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:

```
     lw $t0, addr_a
     push $t0

     lw $t0, addr_b
     push $t0        } move $t1, $t0

     pop $t1
     pop $t0
     sgt $t0, $t0, $t1
     push $t0        } unnecessary but we don't see this until after code generation

.L35: pop $t0
     beq $t0, FALSE, falseLabel
     .
     . # code for true branch
     .
     b doneIfLabel

falseLabel:
     .
     . # code for false branch
     .

doneIfLabel:
```

what if had ↓

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

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: $Ax^7 + Bx^6 + Cx^5 + \ldots$ $O(n)$ adds, $O(n^2)$ mults, n = deg of poly

can be evaluated as

$$(((Ax+B)x+C)x+D)+\ldots$$

$O(n)$ adds, $O(n)$ mults

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

What can be optimized		**Replaced with**
push followed by pop 4 MIPS instrs Note: can't do optimization if have a label associated with pop	push $t0 pop $t0 (same reg)	nothing
	push $t0 pop $t1 (diff reg)	move $t1, $t0
pop followed by push	pop $t0 push $t0 (same reg)	load value from top of stack directly into $t0
branch to next instruction	b label label:	label:
* jump to a jump	b L1 ⋮ L1: b L2	b L2 ... L1: b L2
extra conditions are required		
* jump around a jump	beq $t0,$t1,L1 b L2 L1:	bne $t0,$t1,L2 L1:

Peephole optimization (cont.)

What can be optimized		Replaced with
store followed by load same register, same address	sw $t0, addr lw $t0, addr	sw $t0, addr
load followed by store same register, same address	lw $t0, addr sw $t0, addr	lw $t0, addr
useless operations add 0	add $t0, $t0, 0 add $t0, $t1, 0	nothing move $t0, $t1

multiply by 1 — same as for add —
↳ in MIPS: multiply, then mflo (move from lo) ie 2 instrs

multiplication by 2 shift-left (faster)

some assembly langs have increment command — could use to replace add by 1
(MIPS doesn't)

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

```
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;
        }
    }
}
```

sub expression is invariant with respect to innermost loop

becomes

```
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)
```

```
tmp0 = FP-offset_of_A[0][0][0]
for (i=0; ...
  tmp1 = tmp0 + i*40000;
  for (j=0; ...
    tmp2 = tmp1 + j*400;
    temp = i*j;
    for (k=0; ...
      T0 = temp*k;
      T1 = tmp2 + k*4;
      store T0, 0(T1)
```

Loop-Invariant Code Motion (cont.)

When should we do LICM?

- at IR level, more candidate operations
- assemby 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 – no jumps
- 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