Code optimization

Assume the program is represented in some low-level IR.

Peephole optimization

- find logically adjacent instructions that can be combined
  - use a very small context (3 – 10 instructions)
  - combining i_1 and i_2 => faster i_3
- work at register-transfer language(rtl) level
  - machine description in rtl
  - low-level IR description in rtl
- using pattern matching, synthesize more complex instructions
- useful for implementing many machine dependent optimizations

Code optimization

Generating “peephole” code generators

- provide a one-to-one translation (one-to-one) for IR
- add patterns to improve code (more complex instructions and addressing modes)

Training generator

- feed a set of representative programs to the trainer and let it build a table by exhaustive search
- onetime expense (and it is expensive)
- use a linear time pattern matcher run from the tables produced by the trainer

Code optimization

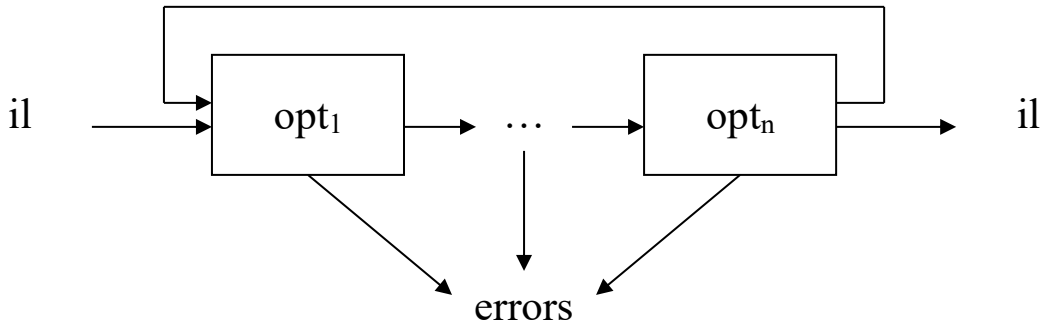
Typical machines

- RT/PC w/o floating point – 70-100 instructions
- MC68020 – millions of possible instructions

Recent work

- these two camps have merged
- highly efficient BURS(Bottom Up Rewrite System) systems
- combines tree-matching and peephole

Optimizer(middle end)



An optimization is a transformation expected to:

1. improve the running time of a program, or
2. decrease its space requirements

Many compilers include an optimizer

- often structured as a series of passes
- tries to improve code quality
- may repeat transformations several times

Optimizing compilers

- produce “improved” code, not “optimal” code
- can sometimes produce worse code

Code optimization

How can optimizations improve code quality?

Machine independent transformations

1. replace a redundant computation with a reference
2. move evaluation to a less frequently executed place
3. specialize some general purpose code
4. find useless code and remove it
5. expose opportunities (enable) for other optimizations

Machine dependent transformations

1. replace a costly operation with a cheaper one
2. hide latency
3. replace a sequence of instructions with a more powerful one

Classical transforamtion examples

Unreachable code

- eliminate code not reached during program execution
- analyze control flow graph

```
        goto L:
        { unreachable code }
L:
```

Control-flow simplification

- remove jumps to jumps
- analyze targets of jumps

```
        goto L:
        { code }
L:      goto M:
        { code }
M:
```

Classical Transformation Examples

Algebraic simplification

- simplify arithmetic expression
- analyze expression trees

```
X := X + 0
X := X * 1
```

Constant folding

- replace constant expression with result
- analyze expression trees

```
A := 5
B := 6
C := B + A
```

Idiom recognition

- replace operations with less expensive idioms
- analyze expression

```
B := A * 16
D := B / 4
```

Classical transformation examples

Available expressions

- reuse values always available
- local/global data flow analysis

```
C := B + D
D := B + D
```

Dead code elimination

- eliminate unnecessary computations
- local/global data flow analysis

```
A := 5
A := 6
```

Copy propagation

- propagate names into copy instruction
- local/global data flow analysis

```
B := A
C := B
```

Basic blocks

Definition

- sequence of code
- control enters at top, exits at bottom
- no branch/halt except at end

Construction algorithm (for 3-address code)

1. determine set of leaders
   A. first statement
   B. target of goto or conditional goto
   C. statement following goto or conditional goto
2. add to basic block all statements following leader up to next leader or end of program

Example:

```
        A := 0
        if (<cond>) goto L
        A := 1
        B := 1
L:      C := A
```

Scope of optimizations

Scope

- peephole – across a few instructions
- local – within basic block
- global – across basic blocks
- refers to both analyses and optimizations

Some optimizations may be applied locally or globally(e.g., dead code elimination)

```
A := 0                  A := 0
A := 1                  if (<cond>) goto L
B := A                  A := 1
                        B := A
```

Some optimizations require global analysis
(e.g., loop-invariant code motion):

```
while (<cond>) do
        A := B + C
        foo(A)
end
```

Types of optimizations

Types of optimizations.

- classical
  reduce the number/cost of instructions executed
- register allocation
  keep values in registers as much as possible
- instruction scheduling
  hide instruction latency, exploit instruction-level parallelism
- data locality
  keep data accesses in faster levels of memory hierarchy(registers, cache, memory)
- multiprocessing
  compute in parallel on multiple processors

Optimization framework

- ideally, maintain separation of concerns
- in practice, integrate optimization algorithms