An optimization is a transformation expected to:

- Reduce programmer effort
- Undo high-level abstractions
- Map data structures to addresses
- Convert method lookups to subroutine calls
- Halve control flow to branches
- Some optimizations not possible for language

Why are optimizers needed?

- Compiler can customize program for processor
- Optimizations by programmer too specific
- Performance depends on architecture
- Maintain performance portability
- Can sometimes produce worse code
- Often structured as a series of passes
- Many compilers include an optimizer

Optimizing compilers:
- May repeat transformations several times
- Tend to improve code quality
- Can sometimes produce worse code
- Can sometimes produce more code
- Can produce "improved" code, not "optimal" code
- Many compilers include an optimizer
- Some optimizations not possible for language

Optimizations:
- Improve the run time of a program
- Decrease its space requirements
- Automatically generate efficient code
- Less work for programmer
- Below "optimal", hand-optimized code
- Can sometimes produce worse code

Reduction optimizer (middle end)
<table>
<thead>
<tr>
<th>Level of optimization</th>
<th>Machine-independent optimizations</th>
<th>Machine-dependent optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>1. remove unnecessary computations</td>
<td>6. expose opportunities (enable) for other</td>
</tr>
<tr>
<td>Intermediate representation</td>
<td>2. simplify control structures</td>
<td>5. find useless code and remove it</td>
</tr>
<tr>
<td>Binary machine code</td>
<td>3. move code to a less frequently executed place</td>
<td>4. specialize some general purpose code</td>
</tr>
</tbody>
</table>

### How can optimizations improve code quality?

The primary benefits of code optimizations include:

- **Reduction of execution time**
- **Improvement of code quality**
- **Support for complex processors**
- **Support for higher levels of abstraction**
- **Reduction of space**
- **Importance of parallel processing**
- **5. Reduction of space**

Machine-dependent optimizations:

1. Replace complex operations with simpler one
2. Replace special instructions with simpler ones
3. Specialize some general purpose code
4. Move code to a less frequently executed place
5. Find useless code and remove it
6. Expose opportunities (enable) for other

Machine-independent optimizations:

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

### Reducing execution time

- Binary machine code
- Intermediate representation
- Source code

### Reducing space

- New areas: Internet applications, embedded
- Space may reduce speed (caches)
- May trade space for speed
- Historically, small expensive memories

### Importance of parallel processing

- Support more complex processors
- Support higher levels of abstraction
- Practically, to avoid assembly coding
Code optimization

Types of optimizations

Classical
- Reduce the number of instructions executed
- In practice, integrate optimization algorithms
- Ideally, maintain separation of concerns

Optimization framework
- Compute in parallel on multiple processors
- Hierarchy (registers, cache, TLB, memory)
- Keep data accesses in faster levels of memory
- Data locality
- Parallelism
- Instruction-level parallelism
- Keep values in registers; eliminate loads/stores
- Register allocation
- Instruction scheduling
- Hide instruction latency; exploit instruction-level parallelism

Profitability
- Translations will be applicable, safe, and profitable.
- Learn how the compiler decides when profitability is particularly tricky.

Profitability is particularly tricky.

Need a clear understanding of these issues.

Apply optimization?
- Can we efficiently and frequently find places to
  apply optimization?

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

Profitability
- Reusable of executing the code?
  Does applying the transformation change the
  safety?

Safety
- The considerations arise in applying a
  transformation.

Three considerations arise in applying a
Classical Transformation Examples

Unreachable code

execute

eliminate unreachable code

control-flow simplification

analyze control flow graph

constant folding

analyze constant expressions with result

constant folding

c = b + a

a = 0

analyze expression trees

simplify arithmetic expressions

algebraic simplification

constant folding

c = b + a

b = 6

a = 5

analyze expression trees

replace constant expressions with result

idiom recognition

c = b + a

b = 6

a = 5

analyze expression trees

replace operations with less expensive idioms

idiom recognition

a = b / 4

b = a * 16

analyze expression trees

replace operations with less expensive idioms

control-flow simplification

analyze targets of jumps

remove jumps to jumps

control-flow simplification

analyze unreachable code

unreachable code

unreachable code

unreachable code

Classical Transformation Examples
Classical Transformation Examples

Available expressions

Reuse values always available

Local/global data flow analyses

Copy propagation

Propagate names into copy instructions

Dead code elimination

Eliminate unnecessary computations

Local/global data flow analyses

Example:

I: T = A
B = 1
A = I
if (cond) goto L
A = 0

Construction algorithm (for 3-address code)

1. Determine set of leaders
2. Add to basic block all statements following (c) statement following goto or conditional goto
   (b) target of goto or conditional goto
   (a) first statement
3. Determine set of leaders
4. no branch/leaf except at end
   control enters at top, exits at bottom
   sequence of code

Definition

T = A
B = 1
A = I
if (cond) goto L
A = 0

Example:

C = B
B = A

Local/global data flow analyses

Propagate names into copy instructions

Copy propagation

Propagate names into copy instructions

Local/global data flow analyses

Copy propagation

Propagate names into copy instructions

Local/global data flow analyses

Example:

C = B
B = A

Available expressions

Reuse values always available

Local/global data flow analyses

Copy propagation

Propagate names into copy instructions

Local/global data flow analyses

Available expressions

Reuse values always available

Local/global data flow analyses

Copy propagation
Scoping of Optimizations

Scoping

peephole across a few instructions

local within basic block

global across basic blocks

interprocedural — across procedures

global — across basic blocks

local — within basic block

peephole — across a few instructions

Scope of Optimizations

Some optimizations may be applied locally or globally:

Some optimizations require global analysis:

( ≠ = dead code elimination)

A = B
B = A
T = cond
T = true
T = false

end

for

A := B + C
while (cond) do

while (cond) do

end

end

if (cond) goto L

L := A

L := 0