Instruction Selection

• How do we choose which instructions to emit from our IR?
• Greatly depends on how we design our IR.
  We see two examples in this lecture:
  ‣ Tree-structured IR — Tiling
  ‣ 3AC sequence — Peephole Matching

Canonical IR — Tiling

Example:  \( x = x + 1; \)

Tiles

• Capture compiler’s understanding of the instruction set
• Tile = sequence of instructions that update a fresh temporary (may need extra mov’s) and associated IR tree
• Each outgoing edge represents a temporary
CISC vs. RISC Tiling

Example Tiles for “+”

Computing a Tiling

- **Maximal Munch** — a greedy approach
  - Start at statement root
  - Find largest tile covering node, matching all children and outgoing edges
  - Invoke recursively on temporaries dangling from tile
  - Generate code for tile
    - (code for children will have been generated in recursive calls)

Timing Model

- **Idea**: associate cost with each tile
  - (proportional to # cycles to execute)
  - Sum of costs approximates execution time

Computing Optimum Tiling

- **Goal**: find minimum total cost tiling of tree
- **Algorithm**: for every node, find minimum total cost tiling of that node and sub-tree.
- **Lemma**: once minimum cost tiling for all descendants of a node is known, one can find minimum cost tiling of the node (sub-tree) by trying out all possible tiles matching the top
  - start from leaves, work upward to top node
Dynamic Programming

Example:

movl a(%ebp), t1
movl i(%ebp), t2
movl (t1, t2), a

Peephole Matching

- **Basic idea:** discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a small sliding window ("peephole") over code and search for improvement
- **Classic examples:**

```
* p := R1
R15 := * p

* p := R1
R10 := R8 * R7
Lc: goto Lc

R15 := * p
store followed by load
goto L10
```

Peephole Matching

- How to implement it?
- Modern instruction selectors break problem into three tasks: (Davidson, 1989)

**Expander**
- Turns IR code into a low-level IR (LLIR) such as RTL
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects

**Simplifier**
- Looks at LLIR through window and rewrites is
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window
Peephole Matching

Matcher
- Compares simplified LLIR against a library of patterns
- Picks low-cost pattern that captures effects
- Must preserve LLIR effects, may add new ones (e.g., set cc)
- Generates the assembly code output

Peephole Matching — Example

Original IR Code

<table>
<thead>
<tr>
<th>op</th>
<th>op2</th>
<th>result1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>y</td>
<td>t</td>
</tr>
<tr>
<td>-</td>
<td>x</td>
<td>t</td>
</tr>
</tbody>
</table>

r₁ = r₁₀
w = r₁¹

LLIR Code

r₁₀ := 2
r₁¹ := @y
r₁₂ := b * r₁₁
r₁₃ := * (r₁₂)
r₁₄ := r₁₀ * r₁₃
r₁₅ := @x
r₁₆ := b * r₁₅
r₁₇ := *(r₁₆)
r₁₈ := r₁₇ – r₁₄
r₁₉ := @w
r₂₀ := b * r₁₉
*(r₂₀) := r₁₈

Assembly (ILOC) Code

loadAI r₁₀, @y
mulI 2, r₁₃
loadAI r₁₂, @x
sub r₁₇, r₁₄
storeAI r₁₈, r₁₉
*(r₁₉) := r₁₈

Peephole Matching — Example

LIR Code

f₁₀ := 2
f₁¹ := @y
f₁₂ := b * f₁₁
f₁₃ := * (f₁₂)
f₁₄ := f₁₀ * f₁₃
f₁₅ := @x
f₁₆ := b * f₁₆
f₁₇ := *(f₁₆)
f₁₈ := f₁₇ – f₁₄
f₁₉ := @w
f₂₀ := b * f₁₉
*(f₂₀) := f₁₈

LLIR Code

i₀ := 2
i₁ := @y
i₂ := b * i₁₁
i₃ := * (i₂)
i₄ := i₀ * i₃
i₅ := @x
i₆ := b * i₆
i₇ := *(i₆)
i₈ := i₇ – i₄
i₉ := @w
i₂₀ := b * i₂₀
*(i₂₀) := i₁₈

Peephole Matching — Example

Steps of the Simplifier

LIR Code

f₁₀ := 2
f₁¹ := @y
f₁₂ := b * f₁₁
f₁₃ := * (f₁₂)
f₁₄ := f₁₀ * f₁₃
f₁₅ := @x
f₁₆ := b * f₁₆
f₁₇ := *(f₁₆)
f₁₈ := f₁₇ – f₁₄
f₁₉ := @w
f₂₀ := b * f₁₉
*(f₂₀) := f₁₈

LLIR Code

i₀ := 2
i₁ := @y
i₂ := b * i₁₁
i₃ := * (i₂)
i₄ := i₀ * i₃
i₅ := @x
i₆ := b * i₆
i₇ := *(i₆)
i₈ := i₇ – i₄
i₉ := @w
i₂₀ := b * i₂₀
*(i₂₀) := i₁₈

Match

loadAI r₁₀, @y
mulI 2, r₁₃
loadAI r₁₂, @x
sub r₁₇, r₁₄
storeAI r₁₈, r₁₉
*(r₁₉) := r₁₈

Peephole Matching — Example

LIR Code

f₁₀ := 2
f₁¹ := @y
f₁₂ := b * f₁₁
f₁₃ := * (f₁₂)
f₁₄ := f₁₀ * f₁₃
f₁₅ := @x
f₁₆ := b * f₁₆
f₁₇ := *(f₁₆)
f₁₈ := f₁₇ – f₁₄
f₁₉ := @w
f₂₀ := b * f₁₉
*(f₂₀) := f₁₈

LLIR Code

i₀ := 2
i₁ := @y
i₂ := b * i₁₁
i₃ := * (i₂)
i₄ := i₀ * i₃
i₅ := @x
i₆ := b * i₆
i₇ := *(i₆)
i₈ := i₇ – i₄
i₉ := @w
i₂₀ := b * i₂₀
*(i₂₀) := i₁₈

Match

loadAI r₁₀, @y
mulI 2, r₁₃
loadAI r₁₂, @x
sub r₁₇, r₁₄
storeAI r₁₈, r₁₉
*(r₁₉) := r₁₈

Peephole Matching — Example

Steps of the Expander

IR

R

LLIR

Simplifier

LLIR

Matcher

ASM

IR

R

LLIR

Simplifier

LLIR

Matcher

ASM
Peephole Matching — Example

LLIR Code

\[ r_{10} := 2 \]
\[ r_{12} := *(b + y) \]
\[ r_{14} := 2 \]
\[ r_{15} := *(r_{12}) \]
\[ r_{16} := b + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := *(r_{20}) \]
\[ r_{20} := b + r_{19} \]

Steps of the Simplifier
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := \ast (b_p + @y) \]
\[ r_{14} := 2 \times r_{13} \]
\[ r_{17} := \ast (b_p + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{19} := @w \]
\[ r_{20} := b_p + r_{19} \]
\[ \ast (r_{20}) := r_{18} \]

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := \ast (b_p + @y) \]
\[ r_{14} := \ast (b_p + @x) \]
\[ r_{17} := r_{13} - r_{13} - r_{13} \]
\[ r_{18} := b_p + @w \]
\[ \ast (r_{20}) := r_{18} \]
Peephole Matching — Example

\[ r_{13} := r_3 + r_1 \]
\[ r_{17} := r_{13} + r_1 \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ (r_3 + r_1) := r_{18} \]

LLIR Code

Making Peephole Matching Work

Details.
- LLIR is largely machine independent
- Target machine described as LLIR → ASM pattern
- Actual pattern matching
  - Use a hand-coded pattern matcher
  - Turn patterns into grammar & use LR parser
- Several important compilers use this technology
- It seems to produce good portable instruction selectors

Its key strength appears to be late low-level optimization.

Optimizations

- Improve program performance
- Must maintain original semantics
  - The observable behavior of the optimized program should be equivalent to that of the original program
- Typically cannot obtain "optimal program"
- Can optimize various aspects of program execution
  - Memory
  - Time
  - Code size
  - Power
  - ...

Why Optimize?

- Programmer may introduce inefficiencies
- Compiler may introduce inefficiencies
  - Make earlier compiler stages easier to deal with

Reminder: Data-flow Analysis

\[ x := 0 \]
\[ y := 1 \]
\[ \text{while } y < n \]
Constant Propagation

- Idea: if on every run, a variable can only take one value, it can be replaced with a constant.
- DFA over the following lattice:

```
+---+---+---+---+
| T | 1 | 2 | 3 |
+---+---+---+---+
```

Basic Optimizations

- Common-subexpression Elimination
  - Eliminate repeated computation inside a basic block (DAG representation)
  - Next time we will see how to do it beyond the boundaries of a single basic block
- Copy-propagation
  - Following an assignment \( x = y \), try to use \( "y" \) whenever \( "x" \) is used (if possible)
  - Potentially makes \( "x" \) into a dead variable, saving the assignment \( x = y \)

Common Subexpression Elimination

- Avoid recomputations
  - ...but be careful:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

```
a = b + c
b = a - d
c = b + c
d = b
```

Static Single-Assignment Form (SSA)

- Every assignment writes to a distinct variable
- Every variable is only assigned once

```
p = a + b
q = p + c
p = p - d
q = p + q
```

```
p1 = a + b
q1 = p1 + c
p2 = p1 - d
q2 = p2 + q1
```

SSA

- Branches?

```
x1 (c)
a = b1
x2
x1 = a + x2
```

```
x1 (c)
a = b1
x2
x1 = a + x2
```

\( \phi \) (phi) function combines different definitions
- \( \phi \) returns the value of \( x1 \) if control passes through the true branch and the value of \( x2 \) if it passed through the false branch
SSA — why should we care?

- Makes it easy to apply many optimizations
  - constant propagation, dead code elimination...

DAG Representation of Basic Blocks

Aliasing is a problem

- We don’t know whether i = j
- Same thing happens with pointers
- Have to handle it conservatively
  - Don’t know that it’s safe ⇒ must assume that it’s conflicting
- A major obstacle to effective optimization

Coming Up
Peephole Optimizations

- Optimizing long code sequences is hard
- A simple and efficient (but sub-optimal) alternative: peephole optimizations
  - Examine a small window ("peep hole") over the code
  - Identify local optimization opportunities
  - Rewrite code "in the window"
- Example
  - Local (single statement) algebraic simplifications

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

```
   f_10 := 2  # Simplified constant
   f_12 := @y # Alias
   f_13 := *(@y, f_12) # Original operation
   f_14 := @x # Alias
   f_15 := f_14 - f_13 # Simplification
   f_19 := *(@w, f_19) # Simplified assignment
```

```
   f_10 := 2  # Simplified constant
   f_12 := @y # Alias
   f_13 := *(@y, f_12) # Original operation
   f_14 := @x # Alias
   f_15 := f_14 - f_13 # Simplification
   f_19 := *(@w, f_19) # Simplified assignment
```
Peephole Matching — Example

LLIR Code

Steps of the Simplifier

r10 := 2
r12 := *(bp + @y)
F0 := r10, r12
F0 := 2 * r12
F0 := bp + r12
F0 := r12
F0 := @x
F0 := bp + r12
F0 := r12

r13 := *(bp + @y)
F0 := r13
F0 := 2 * r13
F0 := bp + r13
F0 := r13
F0 := @x
F0 := bp + r13
F0 := r13

r14 := 2 * r13
F0 := r14
F0 := bp + r14
F0 := r14
F0 := @x
F0 := bp + r14
F0 := r14

r15 := @x
F0 := r15
F0 := bp + r15
F0 := r15
F0 := @x
F0 := bp + r15
F0 := r15

r16 := bp + r15
F0 := r16
F0 := bp + r16
F0 := r16
F0 := @x
F0 := bp + r16
F0 := r16

r17 := *(r16)
F0 := r17
F0 := r17
F0 := r17
F0 := r17
F0 := r17
F0 := r17

r18 := r17
F0 := r18
F0 := r18
F0 := r18
F0 := r18
F0 := r18
F0 := r18

r19 := @w
F0 := r19
F0 := r19
F0 := r19
F0 := r19
F0 := r19
F0 := r19

r20 := bp + r19
F0 := r20
F0 := bp + r20
F0 := bp + r20
F0 := bp + r20
F0 := bp + r20
F0 := bp + r20

*(r20) := r18
*(r20) := r18
*(r20) := r18
*(r20) := r18
*(r20) := r18
*(r20) := r18
Eliminating Redundant Code

1. \( a = x \)
2. \( b = a \)
3. \( a = \text{someFunction}(a) \)
4. \( x = \text{someOtherFunction}(a, x) \)
5. \( \text{if } a > x \text{ goto (2)} \)

- Peephole optimizations are only performed within basic block boundaries

Code Motion

- Identify repeated computation inside a loop that depends only on values that are not modified inside the loop
- Move computation outside of the loop

```
while (x - 3 < y) {
    // … instructions that do not change x or t1
    t1 = x - 3;
    while (t1 < y) {
        // … instructions that do not change a or x
        t1 = t1 + 4;
    }
}
```

Induction Variables & Strength Reduction

1. \( i = 0 \)
2. \( t1 = i \times 4 \)
3. \( t2 = a[t1] \)
4. \( \text{if } t2 > 100 \text{ goto (19)} \)
5. \( \ldots \)
6. \( i = i + 1 \)
7. \( \text{goto (2)} \)
8. \( \ldots \)

- Identify loop counters and relationship with other variables

Program Transformations

- Code motion and strength reduction can improve performance, provided that they maintain program semantics.

- When does an optimization apply to a given program?
- How can this be done automatically?

Program Transformations

- How can a loop be identified?
  - For loop invariant code motion

- Easy to detect in high-level language

- Hard to detect in low-level language and CFG
  - Java bytecode
  - Control Flow Graph
Loop detection
- Loop in CFG has:
  - Loop header
  - Back edge
  - Set of nodes

D dominators
- Use dominators to identify loops
- Node d dominates node n
  - all path from the entry node to n go through d
- Every node dominates itself

Loop detection
- Header
  - dominates loop nodes
- Back edge
  - Target dominates source
- Loop identification
  - Back edge identification

Post dominators
- Node d post-dominates node n
  - all path from n to the exit node go through d
- d post-dominates n does not imply n dominates d
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4
- 1 dominates 4

- 1 does not dominate 4

Computing Dominators

- Using DFA (previous lecture)
- If 1 dominates 2, 3, 4 and 5
  - d dominates 6
- If 1 dominates 6
  - 1 dominates 2, 3, 4 and 5

Computing Dominators

- \text{Dom}(u)
  - Set of nodes that dominate \text{u}
- \text{Dom}(u_0) = \{u_0\}
- \text{Dom}(u) = \cap \{\text{Dom}(v) | v \in \text{pred}(u)} \cup \{u\}

Identify Loops

- Back edge
  - Edge t\rightarrow h s.t. h dominates t
- Loop of a back edge t\rightarrow h
  - h is the loop header
  - Loop nodes
    - all nodes that can reach t without going through t
Identify Loops: Algorithm

- Compute dominator relation
- Identify back edges
- Compute the loop nodes for each back edge

for each node h in dominator tree
for each node n for which there exists a back edge n ⃗→ h
  define the loop with header := h
  body := of all nodes reachable from n by a depth first search backwards from n that stops at h

Loop Preheader

- Needed by several optimizations
  - Loop invariant code motion
- Add additional code

Loop Optimization

- After identifying the loops
- We can use this info for loop optimization
- Loop invariant code motion
- Strength reduction of induction variable
- Induction variable elimination

Loop invariant code motion

```c
for (i=0; i < 10; ++i)
    a[i] = 10*i + x*x;

t = x*x;
for (i=0; i < 10; ++i)
    a[i] = 10*i + t;
```
Loop invariant computation

- \( a := b \land c \) is loop invariant if \( b \) and \( c \):
  - Are constant, or
  - Have only definitions outside the loop, or
  - Have only one definition (each) that are themselves loop invariant
- Reaching definition analysis can be used (DFA)

Loop invariant computation

\[
\begin{align*}
\text{INV} & = \emptyset \\
\text{repeat} & \\
& \text{for each instruction } I \text{ in loop such that } I \notin \text{INV} \\
& \text{if operands are constants, or operands have no definitions inside the loop, or} \\
& \text{operands have exactly one definition } d \in \text{INV} \\
& \text{then } \text{INV} = \text{INV} \cup \{I\} \\
& \text{until no changes in INV}
\end{align*}
\]

Code motion

- Assume \( a := b \land c \) is loop invariant
- When can we hoist it out of the loop?
- Where to?

Valid code motion

Move \( d: a := b \land c \) to pre header is valid when:
1. \( d \) dominates all loop exits where \( a \) is live
2. \( d \) is the only definition of \( a \) in the loop
3. All uses of \( a \) in loop can only be reached from \( d \)

Valid code motion

1. \( d \) dominates all loop exits where \( a \) is live

\[
\begin{align*}
a & = 0; \\
\text{for } (i=0; i<10; ++i) & \\
& \text{if } (f(i)) a = x^*; \text{ else break; } \quad \times \\
b & = a;
\end{align*}
\]

Use dominator relation to check whether each loop exit is dominated by \( d \)
Valid code motion

2. d is the only definition of a in the loop
   
   ```
   for (i=0; i<10; ++i)
   if (f(i)) a = x*x;
   else a = 0;
   ```

   ✓
   
   Scan loop body for any other definitions of a

Valid code motion

3. All uses of a in loop can only be reached from d
   
   ```
   a = 0;
   for (i=0; i<10; ++i)
   if (f(i)) a = x*x;
   else buf[i] = a;
   ```

   ✓
   
   Apply reaching definitions analysis and check each use of a for any definitions of a other than d

Summary

- Optimizations
  ✓ Not real "optimal" code
  ✓ But code that is "better" in some respect
  ✓ Often focuses on runtime
- Techniques
  ✓ Local: data-flow DAG representation, peephole
  ✓ Global: (function-level) Code motion, strength reduction — based on running a DFA first

Next