Many compilers include an optimizer

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

Definition

An *optimization* is a transformation that is *expected* to:

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

*Classical* optimizations make the number of operations actually executed be less than or equal to the number expected by a naive programmer.

More recent optimizations improve the ordering of instructions or the locality of data and instructions.

The point

- produce “improved” code, not “optimal” code
- can sometimes produce worse code
- range of speedup might be from 1.01 to 4 (*or more*)
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 an opportunity for another optimization

*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
Code optimization

Three considerations arise in applying a transformation.

- *safety*
- *profitability*
- *opportunity*

Need a clear understanding of these issues.

- the literature often hides these issues
- every discussion *should* list them clearly
Safety

Fundamental question

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

yes ⇒ don’t do it!

no ⇒ it is safe

Compile-time analysis

• may be safe in all cases (loop unrolling)
• analysis may be simple (dags and cses)
• may require complex reasoning (data-flow analysis)
Profitability

Fundamental question

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

yes ⇒ do it!

no ⇒ don’t do it

Compile-time estimation

• always profitable

• “seat of the pants” rule(s)

• compute benefit \((rare)\)
Opportunity

Fundamental question

Can we locate application sites efficiently?

yes ⇒ compilation time won’t suffer

no ⇒ improvement had better be big

Issues

• provides a framework for applying transformation
• systematically find all sites
• update safety information to reflect previous changes
• order of application (hard)
Code optimization

Successful optimization requires

- test for safety, profitability should be
  - $O(1)$ per transformation, or
  - $O(n)$ for whole routine (maybe $n \log n$)

- profit is local improvement $\times$ executions
  $\Rightarrow$ focus on loops

- want to minimize side effects like code growth

Look for things programmers don’t do well
Example

Loop Unrolling

Idea:

reduce loop overhead by creating multiple successive copies of the loop’s body and increasing the increment appropriately

Safety: always safe

Profitability: reduces overhead
  (instruction cache blowout)
  (subtle secondary effects)

Opportunity: loops

Unrolling is easy to understand and perform.
Example

Matrix-matrix multiply

\[
\begin{align*}
\text{do } i & \leftarrow 1, n, 1 \\
\text{do } j & \leftarrow 1, n, 1 \\
\quad c(i,j) & \leftarrow 0 \\
\text{do } k & \leftarrow 1, n, 1 \\
\quad c(i,j) & \leftarrow c(i,j) + a(i,k) \times b(k,j)
\end{align*}
\]

- \(2n^3\) flops, \(n^3\) loop increments and branches
- each iteration does 2 loads and 2 flops

This is the most overstudied example in the literature
Example

Matrix-matrix multiply  
(4 word cache line)

do i ← 1, n, 1
   do j ← 1, n, 1
      c(i, j) ← 0
      do k ← 1, n, 4
         c(i, j) ← c(i, j) + a(i, k) * b(k, j)
         c(i, j) ← c(i, j) + a(i, k+1) * b(k+1, j)
         c(i, j) ← c(i, j) + a(i, k+2) * b(k+2, j)
         c(i, j) ← c(i, j) + a(i, k+3) * b(k+3, j)

- $2n^3$ flops, $\frac{n^3}{4}$ loop increments and branches
- each iteration does 8 loads and 8 flops
- memory traffic is better
  - $c(i, j)$ is reused  
    (put it in a register)
  - $a(i, k)$ references are from cache
  - $b(k, j)$ is problematic
Example

Matrix-matrix multiply  

(to improve traffic on \( b \))

\[
\begin{align*}
\text{do } j & \leftarrow 1, n, 1 \\
\text{do } i & \leftarrow 1, n, 4 \\
c(i, j) & \leftarrow 0 \\
\text{do } k & \leftarrow 1, n, 4 \\
c(i, j) & \leftarrow c(i, j) + a(i, k) \times b(k, j) \\
& + a(i, k+1) \times b(k+1, j) \\
& + a(i, k+2) \times b(k+2, j) \\
& + a(i, k+3) \times b(k+3, j) \\
c(i+1, j) & \leftarrow c(i+1, j) + a(i+1, k) \times b(k, j) \\
& + a(i+1, k+1) \times b(k+1, j) \\
& + a(i+1, k+2) \times b(k+2, j) \\
& + a(i+1, k+3) \times b(k+3, j) \\
c(i+2, j) & \leftarrow c(i+2, j) + a(i+2, k) \times b(k, j) \\
& + a(i+2, k+1) \times b(k+1, j) \\
& + a(i+2, k+2) \times b(k+2, j) \\
& + a(i+2, k+3) \times b(k+3, j) \\
c(i+3, j) & \leftarrow c(i+3, j) + a(i+3, k) \times b(k, j) \\
& + a(i+3, k+1) \times b(k+1, j) \\
& + a(i+3, k+2) \times b(k+2, j) \\
& + a(i+3, k+3) \times b(k+3, j)
\end{align*}
\]
Example

What happened?

- interchanged $i$ and $j$ loops
- unroll $i$ loop
- fuse inner loops

- $2n^3$ flops, $\frac{n^3}{16}$ loop increments and branches
- first assignment does 8 loads and 8 flops
- $2^{nd}$ thru $4^{th}$ do 4 loads and 8 flops

- memory traffic is better
  - $c(i, j)$ is reused (register)
  - $a(i, k)$ references are from cache
  - $b(k, j)$ is reused (register)
Example

*It is not as easy as it looks*

**Safety:** loop interchange?

  - loop unrolling?
  - loop fusion?

**Profitability:** machine dependent \((mostly)\)

**Opportunity:** find *memory-bound* loop nests

Summary

- chance for large improvement \((3 \times \text{on SPARC})\)
- answering the fundamental questions is tough
- resulting code is *ugly*

*Matrix-matrix multiply is everyone’s favorite example*
Some example optimizations

Peephole optimizations

- redundant loads and stores
- unreachable code
- control-flow simplification
- algebraic simplification
- reduction in strength

Classical optimizations

- copy propagation
- dead code elimination
- code hoisting
- reduction in strength
- expression folding
- constant propagation
- available subexpressions
- live variables