

Intermediate Code; Local Optimization

Adapted from Lectures by
Prof. Alex Aiken and George Necula (UCB)

Code Generation Summary

- We have discussed
 - Runtime organization
 - Simple stack machine code generation
 - Improvements to stack machine code generation
- Our compiler goes directly from AST to assembly language
 - And does not perform optimizations
- Most real compilers use intermediate languages

Why Intermediate Languages?

- When to 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

- Each compiler uses its own intermediate language
 - IL design is still an active area of research
- Intermediate language = high-level assembly language
 - 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 Intermediate Code

- Each instruction is of the form
$$x := y \text{ op } z$$
 - `y` and `z` can be only registers or constants
 - Just like assembly
- Common form of intermediate code
- The AST expression `x + y * z` is translated as
$$t_1 := y * z$$
$$t_2 := x + t_1$$
 - Each subexpression has a "home"

Generating Intermediate Code

- Similar to assembly code generation except that it can use unlimited number of IL registers to hold intermediate results
- `Igen(e, t)` function generates code to compute the value of `e` in register `t`
- Example:
$$\text{igen}(e_1 + e_2, t) =$$
$$\begin{array}{ll} \text{igen}(e_1, t_1) & (t_1 \text{ is a fresh register}) \\ \text{igen}(e_2, t_2) & (t_2 \text{ is a fresh register}) \\ t := t_1 + t_2 & \end{array}$$



An Intermediate Language

$P \rightarrow S P \mid \epsilon$
 $S \rightarrow id := id \text{ op } id$
 | $id := op \ id$
 | $id := id$
 | $push \ id$
 | $id := pop$
 | $if \ id \ relop \ id \ goto \ L$
 | $L:$
 | $jump \ L$

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

CS781(Prasad)

L26IC

7

Definition. 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)
 - Each instruction in a basic block is executed after all the preceding instructions have been executed

CS781(Prasad)

L26IC

8

Basic Block Example

- Consider the basic block
 1. $L:$
 2. $t := 2 * x$
 3. $w := t + x$
 4. $if \ w > 0 \ goto \ L'$
- No way for (3) to be executed without (2) having been executed right before
 - We know (3) can be changed to $w := 3 * x$
 - Can we eliminate (2) as well?

CS781(Prasad)

L26IC

9

Definition. Control-Flow Graphs

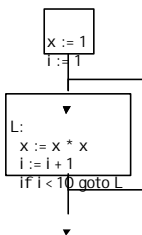
- A **control-flow graph** is a directed graph with
 - Basic blocks as nodes
 - An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B
 - E.g., the last instruction in A is $jump \ L_B$
 - E.g., the execution can fall-through from block A to block B

CS781(Prasad)

L26IC

10

Control-Flow Graphs. 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

CS781(Prasad)

L26IC

11

Optimization Overview

- Optimization seeks to improve a program's utilization of some resource
 - 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

CS781(Prasad)

L26IC

12

A Classification of Optimizations

- For languages like C and Cool there are three granularities of optimizations
 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) and very few do (3)

CS781(Prasad)

L26IC

13

Cost of Optimizations

- 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 terms of compilation time
 - The fancy optimizations are both hard and costly
- **The goal:** maximum improvement with minimum of cost

CS781(Prasad)

L26IC

14

Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
 - Just the basic block in question
- **Example:** algebraic simplification

CS781(Prasad)

L26IC

15

Algebraic Simplification

- Some statements can be deleted
 - $x := x + 0$
 - $x := x * 1$
- Some statements can be simplified
 - $x := x * 0 \Rightarrow x := 0$
 - $y := y ** 2 \Rightarrow y := y * y$
 - $x := x * 8 \Rightarrow x := x \ll 3$
 - $x := x * 15 \Rightarrow t := x \ll 4; x := t - x$
(on some machines \ll is faster than $*$; but not on all!)

CS781(Prasad)

L26IC

16

Constant Folding

- Operations on constants can be computed at compile time
- In general, if there is a statement
 - $x := y \text{ op } z$
 - And y and z are constants
 - Then $y \text{ op } z$ can be computed at compile time
- **Example:** $x := 2 + 2 \Rightarrow x := 4$
- **Example:** if $2 < 0$ jump L can be deleted

CS781(Prasad)

L26IC

17

Flow of Control Optimizations

- Eliminating unreachable code:
 - Code that is unreachable in the control-flow graph
 - 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)

CS781(Prasad)

L26IC

18

Single Assignment Form

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment
- Intermediate code can be rewritten to be 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)
```

- More complicated in general, due to loops

CS781(Prasad)

L261C

19

Common Subexpression Elimination

- Assume
 - Basic block is in single assignment form
 - A definition $x :=$ is the first use of x in a block
- If any 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)
```

CS781(Prasad)

L261C

20

Copy Propagation

- If $w := x$ appears in a block, all subsequent uses of w can be replaced with uses of x

- Example:

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

- This does not make the program smaller or faster but might enable other optimizations
 - Constant folding
 - Dead code elimination

CS781(Prasad)

L261C

21

Copy Propagation and Constant Folding

- Example:

```
a := 5          a := 5
x := 2 * a      ⇒  x := 10
y := x + 6      y := 16
t := x * y      t := x << 4
```

CS781(Prasad)

L261C

22

Copy Propagation and Dead Code Elimination

If

$w := rhs$ appears in a basic block
 w does not appear anywhere else in the program

Then

the statement $w := rhs$ is dead and can be eliminated
- **Dead** = does not contribute to the program's result

Example: (a is not used anywhere else)

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

CS781(Prasad)

L261C

23

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations interact
 - Performing one optimization enables other opt.
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
 - The optimizer can also be stopped at any time to limit the compilation time

CS781(Prasad)

L261C

24



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 optimization:

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

An Example

- Algebraic optimization:

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

An Example

- Copy propagation:

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

An Example

- Copy propagation:

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

An Example

- Constant folding:

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

An Example

- Constant folding:
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f

An Example

- Common subexpression elimination:
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f

An Example

- Common subexpression elimination:
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f

An Example

- Copy propagation:
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f

An Example

- Copy propagation:
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f

An Example

- Dead code elimination:
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f



An Example

- Dead code elimination:

```
a := x * x
```

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

- This is the final form

Peephole Optimizations on Assembly Code

- The optimizations presented before work on intermediate code
 - They are target independent
 - But they can be applied on assembly code also
- **Peephole optimization** is an effective technique for improving assembly code
 - The "peephole" is a short sequence of (usually contiguous) instructions
 - The optimizer replaces the sequence with another equivalent (but faster) one

Peephole Optimizations (Cont.)

- Write peephole optimizations as replacement rules

$$i_1, \dots, i_n \rightarrow j_1, \dots, j_m$$

where the rhs is the improved version of the lhs

- **Example:**

```
move $a $b, move $b $a → move $a $b
```

- Works if `move $b $a` is not the target of a jump

- **Another example:**

```
addiu $a $a i, addiu $a $a j → addiu $a $a i+j
```

Peephole Optimizations (Cont.)

- Many (but not all) of the basic block optimizations can be cast as peephole optimizations
 - **Example:** `addiu $a $b 0` → `move $a $b`
 - **Example:** `move $a $a` →
 - These two together eliminate `addiu $a $a 0`
- Just like for local optimizations, peephole optimizations need to be applied repeatedly to get maximum effect

Local Optimizations. Notes.

- Intermediate code is helpful for many optimizations
- Many simple optimizations can still be applied on assembly language code
- "Program optimization" is grossly misnamed
 - Code produced by "optimizers" is not optimal in any reasonable sense
 - "Program improvement" is a more appropriate term