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

\[
\begin{align*}
x & := 0 \\
y & := 1
\end{align*}
\]

while \( y < n \) do

\[
\begin{align*}
x & := x + 2 \\
y & := y + 1
\end{align*}
\]

\[
\begin{align*}
x & := n - 1 \\
y & := n - 1
\end{align*}
\]

print \( x \) print \( y \)
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:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
```

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:

```plaintext
\[
\begin{align*}
  a &= b + c \\
  b &= a - d \\
  c &= b + c \\
  d &= a - d
\end{align*}
\]
```

```plaintext
\[
\begin{align*}
  a &= b + c \\
  b &= a - d \\
  c &= b + c \\
  d &= b
\end{align*}
\]
```

Static Single-Assignment Form (SSA)

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

```plaintext
\[
\begin{align*}
  p &= a + b \\
  q &= p + c \\
  r &= p - d \\
  s &= p + q
\end{align*}
\]
```

```plaintext
\[
\begin{align*}
  p1 &= a + b \\
  q1 &= p1 + c \\
  r1 &= q1 - d \\
  s1 &= p1 + q1
\end{align*}
\]
```

SSA

- Branches?
  - \( \phi \) (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

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 algebraic simplifications
Peephole Optimizations

- Algebraic simplification
  - a = b + c
  - b = d – d
  - c = c * b
  - e = a + c

- Flow of control
  - 1: if d == 1 goto 3
  - 2: goto 4
  - 3: print(x)
  - 4: e = a + c

- Peephole optimizations are only performed within basic block boundaries

Eliminating Redundant Code

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

- Peephole optimizations are only performed within basic block boundaries

Live Variables

1. x := 2
2. y := 4
3. x := 1
4. if y > x
5. then z := y
6. else z := y * y;
7. x := z

- Out(1) = ∅
- Out(2) = in(1) = ∅
- Out(3) = in(2) = { y }
- In(3) = { x,y }
- Out(4) = { x,y }
- In(4) = { y }
- Out(5) = { y }
- In(5) = { z }
- Out(6) = { y }
Eliminating Redundant Code

- Any assignment after which the assigned variable is dead, is redundant

1: \( x := 2 \)
2: \( y := 4 \)
3: \( x := 1 \)
4: if \( y \leq x \) goto 7
5: \( z := y \)
6: goto 8
7: \( z := y \times y \)
8: \( x := z \)

• Any assignment after which the assigned variable is dead, is redundant

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

Induction Variables & Strength Reduction

- 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

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

Dominators

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

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

### Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4
- 1 does not dominate 4

### Computing Dominators

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

### Computing Dominators

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

### Computing Dominators

- $\text{Dom}(u_0) = \{ u_0 \}$
- $\text{Dom}(u) = \cap \{ \text{Dom}(v) | v \in \text{pred}(u) \} \cup \{ u \}$
Identify Loops

- Back edge
  - Edge \( t \prec h \) s.t. \( h \) dominates \( t \)
- Loop of a back edge \( t \prec h \)
  - \( h \) is the loop header
  - Loop nodes
    - all nodes that can reach \( t \) without going through \( h \)

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 \( n \prec h \)
    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
- Additional code is placed there.

Loop Optimization

- Step 1: identify loops (header, pre-header)
- Apply loop optimizations:
  - Loop invariant code motion
  - Strength reduction of induction variable
  - Induction variable elimination
**Code Motion — Loop Invariant**

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

- An expression is **loop invariant** if it does not change throughout the execution of the loop.
  - Can be hoisted — computed only once

---

**Loop Invariant Computation**

- `a := b \& 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**

```plaintext
INV := ∅
repeat
  for each definition \( \ell \) in loop such that \( I \notin INV \)
  if each operand in \( \ell \):
    is constant, or
    has no definitions inside the loop, or
    has exactly one definition \( d \in INV \)
  then \( INV := INV \cup \{ \ell \} \)
until no changes in INV
```

---

**Code Motion — Loop Invariant**

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

---

**Valid Code Motion**

Move `d: a := b \& 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

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

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

\[
\text{for } (i=0; i<10; ++i)
\]
\[
\text{if } (f(i)) \ a = x^x; \quad \times
\]
\[
\text{else } a = 0;
\]

Scan loop body for any other definitions of \(a\)

Valid code motion

3. All uses of \(a\) in loop can only be reached from \(d\)

\[
a = 0;
\]
\[
\text{for } (i=0; i<10; ++i)
\]
\[
\text{if } (f(i)) \Rightarrow x^x; \quad \times
\]
\[
\text{else } \text{buf}[i] = a;
\]

Apply reaching definitions analysis and check each use of \(a\) for any definitions of \(a\) other than \(d\)

Valid Code Motion

\[
d: a = b \lor c
\]

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 \(d\)

\[
\text{for } (i=0; i<10; ++i)
\]
\[
| e[i] | 40081i + x^x; \quad \text{for } (i=0; i<10; ++i)
\]
\[
t_2 := x^x
\]
\[
t_1 := 10^i
\]
\[
a[i] := t_1 + t_2
\]

Strength Reduction

- If an expression \(e\) is not loop invariant, code motion may still be applied:
  - Compute \(e\) once in loop pre-header
  - Whenever any variable occurring in \(e\) is modified, update computed value

- When is this beneficial?
  - If the cost of updating \(e\) is smaller ("weaker") than recomputing \(e\) anew
Strength Reduction

- Differencing rules:

\[
\begin{align*}
t_1 & := c \ast i \\
\vdots \\
j & := i + 1 \\
t_1 & := t_1 \\
\vdots \\
j & := i + 1 \\
t_1 & := c \ast j \\
\vdots \\
i & := i + 1 \\
t_1 & := t_1 \\
\vdots \\
j & := j + 1 \\
t_1 & := t_1 + j \\
\vdots \\
j & := j + 1 \\
t_1 & := t_1 + i \\
\vdots \\
j & := j + 1 \\
t_1 & := t_1 + i \\
\end{align*}
\]

(c is loop invariant)

Summary

- Optimizations
  - Not real "optimal" code
  - But code that is "better" in some respect
  - Often focuses on runtime

- Techniques
  - Local: expressions in basic block, peephole
  - Global: (function-level) Code motion, strength reduction — based on running a DFA first

Next