Run-time system calls

Can implement operations by...

- compiled machine code, or
- calls to functions in libraries linked with compiled code

Example run-time system calls:

`input`, `output` in PL/0

`printf`/`scanf`, `malloc`/`free`, `strlen`, etc. in C

`new` in Java/C++

garbage collection, threads, exception propagation in Java

Optimizations

Identify inefficiencies in target or intermediate code
Replace with equivalent but better sequences

"Optimize" overly optimistic

- "usually improve" better

An example

Source code:

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

Intermediate code:

```
t1 := *(fp + ioffset)   // i
t2 := t1 * 4
t3 := fp + t2
t4 := *(t3 + aoffset)    // a[i]
t5 := 2
t6 := t5 * 4
t7 := fp + t6
t8 := *(t7 + boffset)    // b[2]
t9 := t4 + t8
*(fp + xoffset) := t9    // x := ...
t10:= *(fp + xoffset)   // x
t11:= 5
t12:= t10 - t11
t13:= *(fp + ioffset)   // i
t14:= t13 * 4
t15:= fp + t14
*(t15 + coffset) := t15  // c[i] := ...
```


Kinds of optimizations

Scope of study for optimizations:

- **peephole**:
  look at adjacent instructions
- **local**:
  look at straight-line sequence of statements
- **global** (**int*ra*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 adjacent instructions with something faster

Example:

```
sw $8, 12($fp)
lw $12, 12($fp)
   ⇒
sw $8, 12($fp)
mv $12, $8
```


More examples

On 68k:

```
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)
```

Do complex instruction selection through peephole optimization

Peephole optimization of jumps

Eliminate jumps to jumps

Eliminate jumps after conditional branches

"Adjacent" instructions = "adjacent in control flow"

Source code:

```
if a < b then
   if c < d then
      # do nothing
   else
      stmt1;
   end;
else
   stmt2;
end;
```


Algebraic simplifications

"constant folding", "strength reduction"

```
z := 3 + 4

z := x + 0
z := x * 1

z := x * 2
z := x * 8


z := x / 8

float x,y; z := (x + y) - y
```

Can be done by peephole optimizer, or by code generator

Local optimization

Analysis and optimizations within a **basic block**

Basic block: straight-line sequence of statements

- no control flow into or out of sequence

Better than peephole

Not too hard to implement

Machine-independent, if done on intermediate code

Local constant propagation

If variable assigned a constant,
replace downstream uses of the variable with constant

Can enable more constant folding

Example:

```
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 dead assignment elimination

If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment

E.g. clean-up after previous optimizations

Example:

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

Intermediate code after constant propagation:

```
x := 50
t6 := exp(50, 3)
y := t6
x := input()
```


Local common subexpression elimination

Avoid repeating the same calculation

Keep track of **available expressions**

Source:

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

Unoptimized intermediate code:

```
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
```

Int*ra*procedural ("global") optimizations

Enlarge scope of analysis to whole procedure

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

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

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

Optimizing compilers usually work at this level

Code motion

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

Can do at source level or at intermediate code level

Source:

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

Transformed source:

```
t1 := b[j];
t2 := 10000;
for i := 1 to 10 do
   a[i] := a[i] + t1;
   z := z + t2;
end;
```


Code motion at intermediate code level

Source:

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

Unoptimized intermediate code:

```
   *(fp + ioffset) := 1
_top:
   if *(fp + ioffset) > 10 goto _done
   t1 := *(fp + joffset)
   t2 := t1 * 4
   t3 := fp + t2
   t4 := *(t3 + boffset)
   t5 := *(fp + ioffset)
   t6 := t5 * 4
   t7 := fp + t6
   *(t7 + aoffset) := t4
   t9 := *(fp + ioffset)
   t10 := t9 + 1
   *(fp + ioffset) := t10
   goto _top
_done:
```


Loop induction variable elimination

For-loop index is **induction variable**

- incremented each time around 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 loop

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 + t;
end;
```

Global register allocation

Try to allocate local variables to registers

If two locals don't overlap, then give to same register
Try to allocate most frequently-used variables to regs first

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;
   return sum;
end foo;
```


Inte*r*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

Enlarging scope of analysis yields better results

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

Optimizations organized as collections of passes

Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write