

Special topics

CompCert: a formally verified compiler

What would you like to hear about?

Optimization

1. Goal: the fewer instruction, the better.
2. Optimization difficulty: preserve the program behavior.
3. Soundness v.s. completeness: mutually exclusive. Refer to page 10 for detail.
4. Optimization is generally hard, but some easy tricks are good enough.
In CFG, the program is a flowchart. Nodes are "basic block"s and edges are control transfers like fallthrough, jump and function calls. **Intra-block** is easier to deal with.

Only peephole is done at MC levels, others are at IR or AST level.

- Peephole

Push followed by pop; Add/sub 0; Mult/div 1; Mult/div -> shift, Add -> Inc; Unneeded jump

- Loop-Invariant Code Motion

- For-loop strength reduction

Loop unrolling, loop fusion.

- Copy propagation

Jump around jump; jump to jump

5. Domination

Course evaluations

Please complete them

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

Let P the original program and P' is the optimized result. There's no perfect way to check if P and P' are equivalent.

Instead, we try to make our optimization process behavior-preserving.

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 terms in program analysis / behavior detection:

Soundness: the output set is a subset of the valid set.
Completeness: the valid set is a subset of the output set.
When you get both, the valid set is the same as the output set.

- Soundness: All results that are output are valid
- Completeness: All results that are valid are output

These terms are necessarily mutually exclusive

- If an algorithm was sound *and* complete, it would either:

1. Solve the halting problem
2. Detect a trivial property

Output: output of the optimizer, a program.
Valid: the output program is equivalent to the original

Soundness: the optimizer always produces program that are equivalent to original program.
Completeness: the optimizer does all possible optimizations.

Back to optimization

We want our optimizations to be *sound* transformations

- In other words, they are always valid, but will miss some behaviors



You may be thinking...

I'm sad because this makes optimization seem pretty limited



Cheer up! Our optimization may be able to 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 output some silly code

- Err on the side of correctness over efficiency

Pattern-match the most obvious problems

CFG for program analysis

Consider the following sequence of instructions:

If there can be a jump here and these four instructions are not a basic block, then we cannot simply remove it.

push {
sw \$t0 0(\$sp)
subu \$sp \$sp 4
pop {
lw \$t0 4(\$sp)
addu \$sp \$sp 4

If it's a "basic block" then you can safely remove it?

We'd like to remove this sequence...

- Is it sound to do so?
- Maybe not!

Review: the CFG

Program as a flowchart

Nodes are “Basic Blocks”

Edges are control transfers

- Fallthrough
- Jump
- *Maybe* function calls

CFG for optimization

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

- This 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

On basic blocks in MC.

Performed *before* machine code generation

On IR or AST.

Peephole optimization 1

Remove no-op sequences

- Push followed by pop

```
push { sw    $t0 0($sp)
      subu   $sp $sp 4
pop   { lw    $t0 4($sp)
      addu   $sp $sp 4
```

- Add/sub 0

```
addu $t1 $t1 0
```

- Mul/div 1

```
mul $t2 $t2 1
```

Peephole optimization 2

Simplify sequences

- Ex. Store then load
- Strength reduction

The diagram illustrates three examples of peephole optimization. Each example consists of assembly code followed by a blue annotation with an arrow pointing to the relevant instruction(s).

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

Useless instruction


```
mul   $t1 $t1 2
```

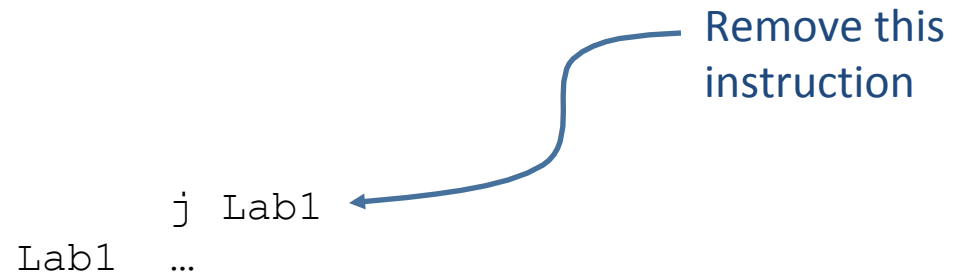
shift-left \$t1


```
add   $t2 $t2 1
```

inc \$t2

Peephole optimization 3

Jump to next instruction



Loop invariant code motion

Loop Invariant Code Motion (LICM)

- Don't duplicate effort in a loop

Goal

- Pull code out of the loop
- “Loop hoisting”

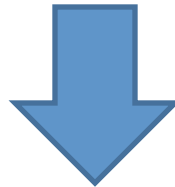
Important due to “hot spots”

- Most execution time due to small regions of deeply-nested loops

LICM example

```
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
Innermost loop



This is done at the IR/source
code level.

```
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: 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

```
tmp0 = FP + offsetA
for (i=0; i<100; i++){
    tmp1 = tmp0 - i*40000
    for (j=0; j<100; j++){
        tmp2 = ind2
        tmp3 = i*j
        for (k=0; k<100; k++){
            T0 = tmp3 * k
            T1 = tmp2 - k*4
            store T0, 0(T1)
        }
    }
}
```

LICM: How should we do it?

Two factors, which really generalize to optimization:

- 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

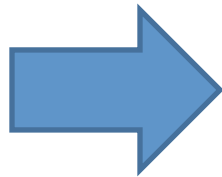
- Merge two sequential, independent loops into a single loop body (fewer jumps)

Jump optimizations

Disclaimer: Require some extra conditions

Jump around jump

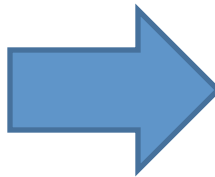
```
        beq $t0,$t1,Lab1
        j    Lab2
Lab1:    ...
        ...
Lab2:    ...
```



```
        bne $t0,$t1,Lab2
Lab1:    ...
        ...
Lab2:    ...
```

Jump to jump

```
        j    Lab1
        ...
Lab1:    j    Lab2
        ...
Lab2:    ...
```



```
        j    Lab2
        ...
Lab1:    j    Lab2
        ...
Lab2:    ...
```

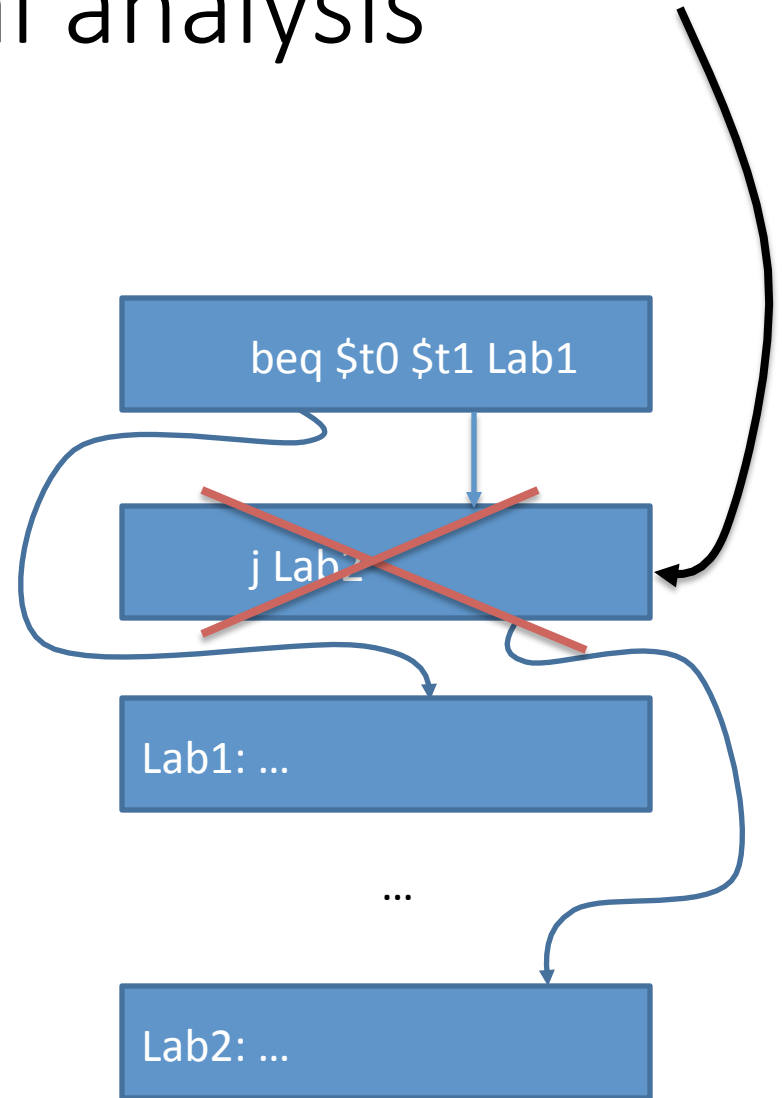
Intraprocedural analysis

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 in order to get to block2
 - This goes by a pretty boring name



Domination

Dominators and post-dominators

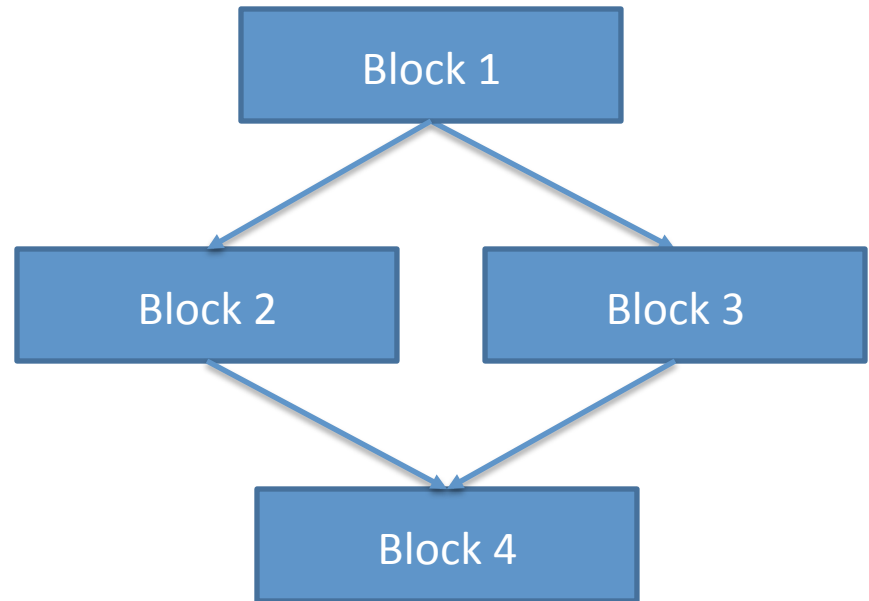
We say that block A dominates block B if A must be executed before B is executed

We say that block A postdominates block B if A must be executed after B

Block 1 dominates block 2, 3, and 4.
Neither 2 nor 3 dominates 4.

4 post-dominates 1, 2, and 3.
Neither 2 nor 3 post-dominates 1.

Control Flow Graph



Semantics preserving

Do we really need semantics preserving optimizations?

Are there examples where we don't?

In summary

Today

- Saw the basics of optimizations
- Soundness vs completeness
- Peephole and simple optimizations

Next time

- Wrap up optimizations
- Basics of static analysis