IL for Arrays & Local Optimizations

Lecture 26

(Adapted from notes by R. Bodik and G. Necula)

Multi-dimensional Arrays

- A 2D array is a 1D array of 1D arrays
- Java uses arrays of pointers to arrays for >1D arrays.
- But if row size constant, for faster access and compactness, may prefer to represent an *MxN* array as a 1D array of 1D rows (not pointers to rows): *row-major order*
- FORTRAN layout is 1D array of 1D columns: *column-major order.*

IL for 2D Arrays (Row-Major Order)

- Again, let S be size of one element, so that a row of length N has size NxS.

  igen(e_1[e_2,e_3], t) =
  igen(e_1, t_1); igen(e_2,t_2); igen(e_3,t_3)
  igen(N, t_4) (N need not be constant)
  t_5 := t_4 * t_2; t_6 := t_5 + t_3;
  t_7 := t_6*S;
  t_8 := t_7 + t_1
  t := *t_8

Array Descriptors

- Calculation of element address for $e_1[e_2,e_3]$ has form $VO + S_1 \times e_2 + S_2 \times e_3$, where
  - VO (address of $e_1[0,0]$) is the *virtual origin*
  - S_1 and S_2 are *strides*
  - All three of these are constant throughout lifetime of array
- Common to package these up into an *array descriptor,* which can be passed in lieu of the array itself.

Array Descriptors (II)

- By judicious choice of descriptor values, can make the same formula work for different kinds of array.
- For example, if lower bounds of indices are 1 rather than 0, must compute

  address of $e[1,1] + S_1 \times (e_2-1) + S_2 \times (e_3-1)$
- But some algebra puts this into the form

  $VO + S_1 \times e_2 + S_2 \times e_3$

  where VO = address of $e[1,1] - S_1 - S_2$

Observation

- These examples show profligate use of registers.
- Doesn't matter, because this is Intermediate Code. Rely on later optimization stages to do the right thing.

Code Optimization: Basic Concepts

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 in 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

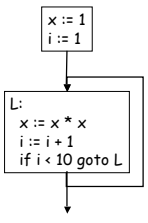
Basic Block Example

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

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
- Frequently abbreviated as CFG

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

Optimization Overview

- Optimization seeks to improve a program's utilization of some resource
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Battery power used, etc.
- Optimization should not alter what the program computes
  - The answer must still be the same

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)

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

Local Optimizations

- The simplest form of optimizations
- 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

  (on some machines << is faster than *; but not on all!)

Constant Folding

- Operations on constants can be computed at compile time
- In general, 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?

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
  - Such basic blocks can be eliminated
- 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)

Single Assignment Form

- Some optimizations are simplified if each assignment is to a temporary that has not appeared already in the basic block
- Intermediate code can be rewritten to be in single assignment form

```
x := a + y          x := a + y
a := x        ⇒     a1 := x
x := a * x          x1 := a1 * x
b := x + a          b := x1 + a1
```

(x_1 and a_1 are fresh temporaries)

Common Subexpression Elimination

- Assume
  - Basic block is in *single assignment form*
- All assignments with same rhs compute the same value
- Example:

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

- Why is single assignment important here?

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
- Again, single assignment is important here.

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
```


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


Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations interact
  - Performing one optimizations 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

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 + b
f := a + d
g := e * f
```


An Example

- Copy propagation:

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


An Example

- Copy propagation:

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


An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
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 language 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

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

  where the rhs is the improved version of the lhs
- Examples:

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

    addiu $a $b k, lw $c ($a) → lw $c k($b)
  - Works if $a not used later (is "dead")

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
- "Program optimization" is grossly misnamed
  - Code produced by "optimizers" is not optimal in any reasonable sense
  - "Program improvement" is a more appropriate term

Local Optimizations. Notes (II).

- Serious problem: what to do with pointers?
  - *t may change even if local variable t does not: *Aliasing*
  - Arrays are a special case (address calculation)
- What to do about globals?
- What to do about calls?
  - Not exactly jumps, because they (almost) always return.
  - Can modify variables used by caller
- Next: global optimizations