Optimizations

Identify inefficiencies in target or intermediate code
- usually IR
- machine-independent optimizations

Goal: replace with equivalent but better sequences
- fewer instructions
- cheaper instructions
- different instructions
- fewer registers

"Optimize" overly optimistic; "usually improve"

Must reproduce the same results

Optimizations

Scope of study for optimizations:
- **peephole**:
  look at single instruction/adjacent instructions
- **local**:
  look at straight-line sequence of statements
- **global** (**intr*a*procedural**):
  look at whole procedure
- **int*er*procedural**:
  look across procedures

Larger scope ⇒ better optimization, more complexity

Peephole optimization

After code generation, look at adjacent instructions
(a "**peephole**" on the code stream)
- try to replace:
  - single instruction
  - adjacent instructions

  with a shorter, faster sequence of code

Algebraic simplifications

Eliminate redundant code

```
x := x + 0
x := x * 1
x := (x + y) - y
```


Algebraic simplifications

Strength reduction
- replace expensive operation with a cheaper one

```
a := b * 8;

mult $5, $4, #8
    ⇒
sll $5, $4, 3
```


Algebraic simplifications

Constant folding: evaluate constant expressions

```
x := 3 + 4

li $4, #7
sw $4, 0(addr of x)
```

Redundant instruction elimination

Eliminate redundant **loads & stores**

```
a := b + b
c := a
...

lw $4, 0(addr of b)
lw $5, 0(addr of b)
add $6, $4, $5
sw $6, 0(addr of a)

lw $4, 0(addr of a)
sw $4, 0(addr of c)
...

   ⇒

lw $4, 0(addr of b)
add $6, $4, $4
sw $6, 0(addr of a)

sw $6, 0(addr of c)
...
```


Redundant instruction elimination

Unreachable code

Unlabeled instructions after a `goto` can be eliminated

```
#define debug 0
...
if (debug) {...}

   ⇒
   if 0 <> 1 goto L2
   debug stmts
L2: ...

   ⇒
   goto L2
   debug stmts      (the peephole opt)
L2: ...

   ⇒
L2: ...
```


Flow of control optimizations

"Adjacent" instructions = "adjacent in control flow"

Eliminate **jumps to jumps**

```
    goto L1
    ...
L1: goto L2

     ⇒

    goto L2
    ...
L1: goto L2
```


Flow of control optimizations

Eliminate **jumps after conditional branches**

```
if a < b then
   if c < d then
      do nothing
   else
      stmt_1;
   end;
else
   stmt_2;
end;
```

assume a in $1, b in $2, c in $3, d in $4

```
    slt $5, S1, $2
    beq $5, $0, L2
    slt $5, $3, $4
    beq $5, $0, L1    ⇒    bne $5, $0, L3
    br L3             ⇒
L1: stmt_1            ⇒    stmt_1
    br L3
L2: stmt_2
L3: ...
```


Instruction selection

```
sub sp, 4, sp
mov r1, 0(sp)
   ⇒
mov r1, -(sp)


mov 12(fp), r1
add r1, 1, r1
mov r1, 12(fp)
   ⇒
inc 12(fp)
```

(68000 code)

Local optimization

Analysis and optimizations within a basic block

Basic block: straight-line sequence of statements

- no control flow into or out of sequence
- start of basic blocks
  - beginning of program
  - target of conditional or conditional goto
  - statement immediately following conditional or conditional goto

Better than peephole

Not too hard to implement

Machine-independent, if done on intermediate code

Local constant folding & constant propagation

Constant folding: evaluate constant expressions

Constant propagation:

- if a variable is assigned a constant, replace downstream uses of the variable with that constant
- why do we want the compiler, not the programmer to do constant propagation?

Can enable more constant folding

Local constant folding & constant propagation

Source:

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

Unoptimized intermediate code:

```
t1 := 10
t2 := 5
t3 := t1 * t2
x := t3

t4 := x
t5 := 3
t6 := exp(t4, t5)
y := t6
```


Local copy propagation

Copy propagation:

- if have an assignment, use RHS in downstream references to LHS
- can lead to **dead code elimination**

```
x := t3
y := z + x
   ⇒
x := t3
y := z + t3
```


Local dead assignment elimination

Eliminate when result is never referenced again

- define: assign to
- use: reference
- **live**: variable is live at some program point if used later on
  otherwise dead

```
x := y + z;
a := y + x;
```


Local common subexpression elimination (CSE)

Common subexpression:

- previously computed
- none of its variables have changed

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Use the previously computed value instead of repeating the same calculation

Keep track of available expressions

Local common subexpression elimination (CSE)

Source:

```
... a[i] + b[i] ...
```

Unoptimized intermediate code:

```
t2 := i * 4
t3 := t2 + &a
t4 := *(t3)

t6 := i * 4
t7 := t6 + &b
t8 := *(t7)

t9 := t4 + t8
```

Int*ra*procedural ("global") optimizations

Enlarge scope of analysis to whole procedure

- more opportunities for optimization
- have to deal with splits, merges, and loops

Can do constant propagation,
common subexpression elimination, etc.
at global level

Can do new optimizations, e.g., **loop optimizations**

Optimizing compilers usually work at this level

Loop-invariant code motion

Goal: hoist **loop-invariant calculations** out of loops

- result does not change as execute the iterations

Source:

```
for i := 1 to 10 do
    a[i] := a[i] + b[j];
    z := z + 100000;
end;
```

Transformed source:

```
var1 := b[j];
var2 := 100000;
for i := 1 to 10 do
    a[i] := a[i] + var1;
    z := z + var2;
end;
```


Code motion at intermediate code level

Source:

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

Unoptimized intermediate code:

```
      *(fp + offset_i) := 1
_top:
      if *(fp + offset_i) > 10 goto _done

      t1 := *(fp + offset_j)
      t2 := t1 * 4
      t4 := *(t2 + &b)

      t5 := *(fp + offset_i)
      t6 := t5 * 4
      *(t6 + &a) := t4

      t9 := *(fp + offset_i)
      t10 := t9 + 1
      *(fp + offset_i) := t10

      goto _top
_done:
```


Loop induction variable elimination

For-loop index is **induction variable**

If used only to index arrays, can rewrite with pointers

Source:

```
for i := 1 to 10 do
    a[i] := a[i] + x;
end;
```

Transformed source:

```
for p := &a[1] to &a[10] do
    *p := *p + x;
end;
```


Global register allocation

Try to allocate global variables to registers

- avoid stores and reloading at basic block boundaries

Make sure a globally used variable doesn't conflict with a local

Example:

```
procedure foo(n:int, x:int):int;

    var sum:int, i:int;
begin

    sum := x;

    for i := 1 to n do

        sum := sum + i;
    end;

    q := 17;
    output q;

    return sum;
end foo;
```


How to do global optimizations?

Represent procedure by a **control flow graph**

Each **basic block** is a node in graph

Branches become edges in graph

Example:

```
procedure foo(n:int);
    var a:array[10] of int;
    var i:int, j:int;
begin
    j := 10;
    for i := 0 to 9 do
        a[i] := a[i] + (n - j * 2);
    end;
end foo;
```

Analysis of control flow graphs

To do optimization,
  first analyze important info, then do transformations

Propagate info through graph

At branches: copy info along both branches

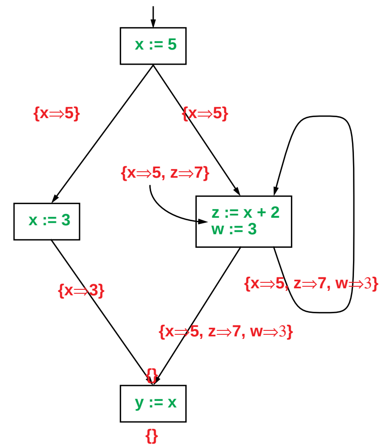
At merges: combine info, being conservative

- constant propagation, CSE: intersection
- live variable analysis: union

At loops: iterate until annotations do not change (**fixpoint**)

E.g., global constant propagation:
  propagate *name*→*constant* mappings through graph

Global constant propagation



Inter*er*procedural optimizations

Expand scope of analysis to procedures calling each other

Can do local, intraprocedural optimizations at larger scope

Can do new optimizations, e.g., **inlining**

Inlining

Replace procedure call with body of called procedure

Source:

```
const pi:real := 3.1415927;
proc circle_area(radius:int):int;
  begin
    return pi * (radius ^ 2);
  end circle_area;
...
r := 5;
...
output := circle_area(r);
```

After inlining:

```
const pi:real := 3.1415927;
...
r := 5;
...
output := pi * (r ^ 2);
```


Summary

Optimizations lead to more efficient code

Enlarging scope of analysis yields better results

- today, most optimizing compilers work at the global/intraprocedural level

Optimizations organized as collections of passes

Some optimizations enable others

- e.g., CSE or constant propagation -> dead assignment elimination
- e.g., constant propagation -> constant folding

Presence of optimizations makes other parts of compiler easier

- register allocation has fewer temporaries
- target code generation only has to generate code

Optimization summary

	Peep hole	Local	Intra-procedural	Inter-procedural
algebraic simplification	x			
strength reduction	x			
constant folding & propagation	x	x	x	x?
redundant ld/st	x	x	x	x?
dead code elimination	x	x		
jumps to jumps/cond. jumps	x			
CSE	x	x	x	x?
copy propagation	x	x	x	x?
loop invariant code motion			x	
induction variable elimination			x	
instruction selection	x	x		
register allocation	x	x	x	x?
code scheduling		x	x	
inlining				x