Optimization
Roadmap

Last time:
– CodeGen for the remainder of AST nodes
– Introduced the control-flow graph

This time:
– Optimization Overview
– Discuss a couple of optimizations
  • Review CFGs
OPTIMIZATION OVERVIEW
Optimization Goals

What are we trying to accomplish?
– Traditionally, speed
– Lower power
– Smaller footprint
– Bug resilience?
The fewer instructions the better
Optimization Guarantees

Informally: Don’t change the program’s output

– We may relax this to “Don’t change the program’s output on good input”

– This can actually be really hard to do
Optimization Difficulties

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

Constantly plagued by the halting problem
– We can only use approximate algorithms to recognize behavior
Program 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 necessarily mutually exclusive
– If an algorithm was sound *and* complete, it would either:
  1. Solve the halting program
  2. Detect a trivial property
Back to Optimization

We want our optimizations to be *sound* transformations

– In other words, they are always valid, but will miss some opportunities for applying a transformation
You May Be Thinking ...

I’m sad because this makes optimization seem pretty limited

Cheer up! Our optimization techniques can detect many *practical* instances of the behavior
Now You May Be Thinking ...

I’m happy because I’m guaranteed that my optimization won’t do any harm

Settle down! Our optimization still needs to be efficient
Or Maybe You Are Thinking ...

I don’t know how to feel about any of this without understanding how often it comes up.
What Can We Do?

We can pick some low-hanging fruit
EXAMPLE OPTIMIZATIONS
Peephole Optimization

A naïve code generator tends to emit some silly code
– Errs on the side of correctness over efficiency
Use pattern-matching to find the most obvious problems
Consider the following sequence of instructions:

\[
\begin{align*}
\text{push} & \quad \begin{cases}
    \text{sw} & \quad t0 \ 0(\text{sp}) \\
    \text{subu} & \quad sp \ sp \ 4 \\
    \text{lw} & \quad t0 \ 4(\text{sp}) \\
    \text{addu} & \quad sp \ sp \ 4
\end{cases}
\end{align*}
\]

We’d like to remove this sequence...

– Is it sound to do so?
– Maybe not!

\[
\begin{align*}
\text{sw} & \quad t0 \ -12(\text{sp})
\end{align*}
\]
Review: The CFG

Program as a flowchart
Nodes are “basic blocks”
Edges are control transfers
– Fall-through
– Jump
– *Maybe* function calls
CFG for Optimization

We can limit our peephole optimizations to *intra-block* analysis

– This approach ensures, by definition, that no jumps will intrude on the sequence

We will assume for the rest of our peephole optimizations that instruction sequences are in one block
Peephole Examples

Called “peephole” optimization because we are conceptually sliding a small window over the code, looking for small patterns
Outline

Four different optimizations
– Peephole optimization
– Loop-Invariant Code Motion
– For-loop strength reduction
– Copy propagation

Performed after machine code generation
Performed before machine code generation
Peephole Optimization 1

Remove no-op sequences
- Push followed by pop
- Add/sub 0
- Mul/div 1

\[
\begin{align*}
\text{push} & \quad \{ \begin{array}{c}
\text{sw} \quad t0 \quad 0(\text{sp}) \\
\text{subu} \quad \text{sp} \quad \text{sp} \quad 4 \\
\text{lw} \quad t0 \quad 4(\text{sp}) \\
\text{addu} \quad \text{sp} \quad \text{sp} \quad 4
\end{array} \} \\
\text{pop} & \quad \{ \begin{array}{c}
\text{addu} \quad \text{t1} \quad \text{t1} \quad 0 \\
\text{mul} \quad \text{t2} \quad \text{t2} \quad 1
\end{array} \}
\end{align*}
\]
Peephole Optimization 2

Simplify sequences
– Ex. Store then load
– Strength reduction

```
sw $t0 -8($fp)
lw $t0 -8($fp)

mul $t1 $t1 2
add $t2 $t2 1
```

Useless instruction

shift-left $t1

inc $t2
Peephole Optimization 3

Jump to next instruction

Remove this instruction
Loop Invariant Code Motion (LICM)

Don’t duplicate effort in a loop!

Goal
– Pull code out of the loop
– “Loop hoisting”

Important due to “hot spots”
– Most execution time due to small regions of deeply-nested loops
LICM 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
        }
    }
}
```

Sub-expression *invariant* with respect to the innermost loop

```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
        }
    }
}
```
LICM Example

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)
LICM: When Should We Do It?

In the previous example, showed LICM on source code
At IR level, more candidate operations
Assembly might be too low-level
- Need a guarantee that the loop is *natural*
  • No jumps into the loop

```c
tmp0 = FP - offsetA
for (i=0; i<100; i++){
    tmp1 = tmp0 + i*40000
    for (j=0; j<100; j++){
        tmp2 = tmp1 + j*400
        temp = i*j
        for (k=0; k<100; k++){
            T0 = temp * k
            T1 = tmp2 + k*4
            store T0, 0(T1)
        }
    }
}
LICM: How Should We Do It?

Two factors, which really apply to all optimizations in general:

– Safety
  • Is the transformation semantics-preserving?
    – Make sure the operation is truly loop-invariant
    – Make sure ordering of events is preserved

– Profitability
  • Is there any advantage to moving the instruction?
    – May end up moving instructions that are never executed
    – May end up performing more intermediate computation than necessary
Other Loop Optimizations

Loop unrolling

– For a loop with a small, constant number of iterations, we may actually save time by just placing every copy of the loop body in sequence (no jumps)

– May also consider doing multiple iterations within the body

Loop fusion

– Merge two sequential, independent loops into a single loop body (fewer jumps)
Jump Optimizations

Disclaimer: Require some extra conditions

Jump around jump

```
beq $t0,$t1,Lab1
  j    Lab2
Lab1:  ...
  ...
Lab2:  ...
```

Jump to jump

```
j    Lab1
Lab1:  j  Lab2
  ...
Lab1:  j  Lab2
  ...
Lab2:  ...
Lab2:  ...
```
Intraprocedural Analysis

The past two optimizations had some caveats
– There may be a jump into your eliminated code
We’d like to introduce a control-flow concept beyond basic blocks:
– Guarantee that block1 must be executed to get to block2
  • This goes by a pretty boring name

beq $t0 $t1 Lab1
j Lab2
Lab1: ...
Lab2: ...

...
Dominators and Post-Dominators

We say that block A dominates block B if A must be executed before B is executed. We say that block A postdominates block B if A must be executed after B.
Semantics Preserving

Do we really need semantics-preserving optimizations?
Are there examples where we don’t?

if the unoptimized code performs output then has a runtime error, is it valid for the optimized code to simply have a runtime error?
Summary

Today

• Saw the basics of optimizations
• Soundness vs. completeness
• Peephole and simple optimizations

Next time

• More optimizations
• Basics of static analysis