Optimization

- Identify inefficiencies in target or intermediate code
- Replace with equivalent but “better” sequences
- "Optimize" is a lie. "Usually improve" is more honest.

Example

```plaintext
x := a[i] + b[2];
c[i] := x - 5;
```

```
t1 := *(fp + ioffset) // i
12 := i * 4
13 := fp + 12
14 := *(fp + aoffset) // a[j]
i5 := 2
16 := i5 * 4
17 := fp + i6
18 := *(t3 + boffset) // b[2]
t9 := t4 + t8
*(fp + xoffset) := t9
```

Kinds of optimizations

- Scope of analysis is central to what optimizations can be performed. A larger scope may expose better optimizations, but is more complex
- Peephole: look at adjacent instructions
- Local: look at straight-line sequences of instructions
- Global (intra-procedural): look at whole procedure
- Interprocedural: look across procedures

Peephole

- After codegen, look at a few adjacent instructions
- Try to replace them with something better
- If you have
  ```plaintext
  sw $8,12($fp)
lw $12,12($fp)
  ```
- You can replace it with
  ```plaintext
  sw $8,12($fp)
  mv $12,$8
  ```
Peephole examples: 68k

If you have

\[
\begin{align*}
\text{sub } \text{sp},4,\text{sp} \\
\text{mov } \text{r1},0(\text{sp})
\end{align*}
\]

Replace it with

\[
\begin{align*}
\text{mov } \text{r1},-(\text{sp}) \\
\text{inc } 12(\text{fp})
\end{align*}
\]

Peephole optimization of jumps

- Eliminate jumps to jumps
- Conditional branch over unconditional branch
- “Adjacent instructions” means “adjacent in control flow”

Peephole optimization of jumps

- If \( a < b \) then
  - If \( c < d \) then
    - # do nothing
  - else
    - \text{stmt1;}
    - \text{end;}
- else
  - \text{stmt2;}
  - \text{end;}

How to do peephole opts

- Could be done at IR and/or target level
- Catalog of specific code rewrite templates
- Scan code with moving window looking for matches

Peephole summary

- You could consider peephole optimization as increasing the sophistication of instruction selection
- Relatively easy to do
- Relatively easy to extend
- Relatively easy to ensure correctness
- Relatively high payoff

Algebraic simplifications by peephole or codegen

- “constant folding” and “strength reduction” are common names for this kind of optimization
  - \( z := 3 + 4 \)
  - \( z := x + 0 \)
  - \( z := x \times 1 \)
  - \( z := x \times 2 \)
  - \( z := x \times 8 \)
  - \( z := x \times 8 \)
  - \( z := (x + y) - y \)

Local optimization

- Analysis and optimizations within a basic block
  - A basic block is a straight-line sequence of statements with no control flow into or out of the middle of the sequence
  - Local optimizations are more powerful than peephole (e.g., block may be longer than peephole window)
    - Not too hard to implement
    - Can be machine-independent, if done on intermediate code
Local constant propagation (aka "constant folding")

- If a constant is assigned to a variable, replace downstream uses of the variable with the constant
- If all operands are const, replace with result
- May enable further constant folding

Example

```plaintext
const count : int = 10;
...
x := count * 5;
y := x ^ 3;
```

Intermediate code after constant propagation

Local dead assignment elimination

- If the left hand side of an assignment is never read again before being overwritten, then remove the assignment
- This sometimes happens while cleaning up from other optimizations (as with many of the optimizations we consider)

Example

```plaintext
const count : int = 10;
...
x := count * 5;
y := x ^ 3;
x := input;
```

Intermediate code after constant propagation

Common subexpression elimination

- Avoid repeating the same calculation
- Requires keeping track of available expressions

CSE example:

```plaintext
t1 := *(fp + ioffset)
t2 := t1 * 4
t3 := fp + t2
t4 := *(t3 + aoffset)
t5 := *(fp + ioffset)
t6 := t5 * 4
t7 := fp + t6
t8 := *(t7 + boffset)
t9 := t4 + t8
```
Intraprocedural optimizations

- Enlarge scope of analysis to entire procedure
- Provides more opportunities for optimization
- Have to deal with branches, merges and loops
- Can do constant propagation, common subexpression elimination, etc. at this level
- Can do new things, too, like loop optimizations
- Optimizing compilers usually work at this level

Code motion

- Goal: move loop-invariant calculations out of loops
- Can do this at the source or intermediate code level

```plaintext
for i := 1 to 10 do
    a[i] := a[i] + b[j];
    z := z + 10000
end
```

Loop induction variable elimination

- For-loop index is an induction variable
  - Incremented each time through the loop
  - Offsets, pointers calculated from it
  - If used only to index arrays, can rewrite with pointers
    - Compute initial offsets, pointers before loop
    - Increment offsets, pointers each time around loop
    - No expensive scaling in the loop

Example

```plaintext
for i := 1 to 10 do
    a[i] := a[i] + x;
end
    *p := *p + x;
end
```
Global register allocation

- Try to allocate local variables to registers
- If two locals don’t overlap, then give them the same register
- Try to allocate most frequently used variables to registers first

Register allocation by coloring

- As before, IR gen as if infinite regs avail
- Build interference graph:
  - x := a+5;
  - y := b*2;
  - z := x/3;
  - a := y-2;
- Colorable with few colors (regs)?
  - NP-hard, but …
- If not, pick a node & generate spill code

Interprocedural optimizations

- What happens if we expand the scope of the optimizer to include procedures calling each other
  - In the broadest scope, this is optimization of the program as a whole
  - We can do local, intraprocedural optimizations at a bigger scope
  - For example, constant propagation
  - But we can also do entirely new optimizations, such as inlining

Interprocedural opt: Issues

procedure P() {
    x: int;
    x := 10;
    Q( );
    x := x+1;
    if x == 11 then
        …
    Q(), but Q declared in P
    …
}

Inlining

Replace procedure call with the body of the called procedure

const pi:real := 3.14159;
proc area(rad:int):int;
begin
    return pi*(rad^2);
end;

r := 5;
...
output := area(r);

Questions about inlining:

- How to decide where the payoff is sufficient to inline?
  - The real decision depends on dynamic information about frequency of calls
  - In most cases, inlining causes the code size to increase; when is this acceptable?
  - Others?
Optimization and debugging

- Debugging optimized code is often hard
- For example, what if:
  - Source code statements have been reordered?
  - Source code variables have been eliminated?
  - Code is inlined?
- In general, the more optimization there is, the more complex the back-mapping is from the target code to the source code … which can confuse a programmer

Summary of optimization

- Larger scope of analysis yields better results
- Most of today’s optimizing compilers work at the intraprocedural level, with some doing some work at the interprocedural level
- Optimizations are usually organized as collections of passes
- The presence of optimizations may make other parts of the compiler (e.g., code gen) easier to write
  - E.g., use a simple instruction selection algorithm, knowing that the optimizer can, in essence, act to improve these instruction selections