CS 536 Announcements for Monday, May 1, 2023

Last Time
- wrap up code generation
- dot-access
- introduce control flow graphs
- control-flow constructs and code generation

Today
- optimization overview
- peephole optimization
- loop optimizations
- copy propagation

Next Time
- wrap up optimization
- wrap up course / review

Recall example from last time

MIPS code outline:

```
lw $t0, addr_a
push $t0

lw $t0, addr_b
push $t0

pop $t1
pop $t0
sgt $t0, $t0, $t1
push $t0

pop $t0
beq $t0, FALSE, falseLabel
.
. # code for true branch
.
b doneIfLabel

falseLabel:
.
. # code for false branch
.
doneIfLabel:
```
Optimization Overview

Goals
Informally: Produce "better" code that does the "same thing" as the original code.
What are we trying to accomplish?
• faster code
• fewer instructions
• lower power
• smaller footprint
• bug resilience?

Safety guarantee
Informally: Don't change the program's output (observable behavior)
• the same input produces the same output
• if the original program produces an error on a given input, so will the transformed code
• if the original program does not produce an error on a given input, neither will the transformed code

However... There's no perfect way to check equivalence of two arbitrary programs
• if there was, we could use it to solve the halting problem
• we'll attempt to perform behavior-preserving transformations
Program Analysis

A perspective on optimization
- recognize some behavior in a program
- replace it with a "better" version

However, halting problem keeps arising:
- we can only use approximate algorithms to recognize behavior

Two properties of program-analysis/behavior detection algorithms
- **soundness**: all results that are output are valid
- **completeness**: all results that are valid are output

Analysis algorithms with these properties are mutually exclusive:
- if an algorithm was sound and complete, it would either:
  - solve the halting problem, or
  - detect a trivial property

Optimization Overview (cont.)

We want our optimizations to be **sound** transformations
- they are always valid
- but some opportunities for applying a transformation will be missed

Our techniques
- can detect many _practical_ instances of the behavior
- won't cause any harm
- but we still want to consider efficiency

Peephole optimization
- naïve code generator errs on the side of correctness over efficiency
- use pattern-matching to find the most obvious places where code can be improved
- look at only a few instructions at a time
Peephole optimization

<table>
<thead>
<tr>
<th>What can be optimized</th>
<th>Replaced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>push followed by pop</td>
<td></td>
</tr>
<tr>
<td>pop followed by push</td>
<td></td>
</tr>
<tr>
<td>branch to next instruction</td>
<td></td>
</tr>
<tr>
<td>jump to a jump</td>
<td></td>
</tr>
<tr>
<td>jump around a jump</td>
<td></td>
</tr>
<tr>
<td>store followed by load</td>
<td></td>
</tr>
<tr>
<td>load followed by store</td>
<td></td>
</tr>
<tr>
<td>useless operations</td>
<td></td>
</tr>
<tr>
<td>multiplication by 2</td>
<td></td>
</tr>
</tbody>
</table>

Do multiple passes?
Loop-Invariant Code Motion (LICM)

Idea: Don't duplicate effort in a loop

Goal: Pull code out of the loop ("loop hoisting")

Important because of "hot spots"
- most execution time due to small regions of deeply-nested loops

Example

```c
for (i=0; i<100; i++) {
    for (j=0; j<100; j++) {
        for (k=0; k<100; k++) {
            A[i][j][k] = i*j*k;
        }
    }
}
```

becomes

```c
for (i=0; i<100; i++) {
    for (j=0; j<100; j++) {
        temp = i*j;
        for (k=0; k<100; k++) {
            A[i][j][k] = temp*k;
        }
    }
}
```

Suppose $A$ is on the stack.

To compute the address of $A[i][j][k]$:

$$FP - offset\_of\_A[0][0][0] + (i*10000*4) + (j*100*4) + (k*4)$$
Loop-Invariant Code Motion (cont.)

When should we do LICM?
- at IR level, more candidate operations
- assembly might be too low-level
  - need guarantee that the loop is natural

How should we do LICM? Factors to consider
- safety – is the transformation semantics-preserving?

- profitability – is there any advantage to moving the instruction?

Other Loop Optimizations

Strength reduction in for-loops
- replace multiplications with additions

Loop unrolling
- for a loop with a small, constant number of iterations, may actually take less time to execute by just placing every copy of the loop body in sequence
- may also consider doing multiple iterations within the body

Loop fusion
- merge 2 sequential, independent loops into a single loop body
Copy propagation

copy statement  
\[ x = y; \]

*definition* of \( x \)

*use* of \( y \)

**Idea:** Suppose we are at *use* \( U \) of \( x \) and a *definition* \( D \) of \( x \) (of the form \( x = y \)) reaches \( U \)

- If
  1) no other definition of \( x \) reaches \( U \) and
  2) \( y \) does not change between \( D \) and \( U \)
- then we can replace the use of \( x \) at \( U \) with \( y \)

**Example**

\[
\begin{align*}
x & = 3; \\
y & = 5; \\
p & = x; \\
\text{if} \ (w \times x > 9) \ {\{ \\
 & \quad x = 4; \\
 & \quad z = x + w \times y; \\
\} \\
\text{else} \ {\{ \\
 & \quad z = 2 \times y + x; \\
\} \\
q & = 5 \times p; \\
s & = z + x; \\
t & = s + y;
\end{align*}
\]
Copy propagation (cont.)

How is this an optimization?

- can create useless code (which can then be removed)

- can create improved code

- constant folding

- if done before other optimizations, can improve results