COMP 150-AVS
Fall 2018

Optimization
Introduction

• An *optimization* is a transformation “expected” to
  ■ Improve running time
  ■ Reduce memory requirements
  ■ Decrease code size

• No guarantees with optimizers
  ■ Produces “improved,” not “optimal” code
  ■ Can sometimes produce worse code
Why are optimizers needed?

- Reduce programmer effort
  - Don’t make programmers waste time doing simple opts
- Allow programmer to use high-level abstractions without penalty
  - E.g., convert dynamic dispatch to direct calls
- Maintain performance portability
  - Allow programmer to write code that runs efficiently everywhere
  - Particularly a challenge with GPU code
Two laws and a measurement

• Moore’s law
  - Chip density doubles every 18 months
  - Until now, has meant CPU speed doubled every 18 months
    - These days, moving to multicore instead

• Proebsting’s Law
  - Compiler technology doubles CPU power every 18 years
    - Difference between optimizing and non-optimizing compiler about 4x
    - Assume compiler technology represents 36 years of progress

• Worse: runtime performance swings of up to 10% can be expected with no changes to executable
  - [http://dl.acm.org/citation.cfm?id=1508275](http://dl.acm.org/citation.cfm?id=1508275)
Dimensions of optimization

• Representation to be optimized
  - Source code/AST
  - IR/bytecode
  - Machine code

• Types of optimization
  - Peephole — across a few instructions (often, machine code)
  - Local — within basic block
  - Global — across basic blocks
  - Interprocedural — across functions
Dimensions of optimization (cont’d)

• Machine-independent
  ▪ Remove extra computations
  ▪ Simplify control structures
  ▪ Move code to less frequently executed place
  ▪ Specialize general purpose code
  ▪ Remove dead/useless code
  ▪ Enable other optimizations

• Machine-dependent
  ▪ Replace complex operations with simpler/faster ones
  ▪ Exploit special instructions (MMX)
  ▪ Exploit memory hierarchy (registers, cache, etc)
  ▪ Exploit parallelism (ILP, VLIW, etc)
Selecting optimizations

• Three main considerations
  ▪ Safety — will optimizer maintain semantics?
    - Tricky for languages with partially undefined semantics!
  ▪ Profitability — will optimization improve code?
  ▪ Opportunity — could optimization often enough to make it worth implementing?

• Optimizations interact!
  ▪ Some optimizations enable other optimizations
    - E.g., constant folding enables copy propagation
  ▪ Some optimizations block other optimizations
Some classical optimizations

• Dead code elimination

  jmp L
  /* unreachable */
  L: ...

  if true then
      ...
  else
      /* unreachable */

  a = 5 /* dead */
  a = 6

  jmp L
  /* unreachable */
  L: goto M
  M: ...

  jmp M
  /* unreachable */
  M: ...

• Also, unreachable functions or methods

• Control-flow simplification
  • Remove jumps to jumps
More classical optimizations

- Algebraic simplification
  
  \[
  x = a + 0 \quad \Rightarrow \quad x = a \quad \Rightarrow \quad x = a * 0 \quad \Rightarrow \quad x = 0
  \]

  - Be sure simplifications apply to modular arithmetic

- Constant folding
  
  - Pre-compute expressions involving only constants

  \[
  \begin{align*}
  a &= 5 \\
  b &= 6 \\
  x &= a + b \quad /* \quad a, b \ \text{dead} */
  \end{align*}
  \]

  \[
  \begin{align*}
  a &= 5 \\
  b &= 6 \\
  x &= 11 \quad /* \quad a, b \ \text{dead} */
  \end{align*}
  \]

- Special handling for idioms
  
  - Replace multiplication by shifting
  - May need constant folding to enable sometimes
More classical optimizations

- Common subexpression elimination

\[
\begin{align*}
a &= b + c \\
d &= b + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
d &= a
\end{align*}
\]

- Copy propagation

\[
\begin{align*}
b &= a \\
c &= b \\
/* b dead */
\end{align*}
\]

\[
\begin{align*}
b &= a \\
c &= a \\
/* b dead */
\end{align*}
\]

\[
\begin{align*}
c &= a
\end{align*}
\]

dead
code
elim
Example

Fortran (!) source code:

..

..

..

sum = 0
do 10 i = 1, n
10 sum = sum + a(i) * a(i)

..

..
Three-address code

1. \( \text{sum} = 0 \)
2. \( i = 1 \)
3. \( \text{if } i > n \text{ goto 15} \)
4. \( t1 = \text{addr}(a) - 4 \)
5. \( t2 = i \times 4 \)
6. \( t3 = t1[t2] \)
7. \( t4 = \text{addr}(a) - 4 \)
8. \( t5 = i \times 4 \)
9. \( t6 = t4[t5] \)
10. \( t7 = t3 \times t6 \)
11. \( t8 = \text{sum} + t7 \)
12. \( \text{sum} = t8 \)
13. \( i = i + 1 \)
14. \( \text{goto 3} \)
15. \( \)
Control-flow graph

1. sum = 0
2. i = 1
3. if i > n goto 15
   T
   15.
   F
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
Common subexpression elimination

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
10a. t7 = t3 * t3
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
Copy propagation

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
10a. t7 = t3 * t3
11. t8 = sum + t7
12. sum = t8
12a. sum = sum + t7
13. i = i + 1
14. goto 3
15.
Invariant code motion

1. \( \text{sum} = 0 \)
2. \( i = 1 \)
2a. \( t1 = \text{addr}(a) - 4 \)
3. \( \text{if } i > n \text{ goto 15} \)
4. \( t1 = \text{addr}(a) - 4 \)
5. \( t2 = i * 4 \)
6. \( t3 = t1[t2] \)
10a. \( t7 = t3 * t3 \)
12a. \( \text{sum} = \text{sum} + t7 \)
13. \( i = i + 1 \)
14. \( \text{goto 3} \)
15. \( \)
Strength reduction

1. sum = 0
2. i = 1
2a. \( t_1 = \text{addr}(a) - 4 \)
2b. \( t_2 = i \times 4 \)
3. if \( i > n \) goto 15
5. \( t_2 = i \times 4 \)
6. \( t_3 = t_1[t_2] \)
10a. \( t_7 = t_3 \times t_3 \)
12a. \( \text{sum} = \text{sum} + t_7 \)
12b. \( t_2 = t_2 + 4 \)
13. \( i = i + 1 \)
14. goto 3
15.
1. sum = 0
2. i = 1
2a. t1 = addr(a) - 4
2b. t2 = i * 4
2c. t9 = n * 4
3. if i > n goto 15
   3a. if t2 > t9 goto 15
6. t3 = t1[t2]
10a. t7 = t3 * t3
12a. sum = sum + t7
12b. t2 = t2 + 4
13. i = i + 1
14. goto 3a
15.
1. \( \text{sum} = 0 \)
2. \( i = 1 \)
2a. \( t1 = \text{addr}(a) - 4 \)
2b. \( t2 = i \times 4 \)
2c. \( t9 = n \times 4 \)
3a. if \( t2 > t9 \) goto 15
6. \( t3 = t1[t2] \)
10a. \( t7 = t3 \times t3 \)
12a. \( \text{sum} = \text{sum} + t7 \)
12b. \( t2 = t2 + 4 \)
13. \( i = i + 1 \)
14. goto 3a
15. \( \)
1. sum = 0
2. i = 1
2a. t1 = addr(a) - 4
2b. t2 = i * 4
2d. t2 = 4
2c. t9 = n * 4
3a. if t2 > t9 goto 15
6. t3 = t1[t2]
10a. t7 = t3 * t3
12a. sum = sum + t7
12b. t2 = t2 + 4
14. goto 3a
15.
Dead code elimination

1. \( \text{sum} = 0 \)

2. \( i = 1 \)

2a. \( t1 = \text{addr}(a) - 4 \)

2d. \( t2 = 4 \)

2c. \( t9 = n \times 4 \)

3a. \( \text{if } t2 > t9 \text{ goto 15} \)

6. \( t3 = t1[t2] \)

10a. \( t7 = t3 \times t3 \)

12a. \( \text{sum} = \text{sum} + t7 \)

12b. \( t2 = t2 + 4 \)

14. \( \text{goto 3a} \)

15. \( \)
Final optimized code

1. \( \text{sum} = 0 \)
2. \( t1 = \text{addr}(a) - 4 \)
3. \( t2 = 4 \)
4. \( t4 = n \times 4 \)
5. \( \text{if } t2 > t4 \text{ goto 11} \)
6. \( t3 = t1[t2] \)
7. \( t5 = t3 \times t3 \)
8. \( \text{sum} = \text{sum} + t5 \)
9. \( t2 = t2 + 4 \)
10. goto 5
11.

unoptimized: 8 temps, 11 stmts in innermost loop
optimized: 5 temps, 5 stmts in innermost loop

1 index addressing 2 index addressing
1 multiplication 3 multiplications
2 additions 2 additions & 2 subtractions
1 jump 1 jump
1 test 1 test
1 copy 1 copy
1. sum = 0
2. t1 = addr[a] - 4
3. t2 = 4
4. t4 = 4 * n
5. if t2 > t4 goto 11
6. t3 = t1[t2]
7. t5 = t3 * t3
8. sum = sum + t5
9. t2 = t2 + 4
10. goto 5
11.
n = 1; k = 0; m = 3;

read x;

while (n < 10) {
    if (2 + x \geq 5) k = 5;
    if (3 + k == 3) m = m + 2;
    n = n + k + m;
}

General code motion
1. n = 1; 2. k = 0; 3. m = 3;

4. read x;

5. while (n < 10) {

6. if (2 * x ≥ 5) 7. k := 5;

8. if (3 + k == 3) 9. m := m + 2;

10. n = n + k + m;

11. }

- Invariant within loop and therefore moveable
- Unaffected by definitions in loop and guarded by invariant condition
- Moveable after we move statements 6 and 7
- Not moveable because may use def of m from statement 9 on previous iteration
n = 1; k = 0; m = 3;
read x;
while (n < 10) {
    if (2 * x ≥ 5) k = 5;
    if (3 + k == 3) m = m + 2;
    n = n + k + m;
}

n = 1; k = 0; m = 3;
read x;
if (2 * x ≥ 5) k = 5;
t1 = (3 + k == 3);
while (n < 10) {
    if (t1) m = m + 2;
    n = n + k + m;
}
n = 1; k = 0; m = 3;
read x;
if (2 * x \geq 5) k := 5;
t1 = (3 + k == 3);
if (t1)
    while (n < 10) {
        m = m + 2;
        n = n + k + m;
    }
else
    while (n < 10)
        n = n + k + m;

Specialization of while loop depending on value of t1
(Global) common subexpr elim

z = a * b
r = 2 * z

q = a * b

u = a * b
z = u / 2

w = a * b

Can be eliminated since \( a * b \) is available, i.e., calculated on all paths to this point.

Cannot be eliminated since \( a * b \) is not available on all paths reaching this point.
(Global) common subexpr elim

Ensure $a \times b$ is assigned to the same variable $t$ so it can be used for the assignment to $u$. 
Copy propagation

We can then forward substitute \( t \) for \( z \)…
Dead code elimination

...and eliminate the assignment to z since it is now dead code.
What else can we do?

t = a * b
r = 2 * t

w = a * b

u = t
z = u / 2

q = t
Partial redundancy elimination

We can compute $a \times b$ on paths where it is not available…

Then eliminate the now fully redundant computation of $a \times b$