{op } y$$

- y and z are registers or constants

- Common form of intermediate code

- The expression x + y * z is translated

$$t_1 := y * z$$

$$t_2 := x + t_1$$

- Each subexpression has a name

Intermediate Code Generation

- Similar to assemb[illegible]neration
- But use any number of IL registers to hold intermediate results

Intermediate Code Generation

- You should be able to use intermediate code
    - At the level discussed in lecture
- You are not expected to know how to generate intermediate code
    - Because we won't discuss it
    - But really just a variation on code generation . . .

An Intermediate Language

P → S P | ε
S → id := id op id
  | id := op id
  | id := id
  | push id
  | id := pop
  | if id relop id goto L
  | L:
  | jump L

- id's are register names
- Constants can replace id's
- Typical operators: +, -, *

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

Basic Block Example

- Consider the basic blo

```
1. L:
2.   t := 2 * x
3.   w := t + x
4.   if w > 0 goto L'
```


- (3) executes only after (2)
  - We can change (3) to w := 3 * x
  - Can we eliminate (2) as well?

Control-Flow Graphs

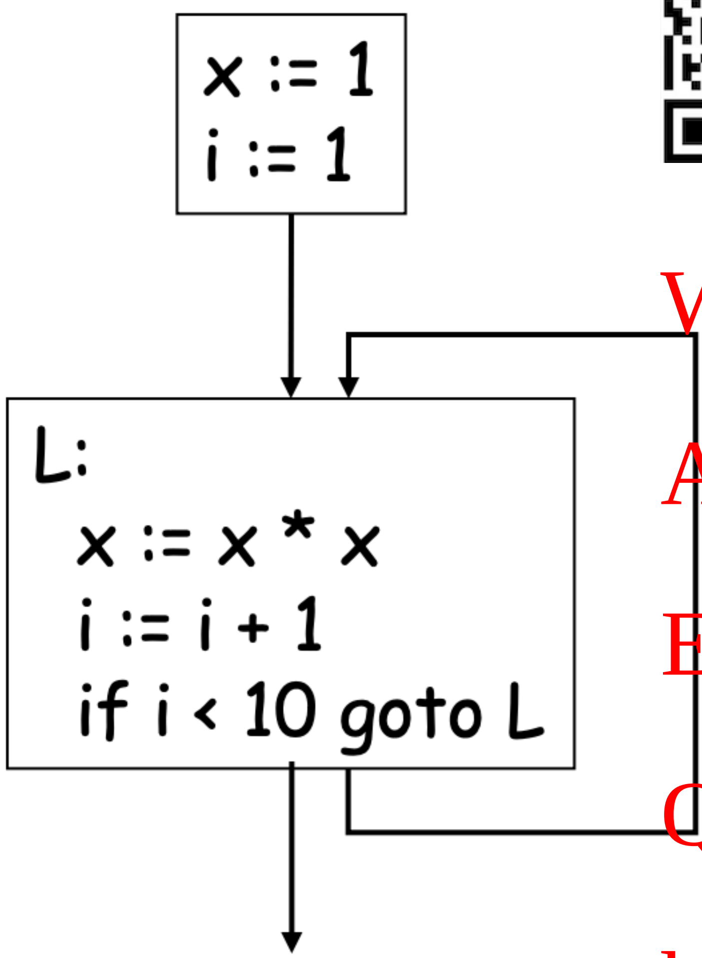
- A control-flow gra[illegible]rected graph with
  - Basic blocks as nodes

  - An edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B

    - E.g., the last instruction in A is *jump* L_B

    - E.g., execution can fall-through from block A to block B

CFG Example



- The body of a method (or procedure) can be represented as a control-flow graph

- There is one initial node

- All "return" nodes are terminal

Optimization Overview

- Optimization seek[illegible]ve a program’s resource utilization
    - Execution time (most often)
    - Code size
    - Network messages sent, etc.
- Optimization should not alter what the program computes
    - The answer must still be the same

Classification of Optimization

- 1. ***Local*** optimizations: Apply to a basic block in isolation
- 2. ***Global*** optimizations: Apply to a control-flow graph (method body) in isolation
- 3. ***Inter-procedural*** optimizations: Apply across method boundaries
- Most compilers do (1), many do (2), few do (3)

Cost of Optimization

- In practice, a conscious decision is made not to implement the fanciest optimization known

- Why?

  - Some optimizations are hard to implement
  - Some optimizations are costly in compilation time
  - Some optimizations have low benefit
  - Many fancy optimizations are all three!

- Goal: Maximum benefit for minimum cost

Local Optimization

- The simplest form [illegible]zations
- No need to analyze the whole procedure body

  - Just the basic block in question

- Example: algebraic simplification

Algebraic Simplification

- Some statements can be deleted

```
x := x + 0
x := x * 1
```

- Some statements can be simplified

```
x := x * 0      ⇒   x := 0
y := y ** 2     ⇒   y := y * y
x := x * 8      ⇒   x := x << 3
x := x * 15     ⇒   t := x << 4; x := t - x
```

Constant Folding

- Operations on constants can be computed at compile time
  - If there is a statement *x := y op z*

  - And y and z are constants

  - Then *y op z* can be computed at compile time

- Example: *x := 2 + 2* => *x := 4*

- Example: *if 2 < 0 jump L* can be deleted

- When might constant folding be dangerous?

Control-flow Optimizations

- Eliminate unreachable basic blocks:
  - Code that is unreachable from the initial block
  - E.g., basic blocks that are not the target of any jump or "fall through" from a conditional
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller
  - And sometimes also faster
  - Due to memory cache effects (increased spatial locality)

Static Single Assignment (SSA)

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment

- Rewrite intermediate code in *single assignment form*


```
x := z + y         b := z + y
a := x       ⇒     a := b
x := 2 * x         x := 2 * b
```

(b is a fresh register)

Non-trivial due to loops and recursions

Common Subexpression Elimination

- *If*
  - Basic block is in single assignment form
  - A definition x := is the first use of x in a block

- ***Then***

  - When two assignments have the same rhs, they compute the same value

- Example:

```
x := y + z              x := y + z
…             ⇒         …
w := y + z              w := x
```

(the values of x, y, and z do not change in the … code)

Copy Propagation

- If $w := x$ appears in [illegible] replace subsequent uses of w with uses of x
  - Assumes single assignment form

  - Example:


```
b := z + y                    b := z + y
a := b           ⇒            a := b
x := 2 * a                    x := 2 * b
```


  - Only useful for enabling other optimizations

    - Constant folding
    - Dead code elimination

Applying Local Optimizations

- Each local optimi[illegible]s little by itself
- Typically optimizations interact

    - Performing one optimization enables another

- Optimizing compilers repeat optimizations until no improvement is possible

    - The optimizer can also be stopped at any point to limit compilation time

Peephole Optimizations

- Write peephole optimizations as replacement rules where the rhs is the improved version of the lhs

$i_1, \ldots, i_n \to j_1, \ldots, j_m$

- The “peephole” is a short sequence of (usually contiguous) instructions

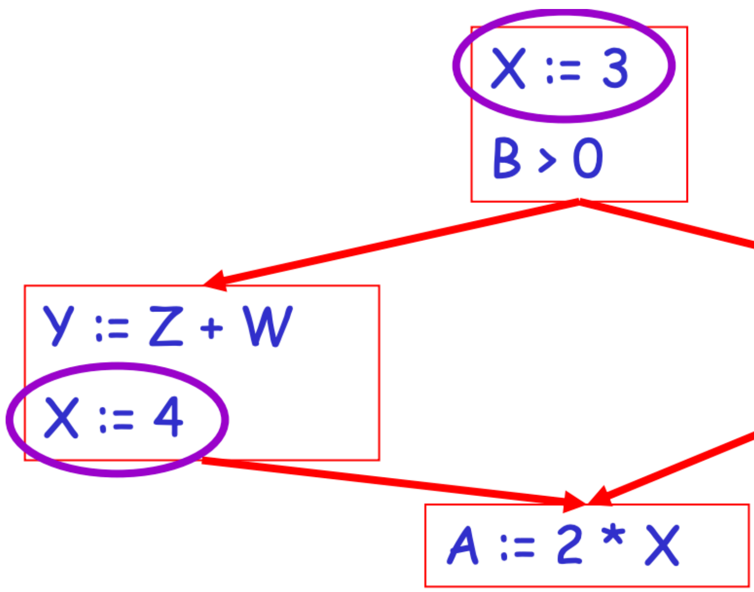
- The optimizer replaces the sequence with another equivalent one (but faster)

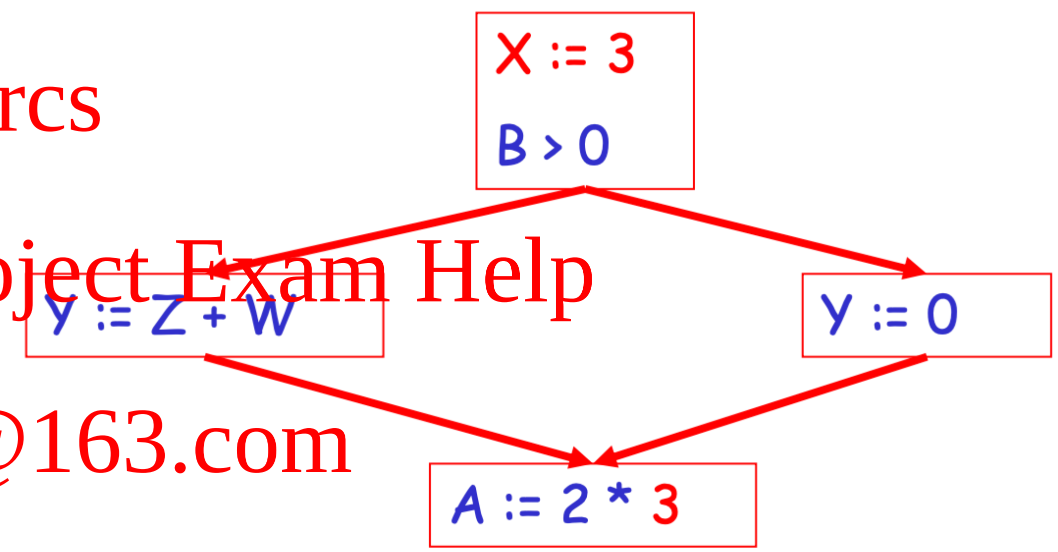
Global Optimizations

- Extend same optimizations to an entire control-flow graph

Global Optimizations

- Extend same optimizations to an entire control-flow graph





Correctness

- The correctness co [illegible] not trivial to check
- “All paths” includes paths around loops and through branches of conditionals
- Checking the condition requires global analysis
- An analysis of the entire control-flow graph

An Example

- Initial code:

```
a := x ^ 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic … tion:

```
a := x ^ 2
b := 3
c := x
d := c * c
e := d * 2
f := a + d
g := e * f
```

An Example

- Algebraic ... tion:

```
a := ^  ^
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

An Example

- Copy prop

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

An Example

- Copy prop

```
a := x * x
b := 3
c := x
d := x * x
e := 3 * 1
f := a + d
g := e * f
```

An Example

- Constant

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Dead cod[...]on:

a := x * x

f := a + a
g := 6 * f

- This is the final form