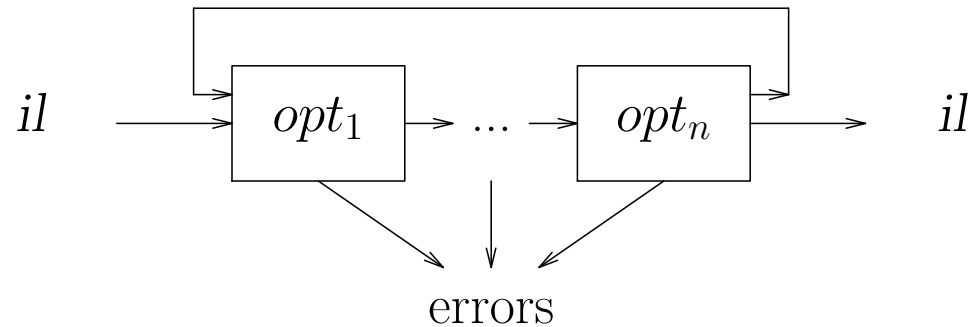


Optimizer (middle end)



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

Why are optimizers needed?

Reduce programmer effort

- automatically generate efficient code
- less work for programmer
- below “optimal” hand-optimized code

Undo high-level abstractions

- some optimizations not possible for language
- flatten control flow to branches
- convert method lookups to subroutine calls
- map data structures to addresses

Maintain performance portability

- performance depends on architecture
- optimizations by programmer too specific
- compiler can customize program for processor

Code optimizations

Reduce execution time

- historically, to avoid assembly coding
- support higher levels of abstraction
- support more complex processors
- important applications: science, databases

Reduce space

- historically, small expensive memories
- may trade space for speed
- space may reduce speed (caches)
- new areas: internet applets, embedded processors

Level of optimization

- source code
- intermediate representation
- binary machine code
- at run-time

Code optimization

How can optimizations improve code quality?

Machine-independent transformations

1. remove unnecessary computations
2. simplify control structures
3. move code to a less frequently executed place
4. specialize some general purpose code
5. find useless code and remove it
6. expose opportunities (enable) for other optimizations

Machine-dependent transformations

1. replace complex operation with simpler one
2. exploit special instructions (MMX)
3. exploit memory hierarchy (registers, cache)
4. exploit parallelism (ILP, VLIW, vectors)

Code optimization

Types of optimizations

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

Optimization framework

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

Code optimization

Three considerations arise in applying a transformation.

- *safety*

Does applying the transformation change the results of executing the code?

- *profitability*

Is there a reasonable expectation that applying the transformation will improve the code?

- *opportunity*

Can we efficiently and frequently find places to apply optimization?

Need a clear understanding of these issues.

Profitability is particularly tricky...

Learn how the compiler decides when transformations will be applicable, safe, and profitable.

Classical optimizations

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 optimizations

Algebraic simplification

- simplify arithmetic expressions
- analyze expression trees

```
A := 0
C := B + A
```

Constant folding

- replace constant expressions with result
- analyze expression trees

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

Idiom recognition

- replace operations with less expensive idioms
- analyze expression trees

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

Classical optimizations

Available expressions

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

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

Dead code elimination

- eliminate unnecessary computations
- local/global data flow analysis

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

Copy propagation

- propagate names into copy instructions
- 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

Optimizations may be characterized by their scope

- *peephole* — across a few instructions
- *local* — within basic block
- *global* — across basic blocks
- *interprocedural* — across procedures

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 optimizations):

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