CS 536 Announcements for Monday, April 22, 2024

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

Today
- optimization overview
- peephole optimization
- loop optimizations

Next Time
- copy propagation

Recall example from last time

MIPS code outline:

```mips
lw $t0, addr_a
push $t0

lw $t0, addr_b
push $t0

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

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

falseLabel:
.
  # code for false branch
  .

doneIfLabel:
```

move $t1, $t0
unnecessary but we don't see this until after code generation
Optimization Overview

Goals
Informally: Produce "better" code that does the "same thing" as the original code.
What are we trying to accomplish?
- faster \text{code}
- fewer \text{instructions}
- lower \text{power}
- smaller \text{footprint}
- buy \text{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

Does order need to be preserved?
- when output is generated
- different order of ops in floating-point arithmetic may produced different results

Aside: evaluating polynomials: $A x^7 + B x^6 + C x^5 + \ldots$ can be evaluated as

\[
\left(\left(\left(A x + B\right) x + C\right) x + D\right) + \ldots
\]

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

- done after code is generated
## 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>nothing</td>
</tr>
<tr>
<td>4 MIPS instrs</td>
<td>move $t1, $t0</td>
</tr>
<tr>
<td>Note: can't do optimization if have a label associated with pop</td>
<td>load value from top of stack directly into $t0</td>
</tr>
<tr>
<td>pop followed by push</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>branch to next instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>jump to a jump</td>
<td></td>
</tr>
<tr>
<td>extra conditions are required</td>
<td></td>
</tr>
<tr>
<td>jump around a jump</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Peephole optimization (cont.)

<table>
<thead>
<tr>
<th>What can be optimized</th>
<th>Replaced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>store followed by load</td>
<td><code>sw $t0, addr</code> <code>lw $t0, addr</code></td>
</tr>
<tr>
<td>same register, same address</td>
<td></td>
</tr>
<tr>
<td>load followed by store</td>
<td><code>lw $t0, addr</code> <code>sw $t0, addr</code></td>
</tr>
<tr>
<td>same register, same address</td>
<td></td>
</tr>
<tr>
<td>useless operations</td>
<td><code>add $t0, $t0, 0</code></td>
</tr>
<tr>
<td><code>add $t0, $t1, 0</code></td>
<td><code>nothing</code> <code>move $t0, $t1</code></td>
</tr>
<tr>
<td>multiply by 1</td>
<td>same as for add -</td>
</tr>
<tr>
<td>L-&gt; in MIPS: multiply, then mflo (move from lo) ie 2 instrs</td>
<td></td>
</tr>
<tr>
<td>multiplication by 2</td>
<td><code>shift-left (faster)</code></td>
</tr>
<tr>
<td>some assembly langs have increment command — could use to replace</td>
<td></td>
</tr>
<tr>
<td>(MIPS doesn’t)</td>
<td>add by 1</td>
</tr>
</tbody>
</table>

**Do multiple passes?**

- `pop $t0`
- `add $t0, $t0, 0 -> remove on 1st pass`
- `push $t0` 2nd pass
- `lw $t0, 4($sp)`

**Fixed # of passes?**

Or run passes until no more changes to the code?
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]:

$$\text{FP} - \text{offset of } A[0][0][0] + (i*10000*4) + (j*100*4) + (k*4)$$
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
  - no jumps into middle of the loop

How should we do LICM? Factors to consider
- safety – is the transformation semantics-preserving?
  - make sure operation is truly loop-invariant
  - ordering of events is preserved
- profitability – is there any advantage to moving the instruction?
  - may end up – moving instructions that are never (or rarely) executed
  - performing more intermediate computation than necessary

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
- fewer jumps

Loop fusion
- merge 2 sequential, independent loops into a single loop body
- fewer jumps